

Final Report

Nabeel Amir

School of Computer Science and Informatics

Supervised by Frank C. Langbein

Moderated by Yu Kun Lai

Student Number: c1112357

Abstract

This report looks into the background, design, implementation and testing of creating a racing game for Android devices, that uses data from actual locations around the world for the map. The report also discusses the problems faced, my reflection on the project as a whole, as well as how the game can be improved in the future.

Acknowledgements

I would like to thank the following people for their assistance in completing this project; my supervisor Frank C. Langbein for his vital support, guidance and feedback, Mario Zechner who wrote the book 'Beginning Android Games', which guided me through the game development process and is the creator of the LibGDX framework, which I used to create the game. I would also like to thank all the contributors of LibGDX, without them it would have made the job of creating the game so much more difficult. I would like to thank Michal Migurski, for providing the JSON files I used to create the maps and Yu Kun Lai for providing constructive feedback of my initial report. Finally, I would like to thank Mark Whelton and Paul Schwann from BT, for providing me with additional ideas on how to improve this report.

Contents

1.	Introduction	6
1.1.	Goals of project	6
1.2.	Target Audience and Beneficiaries.....	6
1.3.	Scope of project	7
1.4.	Approach	7
1.5.	Assumptions	7
1.6.	Summary of outcomes	7
2.	Background	8
2.1.	Android Development	8
2.2.	Existing Applications.....	8
2.2.1.	Micro Machines	9
2.2.2.	Real World Racing	10
2.2.3.	Retro Racing	11
2.3.	Selecting Features and Constraints.....	11
2.4.	Stakeholders.....	12
2.5.	Methods and Tools.....	12
2.5.1.	Framework	12
2.5.2.	Android SDK and Eclipse	14
2.6.	Theory	15
2.6.1.	Game Design	15
2.6.2.	Represent Roads	15
2.6.3.	Pathfinding.....	17
2.6.4.	Artificial Intelligence	18
3.	Specification.....	19
3.1.	Requirements	19
3.2.	Methodology.....	20
4.	Design.....	21
4.1.	UML Diagram.....	21
4.2.	User Interface.....	22
4.2.1.	Main Menu.....	22
4.2.2.	Race Select Screen	23
4.2.3.	Loading Game Screen	24

4.2.4.	Game Screen	25
4.3.	System Architecture	25
4.3.1.	LibGDX.....	25
4.3.2.	Camera	28
4.3.3.	Maps.....	28
4.3.4.	Graph	31
4.3.5.	Pathfinding.....	33
4.3.6.	Physics.....	33
4.3.7.	Artificial Intelligence	36
4.3.8.	Rendering.....	36
5.	Implementation	38
5.1.	Game Implementation	39
5.1.1.	Structure	39
5.1.2.	Game Description	40
5.2.	Problems	45
6.	Testing.....	48
6.1.	Software Verification	48
6.2.	User Testing.....	56
6.3.	Performance Testing	57
6.3.1.	Quadtree	57
6.3.2.	Pathfinding.....	58
7.	Future Work.....	64
8.	Conclusion.....	66
9.	Reflection	67
10.	References	68

1. Introduction

1.1. Goals of project

The ultimate goal of this project is to create a top-down racing game for Android devices. The game itself will involve a race against other computer controlled opponents from one side of the map to the other. The unique part of the game is that the maps will be generated from data (such as roads, paths and buildings) retrieved from OpenStreetMap, which is essentially the open source version of Google Maps. This will mean the game will allow the user to race anywhere around the world.

The video game industry is worth billions of pounds and this amount increases every year; no other sector has experienced the same explosive growth as this industry. Smartphones are becoming much more widely available and 36% of gamers play games on their smartphone (Association, The Entertainment Software, n.d.).

Like the majority of individuals, especially around my age, have grown up with video games in their lives. The sense of enjoyment and escapism they provide is not comparable to other forms of entertainment, such as TV and Film. The interactivity of games and goal they provide can make players addicted. The impact it has had on us is phenomenal and for a very long time I have wanted to develop a game that has the same impact that other games have had on me and now is the time to do it.

Developing on the Android platform will mean I have to program in JAVA and get to grips with the Android SDK. I will also have to learn the fundamentals of game programming and stick to the lifecycle associated with developing games. The Android SDK does not provide a comprehensive set of tools to create a game straight away; it does require a lot of extra work. I could potentially create my own framework or perhaps use an existing one, to ease the process and produce software that is appealing and efficient; this will be discussed in more detail later on.

There are several problems that are required to be solved in order to have a successfully working game at the end, such as drawing the game objects on screen, how to retrieve and represent the map in order to have pathfinding and rendering that is visually appealing, how the artificial intelligence will work and how to interact with the game to make it enjoyable.

This report will go into a reasonable amount of detail about knowledge that is required beforehand to understand the project; it will go into some detail about how I intend to solve the problem, with justification. I will then go into detail about how I implemented the project and problems that I faced, then an evaluation of the project and some of the methods I used. I will then discuss how I could improve the project, draw some conclusions and reflect on the project as a whole.

1.2. Target Audience and Beneficiaries

The application is targeted towards teenagers who enjoy playing racing games and want a new twist on the racing game genre. I will keep this in mind when designing the user interface, drawing the graphics and creating the physics.

Another audience that could have interest in the project is someone who is beginning to look at game development, particularly racing games and needs guidance on

how to start; then implement more advanced features such as artificial intelligence, pathfinding and various other techniques. Individuals with higher knowledge of game development could also have an interest in the game, to see my approach of including real world locations in the game.

I would consider myself as a beneficiary as I have improved a lot of my existing programming skills and gained a lot of additional knowledge of various topics such as Object Orientated Programming.

1.3. Scope of project

The scope is everything that needs to be accomplished in order for me to have a successful project. There is a lot to do when creating a game; there are many algorithms and theories and there are different approaches to each one and choosing the right one is vital. This is discussed in more detail in the design section, with justification of each decision.

1.4. Approach

I shall approach this project in a systematic and methodical manner. I will make heavy use of object orientated programming to ease the work process. Before I began the project I took steps to ensure that everything I was planning on implementing was designed first, whether it be sketches with a pencil and paper (see appendix), or using tools on the computer such as Dia to draw class diagrams. Before I dived into programming, I made some test programs, to get to grips with everything.

1.5. Assumptions

My main assumption is that the end product is a prototype, although I will try my best to make the project visually appealing, I will concentrate more on the coding aspect of it. I have assumed that the Android device the end game will be played on has a touch screen. I have assumed that the user will not want to have AI that wins all the time, therefore I will have to find some way of making the game winnable.

1.6. Summary of outcomes

At the end of this project I aim to have the following features implemented in the game:

- Game should be compatible on at least Android 2.2, be able to resize and fit various screens sizes and use the touch screen to control the car;
- Relatively realistic life physics i.e. friction, acceleration etc. in a 2D, top-down perspective;
- Map should be created using real data from OpenStreetMap, the data should contain data about the roads and buildings;
- Computer opponents should have some form of artificial intelligence for pathfinding and for interacting with other cars. They should not always race optimally, to give the human player a chance to win;
- Create an intuitive user interface, that is simple to understand and appealing.

2. Background

2.1. Android Development

Before actually discussing Android development, I will briefly talk about the Android platform.

Android is an operating system based on the Linux kernel and designed predominantly for touchscreen mobile devices such as smartphones and tablets. It was first developed by Android Inc., which Google backed financially and later bought in 2005. Android was officially revealed in 2007 along with the founding of the Open Handset Alliance - a group of hardware, software, and telecommunication companies keen to progress open standards for mobile devices (Android, n.d.).

Android provides a rich application framework that allows anyone to build innovative apps and games for mobile devices. Android development is the process of creating applications for the Android operating system. They are usually developed in the JAVA programming language using the Android SDK, but there are many other development tools available to use. As of July 2013, more than 1 million applications have been developed for Android (Warren, 2013).

Android devices can be interacted with using touch inputs, such as swiping, tapping and pinching. There is also internal hardware such as accelerometers, gyroscopes and proximity sensors that also allow a unique form of interaction.

Android's source code is released by Google under open source licenses, which has encouraged a large community of developers and enthusiasts to use the open-source code as a foundation for community-driven projects, which add new features for advanced users (Amadeo, 2013).

Android includes support for high performance 2D and 3D graphics with the Open Graphics Library (OpenGL) which is a cross-platform graphics API that specifies a standard software interface for 3D graphics processing hardware (Android, n.d.).

2.2. Existing Applications

There are many racing games that exist in today's market which have been around for many years. They range from ultra-realistic simulations such as Gran Turismo, which take into account many factors to make the game as close to the real thing as possible, to the casual arcade racer such as Mario Kart. I used my personal knowledge of the games to discover existing applications and search engines to discover more of them. The ones I had not heard of, I tried out; I did not restrict it to just mobiles. I also looked at games on other platforms as well, however I focused more on top down racing games.

2.2.1. Micro Machines



Micro Machines is one of the first top down racing games and at the time of its release, received very good reviews. It involves racing miniatures of various vehicle types across a particular terrain found around the house; for example, on a desk, in the sandpit and even a snooker table. The game is available to play on NES, Amiga, PlayStation 2, Xbox, GameCube, Game Boy Advance and various other consoles (Servo, 2002). I will aim to mimic its style and simplicity.

2.2.2. Real World Racing



Real World Racing is a top-down racing game using satellite images that lets you race through the world's biggest cities such as London, Paris, Berlin and Rome (Playstos, n.d.). The game is available to download on PC via the Steam Store. The maps have a similar system to what I will use, i.e. real life data, however the maps in the game have been pre-generated and are restricted to the cities that the developer decided to use, whereas my application will generate the map at run time and can use data from anywhere around the world. The gameplay will also be different. In this application there is one route to the finish line, whereas in my application it is up to the user to choose the best route.

2.2.3. Retro Racing



Retro Racing is a fun and simple top down racing game available to download on various platforms including Android (Mr Qwak, 2014). I will take a lot of inspiration from this game for the interface and the control layout. The main menu is attractive and changes colour depending on the colour of the car you choose, the buttons and text animate when pressed. The physics are relatively simple and the graphics clean and appealing. The controls are straightforward and I like the fact that they are semi-transparent so it does not completely obstruct the view of the game.

2.3. Selecting Features and Constraints

After reviewing the games, I now have to identify all the features that I would implement. I have gained a lot of knowledge from my research; despite being an enthusiastic racing gamer, I still managed to find games that I had not previously heard of such as Real World Racing. All the games I tested were final products and due to the timescale, I will not be able to achieve a project that is completely finished and can be put out to the general public to play without any hiccups.

From all the applications I tested, I identified the following core functions which I will implement:

- Physics - the car will have to be able to move using the front two wheels for steering and depending on the drive of the system, whether power is applied to all wheels, just the front two or the back two. This physics only needs to be simulated in 2D in a top down perspective. When a car hits an object, then the relevant forces should be applied to the car and object, in the appropriate direction. Appropriate collision detection also needs to be used to identify these crashes;
- Maps - they will need to have a start and finish line, with clearly marked out roads and potentially, direction arrows to assist the user in finding the finish line;
- Attractive user interface - the interfaces of the games tested are very appealing; each one also has a heads up display showing various in-game parameters, such as the time currently elapsed.
- Artificial Intelligence - all the games tested had some form of AI to make the game more challenging and fun to play. One thing to note was that each of the cars in the game were not as powerful as the player's cars, to make the game winnable by the player;
- Pathfinding - each game employed some form of pathfinding, the actual method used is unknown, but it could have been calculated at runtime or perhaps during compile time and saved to a file;
- Graphics - this is very important to make the game more appealing;
- Camera - all top down games have the same camera, the player in the centre and a view of the world around it.

There are various constraints on the project that will limit the quality of the work produce, the first major one is the time constraint; because of this, the final product will be more of a prototype, rather than a product ready for distribution.

2.4. Stakeholders

A stakeholder is anyone that has an interest in the project. My main stakeholders are my users, who will probably be teenagers who have a keen interest in racing games. Once the application has been fully completed, it will be distributed on the Android Market (PlayStore) for those users to download. As the developer I am also a stakeholder.

2.5. Methods and Tools

2.5.1. Framework

To create the application I will use various tools to ease the process. The first thing to do is to decide how I am going to approach the project. There are numerous ways of developing games for Android phones: I could do the whole thing in pure JAVA using the Android SDK and create my own framework to handle graphics and other aspects of the game, use an existing framework or even a fully-fledged game engine with lots of high level methods. Due to the time constraint, I will not be creating a framework from scratch, and also because of "reinventing the wheel" - if there is something out there that does what you want, use it.

There is an extensive list of engines and frameworks to use, so it is a good idea to distinguish between them. A framework gives the developer control over every aspect of the game development environment. However, this comes at a price of having to figure out your own way of doing things (for example, how you organise your game world, how you handle screens and transitions, and so on). An engine is more streamlined for specific tasks. It dictates how you should do things, giving you easy-to-use modules for common tasks and a general architecture for your game. The downside is that the game might not fit the solutions the engine offers the developer. Most often than not, they'll have to modify the engine itself to achieve their goals, which may or may not be possible depending on whether the source code is available (Zechner, 2012). With this crucial difference in mind, for my purposes, I think it would be best to use a framework rather than an engine.

Choosing a framework will be difficult, so before I decide which one to use I have made a list of requirements that should be met in order for me to use it:

- The language is not too important, however I would prefer JAVA or C;
- It must be free to develop using it; paying for it to be publishing is ok, but I would prefer the whole thing to be free;
- Cross platform is an advantage, however, it may mean that it will work well on one platform, but there may be issues or bugs on another. So if it worked natively on Android it would be better;
- Documentation is very important, there must be an extensive list of documentation to describe how to utilise the various functions.

With those in mind, below is a table comparing the features of Android game frameworks (All Android Game Engines, n.d.):

Framework Name	Language	Price	Cross-Platform	Documentation	Comments
LibGDX (LibGDX, 2014)	JAVA	Free	Yes	Yes, also has lots of demos and tutorials	
Slick-AE (Ninja Cave, n.d.)	JAVA	Free	Yes	Yes	Built on top of LibGDX
AndEngine (AndEngine, n.d.)	JAVA	Free	No, just Android	No, but there are demos, tutorials and books	
BatteryTech SDK (BatteryTech, n.d.)	C++	Free to develop with, but a one-time license must be purchased	Yes		
Emo Framework (Emo, n.d.)	C, C++, Objective-C	Free	Yes, but only Android and iOS	Well detailed documentation	Does not look as though it has been updated for a while

Each of the frameworks have their strengths and weaknesses. The reason I would like for it to be cross-platform, is because it will allow for quick deployment and testing, especially if it can be compiled and run on a desktop. Taking everything into account, the one which I will use is LibGDX, because of its ability to be cross-platform, it is open source and it has a very active forum (LibGDX, 2014). With LibGDX you code everything in Java and then it ports the code to different platforms, for example with the Web, it translates the java code to JavaScript.

LibGDX has the following features that I will be utilising to make my game:

- Texture - wraps a standard OpenGL ES texture;
- Shape rendering - Renders points, lines, rectangles, filled rectangles and boxes, used for debugging;
- Orthographic Camera - A camera with orthographic projection;
- SpriteBatch - Draws batched quads using indices;
- TextureAtlas - Loads images from texture atlases created by TexturePacker;
- BitmapFont - Renders bitmap fonts;
- JSOM writer and reader;
- XML writer and reader;
- Abstractions for mouse and touch-screen, keyboard for input;
- Vector2 class - Encapsulates a 2D vector;
- Box2D - control physics;
- File system abstraction for all platforms;
- Scene2D - high level objects to create user interfaces;
- Net interface - allow connection to the internet.

LibGDX has a lot of high level objects and the ability to go as low down as you want. However, it will still require a lot of work to complete it. Below is a list of things that I will have to implement:

- Figure out how to organise all the objects, cars, world, map etc.;
- LibGDX has a feature that allows it to create a nice user interface; I will have to utilise it to create mine;
- Utilise the net interface to fetch map data, then process that data to be displayed in the game;
- Use Box2D to create the objects that are collidable, such as the car and wheels;
- Figure out how to only render the objects on screen;
- Figure out how to represent the roads and how to do pathfinding;
- Figure out how to do AI.

2.5.2. Android SDK and Eclipse

To develop the Android applications, using LibGDX, I require interaction with the Android SDK, there is a neat download on the Android developers website which includes everything required to start developing applications, which includes Eclipse along with the ADT plugin, Android SDK Tools, Android Platform-tools, the latest Android platform and the latest Android system image for the emulator already installed and ready to use (Android, n.d.). I will then have to setup all the LibGDX libraries as well, there is a tool called LibGDX

Project Setup that gets all the necessary files to create the application, with some additional demo classes and then it can be imported into Eclipse (Ribon, 2012).

2.6. Theory

There are various things that need to be considered when making the game, below is theory associated with the various tasks.

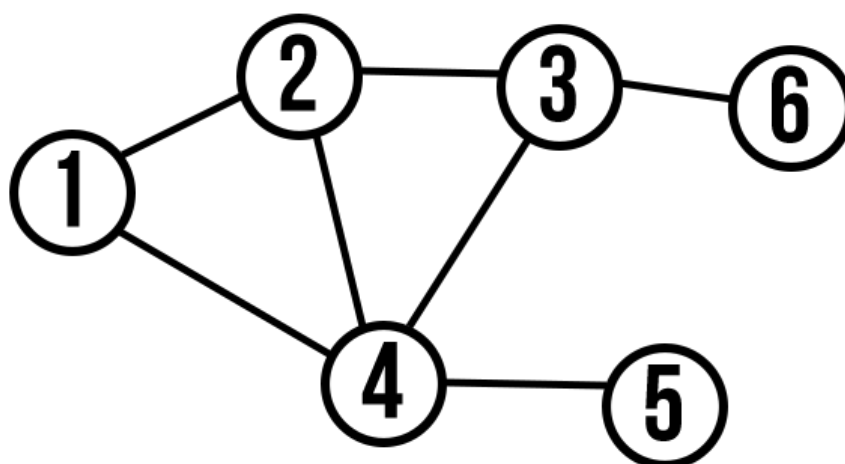
2.6.1. Game Design

Game design is the process of designing the content and rules of a game in the development stage. It involves the design of gameplay, environment, storyline, and characters. Video game design requires artistic and technical competence as well as writing skills (Rouse III , 2001).

2.6.2. Represent Roads

Finding an adequate way of representing the roads is important, because this will then be used to draw the road and to carry out pathfinding.

There is a very simple way of representing a road network and that is as a graph. A graph is made up of two sets called Vertices and Edges. The Vertices are drawn from some underlying type, and the set may be finite or infinite. Each element of the Edge set is a pair consisting of two elements from the Vertices set. Graphs are often depicted visually, by drawing the elements of the Vertices set as circles, and drawing the elements of the edge set as lines (also called arcs) between the circles. There is an arc between v_1 and v_2 if (v_1, v_2) is an element of the Edge set (Sheard, n.d.). Below is a very simplified diagram of a graph:

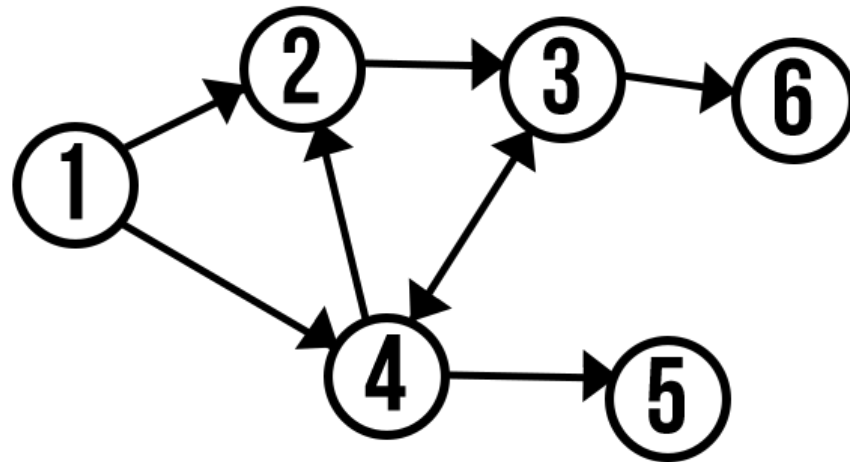


The graph has the following features:

- The Vertices set = $\{1, 2, 3, 4, 5, 6\}$
- The Edge set = $\{(1, 2), (1, 4), (2, 4), (2, 3), (3, 4), (3, 6), (4, 5)\}$

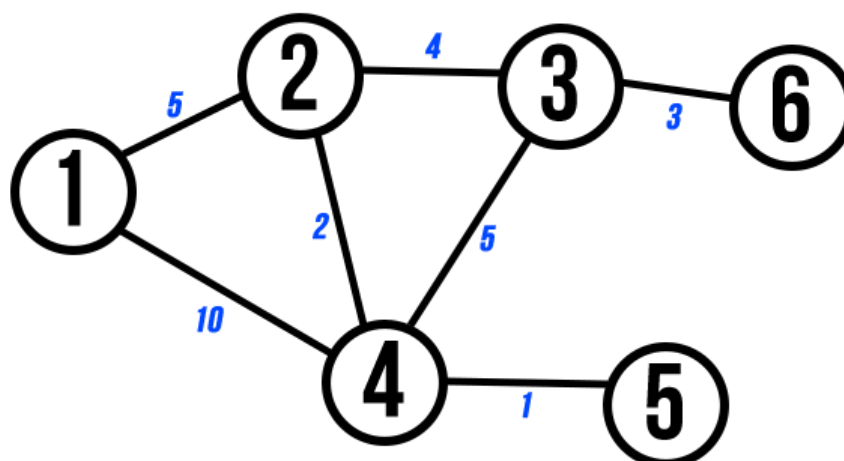
There are various types of graphs, below is a brief description of each one:

- Undirected Graphs - the graph above is an example of an undirected graph. They are drawn with straight lines between the vertices and the order of them does not matter. For example the first edge (1,2) can be written as (2,1) and the meaning is unchanged.
- Directed Graphs - the order of the vertices and edges matter, arrows are usually drawn to show the direction. Below is a diagram of this



The graph has the following features:

- The Vertices set = {1,2,3,4,5,6}
- The Edge set = {(1,2),(1,4),(4,2),(2,3),(3,4),(4,3),(3,6),(4,5)}
- Cyclic graphs - a directed graph with at least one cycle. A cycle is described as a path along the directed edges from a vertex onto itself. The diagram above showing the directed graph has a cycle 4->2->3->4.
- Weighted graphs - a weighted graph is graph which has its edges labelled with integers or floats that indicate the cost of going down a particular path. Below is an example of this:



- The Vertices set = {1,2,3,4,5,6}
- The Edge set = {(1,2,5),(1,4,10),(2,4,2),(2,3,4),(3,4,5),(3,6,3),(4,5,1)}

2.6.3. Pathfinding

Pathfinding is a major aspect of the game; it needs to be used by the AI to find the shortest path to the goal and also show the user a possible route they could take to get to the finish line. The first step is to pick a start and goal node from the graph and then the adjacent nodes are continually explored until the goal is found. The fashion that the nodes are explored must be efficient; the idea is to find the shortest path to the goal in the quickest time.

One such algorithm is a Breadth First Search; it is an uninformed search method that aims to expand and examine all nodes of a graph or combination of sequences by thoroughly searching through every solution. It does not use a heuristic algorithm (Anon., n.d.).

From the position of the algorithm, all child nodes obtained by expanding a node are added to a first in, first out queue. Nodes that have not yet been explored are stored in a queue called "open" and then once examined are placed in the structure called "closed". Below is the pseudocode (Alpcentauri, n.d.):

1. Declare two empty lists: Open and Closed.
2. Add Start node to our Open list.
3. While our Open list is not empty, loop the following:
4. Remove the first node from our Open List.
5. Check to see if the removed node is our destination.
6. If the removed node is our destination, break out of the loop, add the node to our Closed list and return the value of our Closed list.
7. If the removed node is not our destination, continue the loop (go to Step 8).
8. Extract the neighbours of our above removed node.
9. Add the neighbours to the end of our Open list

Another algorithm is A*, it works in the following way:

1. Begin at the starting point and add it to an open list (similar to the open list in the breadth first search) of nodes to be considered;
2. The algorithm looks at all the reachable nodes adjacent to the starting point, adding them to the open list too. For each of these nodes, save the starting point as its "parent node". This parent node is important when tracing the path back to the goal;
3. Drop the starting node A from the open list, and add it to a closed list of nodes (similar to the closed list in the breadth first search).

The next step is to choose which one of the adjacent nodes on the open list and more or less repeat the earlier process, but how to pick the best node, the one with the lowest scoring cost.

The key to determining which node to use when figuring out the path is the following equation:

$$F = G + H$$

Where:

- G - the movement cost to move from the starting point to a given node in the graph, following the path generated to get there.
- H - the estimated movement cost to move from that given node on the graph to the final destination. This is often referred to as the heuristic; the reason why it is called that is because it is a guess. There are various ways of calculating this.

To continue the search, the lowest F score node is chosen from all those that are on the open list. Then the following is done with the selected node:

4. Drop it from the open list and add it to the closed list;
5. Check all of the adjacent nodes. Ignoring those that are on the closed list, add nodes to the open list if they are not on the open list already. Make the selected node the "parent" of the new nodes;
6. If an adjacent node is already on the open list, check to see if this path to that node is a better one. In other words, check to see if the G score for that node is lower if we use the current node to get there. If not, don't do anything. On the other hand, if the G cost of the new path is lower, change the parent of the adjacent node to the selected node. Finally, recalculate both the F and G scores of that node (Lester, 2005).

2.6.4. Artificial Intelligence

Artificial Intelligence is the branch of Computer Science concerned with making computers behave like humans. The term was coined in 1956 by John McCarthy. Artificial Intelligence includes the following areas of specialisation:

- games playing - programming computers to play games against human opponents;
- expert systems - programming computers to make decisions in real-life situations;
- natural language - programming computers to understand natural human languages;
- neural networks - systems that simulate intelligence by attempting to reproduce the types of physical connections that occur in animal brains (Anon., n.d.).

In racing games, the simplest AI would look at one component of the car to control and that is the steering. The car would also have to do some form of pathfinding to make a decision on where to go, whilst avoiding any static and dynamic obstacles in the way. More advanced AI would also look at the power applied to the wheels to ensure the car control is as realistic as possible.

The most advanced AI use neural networks. The game Collin McRae Rally 2 is one of the first applications of neural networks in computer games. It is responsible for keeping the computer player's car on track while letting it negotiate it as quickly as possible and uses the multi-layered perceptron model. The neural network's input parameters are information such as: curvature of the road's bend, distance from the bend, type of surface, speed and the vehicle's properties. It is up to the neural network to generate output data to be passed further to the physical layer module, that data being selected in such a way that the car travels and negotiates obstacles or curves at a speed optimal for the given conditions. Thanks to this, the computer player's driving style appears highly natural, especially when compared to other forms of AI. The computer can avoid small obstacles, cut bends, begin turning appropriately soon when on a slippery surface etc (Software Developer's Journal, 2013).

3. Specification

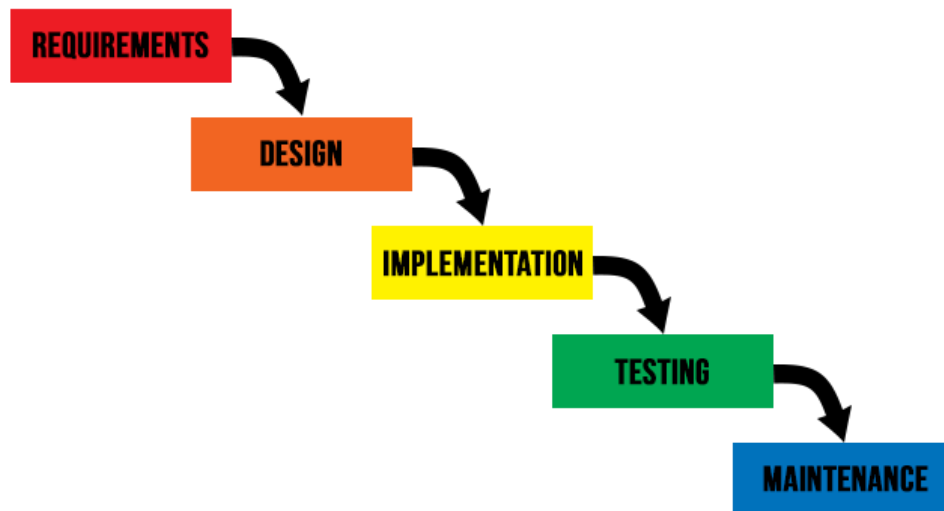
3.1. Requirements

After extensive research, I have made a list of the following core requirements (what the system should do):

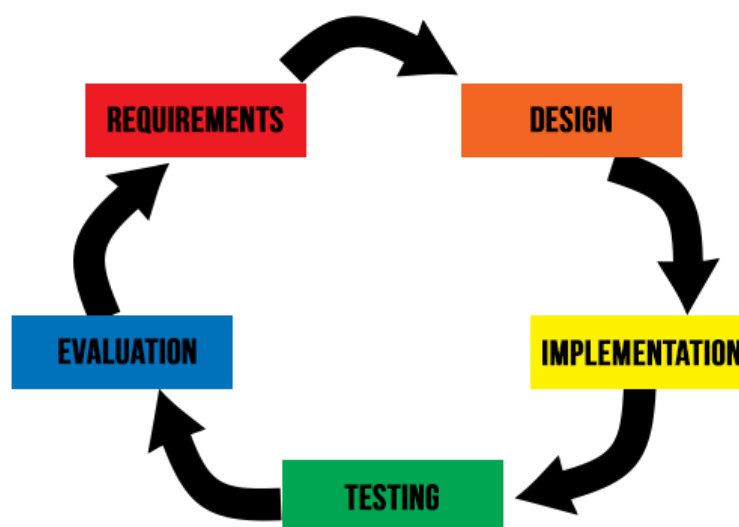
- Physics - the car will have to be able to move using the front two wheels for steering and depending on the drive of the system, whether power is applied to all wheels, just the front two or the back two. This physics only needs to be simulated in 2D in a top down perspective. When a car hits an object it, such as another car or building, it will bounce off it.
- Maps - the maps will be generated using data from OpenStreetMap, this will be processed at runtime when the user selects where they want to race. The data should at least have the road data and if possible other features such as buildings and location of greenery.
- User interface - the interface will consist of various screens, they must all be attractive and appealing; below is a list of them:
 - Main menu - this should be the first screen shown and is the main access point to other screens;
 - Settings screen - change various in game parameters;
 - About screen - simple screen having details about the creation of the game;
 - Race select screen - a screen that allows the user to choose where they want to race;
 - Game screen - where the user actually plays the game;
 - Pause screen - shown when the user wishes to pause the game
- Artificial Intelligence - should have some additional cars that the player can race against to make the game more entertaining for the user.
- Pathfinding - The shortest path from the start to the finish line will be determined at runtime, once the map has been generated. It forms the core of the AI and needs to be fast and efficient.
- Graphics - I will draw the majority of the graphics myself and will try to make them as appealing as possible.
- Camera - to make my game consistent with the top down genre, the camera will follow the player and render everything that is in view.
- Adapt - the game should be able to adapt to various Android screen sizes.
- Consistent - the game should use an attractive colour scheme and use clear fonts
- Small download - the size of the file is important, Android users do not like unnecessary large files on their phone; I aim to make the game under 25MB.
- Run well - the game must run with a minimum of 30 frames per second.

3.2. Methodology

The original software development model I was going to use was the standard Waterfall Model, where each phase must be completed fully before the next phase can begin. Below is a diagram showing this:



However, this eventually broke down and I ended up using a more iterative and incremental approach, where software is developed using cycles. Although at the beginning I had a design and requirements, when I began implementing the code, I found myself continually going back to the design, altering them, then implementing it, then testing, then going back to the design (ISTQB Exam Certification, n.d.). Below is a diagram showing roughly this procedure:

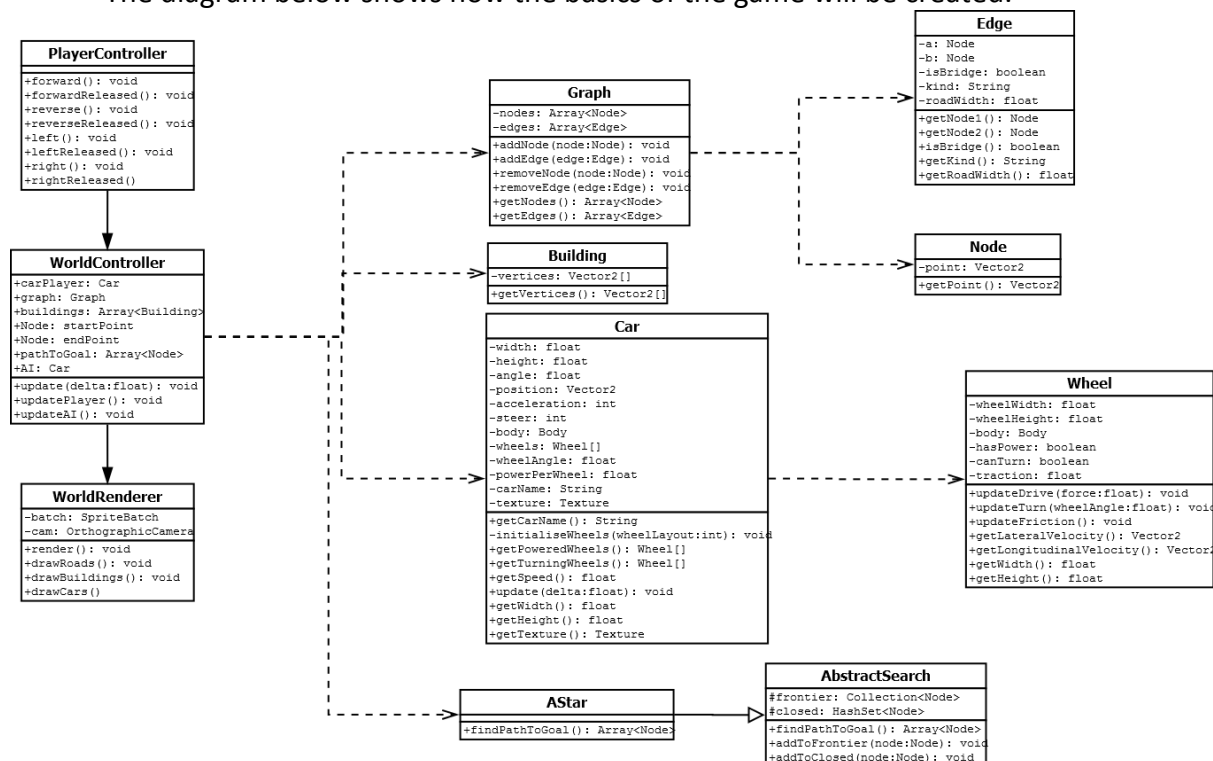


4. Design

In this section I will look at how the final game will look and go into a little detail of how it may function.

4.1. UML Diagram

The diagram below shows how the basics of the game will be created.



The diagram has been heavily simplified to make it clear what is going on. The PlayerController, WorldController and WorldRenderer are parts of an architectural pattern known as the Model-View-Controller (MVC). The pattern divides the gameplay into three interconnected parts:

- Model - represents knowledge, it handles all the data and logic;
- View - is a visual representation of the model;
- Controller - takes the users input and updates the model accordingly.

In my case the Model is the WorldController, which has many other models such as the Car and Building, the View is the WorldRenderer, which takes the models from the WorldController to be updates and the Controller is the PlayerController, which takes input from the user and updates the models, such as the car (Atwood, 2008).

4.2. User Interface

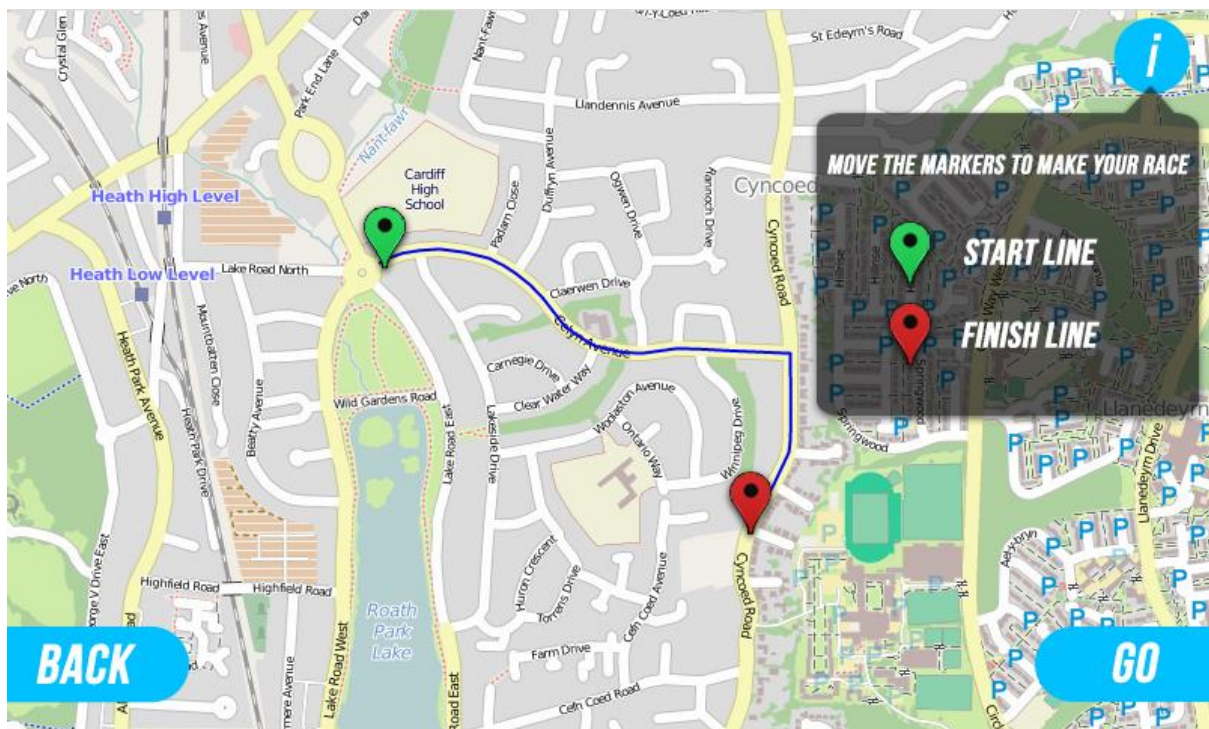
Below are sketches of the user interface, the majority of the assets have been made by me, the font used is Bebas Neue.

4.2.1. Main Menu



This is the main menu and is the first screen user is greeted with; from there they can interact with the rest of the application. It is simple and only has three buttons that allow it to go to various other parts of the game. I have chosen to go for a colour scheme consisting of a sky blue colour, with dark greys and white. The game will be called World Racing.

4.2.2. Race Select Screen



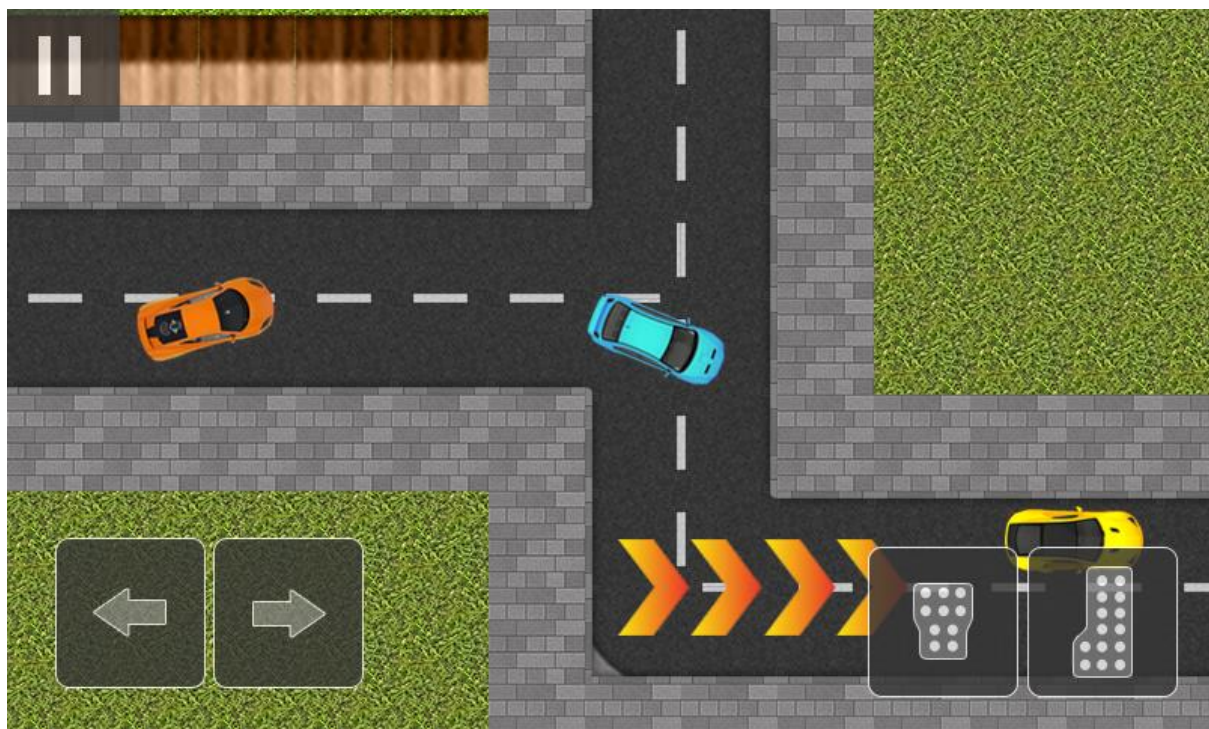
Above is the design for the race select screen. It features two markers that can be moved around on the map (YourNavigation, 2014) to change the start and finishing line, a possible route is also shown. It also has an information button which can be shown and hidden when the “i” button is pressed in the top right hand corner. It also has a button to go back to the main menu and a button called go, which can be pressed once the user has decided where to race.

4.2.3. Loading Game Screen



This is the loading screen, it is shown once the user has selected where to race. Its purpose is to give feedback to the user that the game is fetching and creating the map. The wheel like object on the top right corner of the screen rotates to give visual feedback.

4.2.4. Game Screen



This is the actual game screen; there are buttons on screen to control the movement of the car and also a pause button. They have purposely been made semi-transparent, so they do not completely obstruct the game view. There are additional items such as the chevrons to indicate to the user where to drive to reach the goal.

All the images are stored in a TextureAtlas, this is where multiple images are combined as a single image. The reason this is done is because binding lots of textures is relatively expensive in Android, so it is ideal to store many smaller images on a larger image, bind the larger texture once, and then draw portions of it many times (Zechner, 2012). There is a special tool called libgdx-texturepacker-gui which does this (Ribon, 2012). There is a class in LibGDX framework that allows you to access the individual images using get methods.

4.3. System Architecture

4.3.1. LibGDX

Before I get into the nitty gritty details of the project I will briefly talk about how LibGDX works. As mentioned previously it is cross-platform. The advantage of this is that the code is written once then it can be deployed to multiple platforms, however there is some platform specific code that needs to be written, which are the starter classes.

For each platform that is targeted, a piece of code instantiates a concrete implementation of a class called Application Interface, which is provided by the back-end for the platform. The actual code of the application is located in a class that implements the ApplicationListener interface.

An ApplicationListener is called when the Application is created, resumed, rendering, paused or destroyed (see table below). All methods are called in a thread that has the OpenGL context current. Therefore graphic resources can be created and manipulated safely. The ApplicationListener interface follows the standard Android activity life-cycle and is emulated on the desktop accordingly.

An instance of this class is passed to the respective initialisation methods of each back-end's Application implementation (whether it be Android, iOS, Desktop etc.). The application will then call into the methods of the ApplicationListener at appropriate times.

These life-cycle events can be used by implementing the ApplicationListener interface and passing an instance of that implementation to the Application implementation of a specific back-end. From there on, the Application will call the ApplicationListener every time an application level event occurs (badlogic, 2013). A bare-bones ApplicationListener implementation may look like this:

```
public class MyGame extends ApplicationListener {
    public void create() {
    }

    public void resize(int width, int height) {
    }

    public void render() {
    }

    public void pause() {
    }

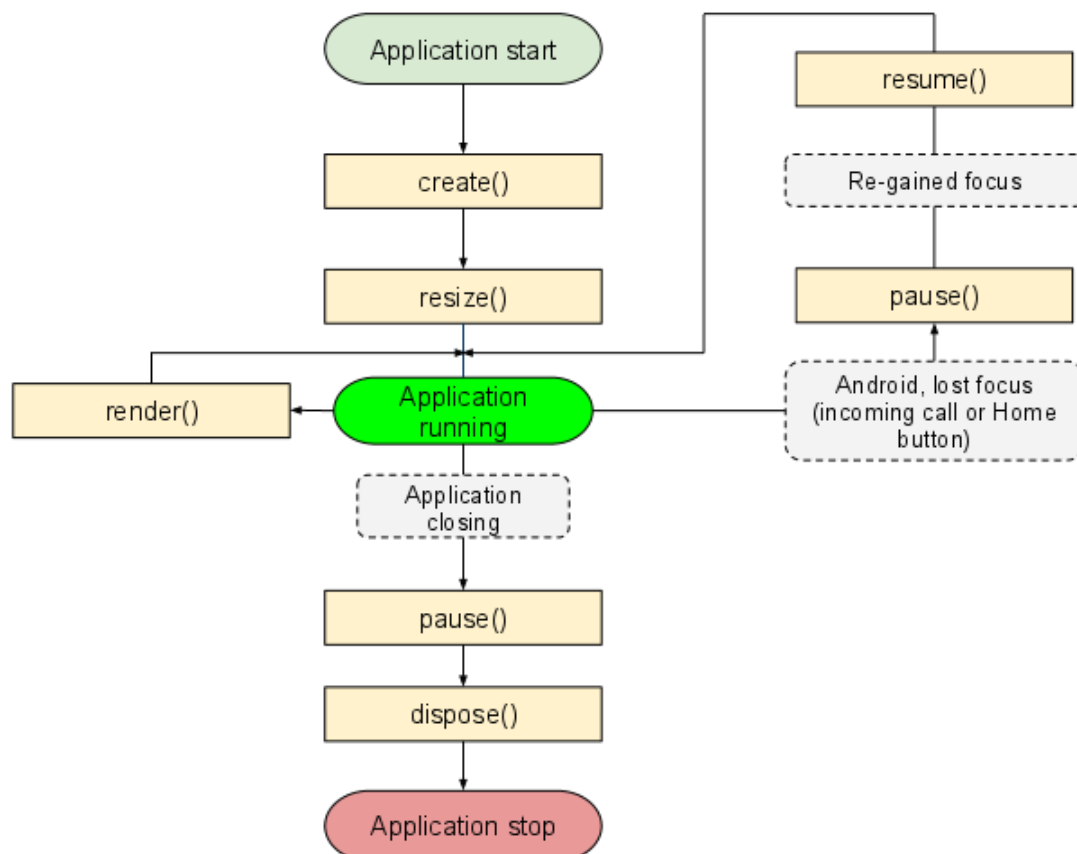
    public void resume() {
    }

    public void dispose() {
    }
}
```

Once passed to the Application, the ApplicationListener methods will be called as follows:

Method	Description
<code>create()</code>	Method called once when the application is created.
<code>resize(int width, int height)</code>	This method is called every time the game screen is re-sized and the game is not in the paused state. It is also called once just after the <code>create()</code> method. The parameters are the new width and height the screen has been resized to in pixels.
<code>render()</code>	Method called by the game loop from the application every time rendering should be performed. Game logic updates are usually also performed in this method.
<code>pause()</code>	On Android this method is called when the Home button is pressed or an incoming call is received. On desktop this is called just before <code>dispose()</code> when exiting the application.
<code>resume()</code>	This method is only called on Android, when the application resumes from a paused state.
<code>dispose()</code>	Called when the application is destroyed. It is preceded by a call to <code>pause()</code> .

The following diagram illustrates the life-cycle visually:



LibGDX has various modules that provide a means of interacting with the operating system on the different platforms; these can be accessed via static fields of the Gdx class. This is essentially a set of global variables that allows easy access to any module of LibGDX.

For example to access the audio module one can simply write the following:

```
AudioDevice audioDevice = Gdx.audio.newAudioDevice(44100, false);
```

Gdx.audio is a reference to the backend implementation that has been instantiated on application startup by the Application instance. Other modules are accessed in the same fashion, e.g. Gdx.app to get the Application, Gdx.files to access the Files implementation and so on (brutalbutler, 2014).

4.3.2. Camera

The camera in the game has a position and orientation in 3D space. It also has a viewport which defines the size and resolution of the final image and the coordinate system. These need to be defined in the game in order for it to work, as well as the type of projection (Zechner, 2012). There are two main types:

- Perspective projection - similar to our eyes, where for example an object far away seems smaller, than if it was close to you;
- Parallel (orthographic) projection - object will have the same size in the final image.

For my game I have chosen to use a camera 800 units on the x axis and 480 units along the y axis, the reason being is that most common Android devices have this resolution. After experimenting with my design I have decided to define a conversion between metres and pixels, which will help in rendering images correctly. The scale I have chosen is that 1 metre in real life represents 20 pixels on screen.

4.3.3. Maps

A major part of the game is figuring out how to display the real world map data into the game. Thankfully OpenStreetMap, which is as a collaborative project to create a free editable map of the world allows you to do this. You can get the data about a specific location as raster tiles, which are square bitmap graphics displayed in a grid arrangement (similar to how you see interactive maps online such as Google Maps); but more to my liking is vector tiles (OpenStreetMap, 2014), which are similar to the raster tiles, but instead of raster images the data returned is a vector representation of the features in the tile.

There are two major ways of obtaining the vector tiles; one way is in OSM format, you can obtain the data using a query similar to this:

```
GET /api/0.6/map?bbox=left,bottom,right,top
```

Where:

- left is the longitude of the left (westernmost) side of the bounding box;
- bottom is the latitude of the bottom (southernmost) side of the bounding box;
- right is the longitude of the right (easternmost) side of the bounding box;
- top is the latitude of the top (northernmost) side of the bounding box.

This gives a vast amount of data about a particular location in XML format, such as the location of a street as latitude and longitude coordinates, the street's name, the type of road and more.

However, the problem with doing it this way is that XML files are very bulky, it is not easily cacheable and it would require a lot of pre-processing to only get the data that is required (OpenStreetMap, 2013).

A better alternative is a service called Mapnik Vector tiles (Migurski, 2014), which gives data in GeoJSON (GeoJSON, 2014) format, which is essentially a format for encoding a variety of geographic data structures in JSON format. The code typically looks like this:

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [125.6, 10.1]
  },
  "properties": {
    "name": "Dinagat Islands"
  }
}
```

You query the data like this

<http://tile.openstreetmap.us/vectiles-type/z/x/y.json>

Where:

- type is **the what** data you are trying to retrieve, highroad (all the roads and paths in the area), building etc.;
- z is the zoom level;
- x is the x tile coordinate;
- y is the y tile coordinate.

This method of storing the data as zoom, x, y coordinates is very simple. It essentially has the world stacked at different zooms as a pyramid, at zoom level 0, the whole world is displayed [with only one tile at (0,0)] at zoom level 1, the tile at zoom level 0 is split into 4 (with more detail) and accessed like this, (0,0), (0,1), (1,0) and (1,1). This is done continually and depending on the source the zoom level can go up to 19 or higher (OpenStreetMap, 2014).

The advantages of this method is that firstly it is stored in JSON files which tend to be more lightweight than XML values and it also allows me to get objects that are relevant to me, if I want the roads I use the highroad type. Storing the map as tile coordinates and zoom levels also means that it is easily cacheable, which means the data can be downloaded quickly to display in the game.

The downside is, if I wish to get a large region at a high zoom, I will have to make multiple requests to fetch the data, this multiple data will then have to be stitched together to form a map; whereas if I were to get the map using the previous method, all the data would be contained in one file. Despite this disadvantage I have still chosen to use Mapnik Vector Tiles.

Another possibility, if I have too much trouble with the data, is to use image processing techniques to get map as image file and then threshold etc. to get the location of roads and buildings, then these can be inserted into the game.

After the map data has been retrieved the next problem is how to display this in the game. A typical road file in geoJSON format looks like this:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "geometry": {
        "type": "LineString",
        "coordinates": [[-122.275444, 37.811706], [-122.277832, 37.812634]]
      },
      "type": "Feature",
      "id": "6369500",
      "clipped": true,
      "properties": {
        "kind": "minor_road",
        "sort_key": -6,
        "is_link": "no",
        "is_tunnel": "no",
        "is_bridge": "no",
        "railway": null,

        "highway": "residential"
      }
    }
  ]
}
```

The most important part of the file is the coordinates, they are stored as pairs of numbers, the first number is the Longitude and the second is Latitude given in decimal degrees format, which represents a point on the Earth. Latitude lines run horizontally on the earth and range from -90 to 90, whereas Longitude lines are vertical and range from -180 to 180. These points are essentially on a sphere (the earth) they therefore have to be projected onto a 2D plane so that they can display accurately what the point looks like (Rosenberg, n.d.).

There are various projection methods including sinusoidal projection, orthographic projection, general perspective projection and more. However, arguably the most accurate is the Transverse Mercator. It is an adaptation of the standard Mercator projection; the property of it that made it useful is that it preserves angles and it is highly accurate. Most of OpenStreetMap uses the Transverse Mercator projection, so it would be unwise of me to adopt another method.

The function for the Transverse Mercator Projection takes the latitude and longitude and converts it to metres (Anon., 2013). This can then be used to display the objects in my games. The type of the coordinates are also important, as they dictate the shape of the object to display, a LineString is an array of two or more points (most of the roads are encoded like this) and a Polygon is an array of three or more points that form a 2D shape (this is used to display buildings).

Below are the static classes that are used to store the JSON data, although not all of the variables are actually used, they are stored for future use:

```

public class Build {
    public Array<Features> features;
    public String type;
}

public class Features {
    public Geometry geometry;
    public String id;
    public Properties properties;
    public boolean clipped;
    public String type;
}

public class Geometry{
    public Array<Object> coordinates;
    public String type;
}

public class Properties {
    public String kind;
    public int sort_key;
    public String is_link;
    public String is_tunnel;
    public String is_bridge;
    public String railway;
    public String highway;
}

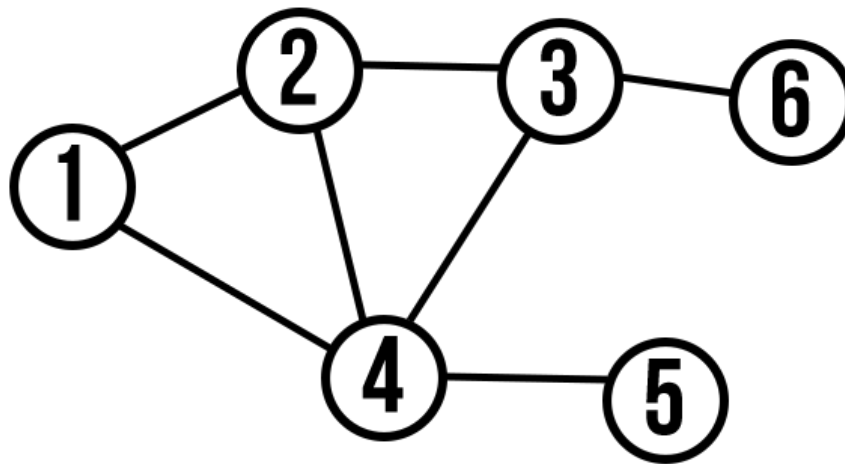
```

4.3.4. Graph

Once the points of the road have been obtained the next stage would be to identify how to display the points in order for them to be rendered and for pathfinding. As mentioned previously, the best way to represent the problem is as a graph; a graph is a set of vertices and a collection of edges that each connect a pair of vertices. With my problem, the vertices represent intersections and edges represent streets. I could give a cost (weighted graph) to each edge; high costs would mean the road is long and busy, low costs would mean the road is small and not busy. However I will not do this, although it may be something I will consider in the future. Another thing to note is that the graph is undirected.

I then need to decide how I am going to represent the edges in my program. There are two main ways of doing this (Sedgewick & Wayne, 2013):

- Adjacency list - vertices are stored as objects, and every vertex stores a list of adjacent vertices;
- Adjacency matrix - A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices.



Given the graph above the data structures will represent the nodes in the following way:

- Adjacency Matrix

Vertex	Vertices Adjacent To
1	2,4
2	1,4,3
3	2,4,6
4	1,2,3,5
5	4
6	3

- Adjacency Matrix

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	1	1	0	0
3	0	1	0	1	0	1
4	1	1	1	0	1	0
5	0	0	0	1	0	0
6	0	0	1	0	0	0

Both representations have their advantages and disadvantages, the adjacency list uses less storage than the matrix, adding a vertex also takes up less time. Adding an edge is the same for both, however removing an edge is quicker for the matrix rather than the list, because when removing, it has to find all vertices and edges (Algorithm and Data Structures, n.d.).

After careful consideration, I believe the best way to represent the graph is as an adjacency list as it is much more efficient than the matrix and uses less space.

4.3.5. Pathfinding

Once the representation has been achieved, pathfinding will need to be considered. There are numerous ways of doing it, but the ultimate goal is to find the shortest path between two points in a fast time. The best algorithm is A*, it uses heuristics to find the shortest path; another is breadth first search, which would find the path that crosses the least amount of vertexes, although it seems acceptable, it would not necessarily be the shortest path.

There are numerous others, however I will adopt A* as in theory, providing there is a good heuristic function, it should find the shortest path. The algorithm will start at a given point and then use a heuristic function to estimate the shortest path to goal, as it traverses the graph. It follows the path of the lowest expected total cost and also keeping a sorted priority queue of alternate path segments. The heuristic function I will use is the straight line distance. A* is complete and will always find a solution if one exists.

4.3.6. Physics

Perhaps the most crucial aspect of the game is the physics; getting this wrong can make a difference with a good game and a bad game. Thankfully LibGDX has a physics engine called Box2D. Box2D is a 2D physics library and is one of the most popular ones for 2D games and has been ported to many languages and many different engines. Box2D was originally written in C++, however there is a thin Java wrapper which allows it to be used in LibGDX.

When setting up Box2D the first that needs to be created is the world. The world object holds all the physics objects or bodies and simulates the feedback between them. However it does not render the objects; that is down to me and is explained in the next section. Having said that, LibGDX comes with a class called Box2DDebugRenderer which draws the outlines of the objects in the world, which is an extremely handy tool for debugging purposes.

To create the world the following code is used:

```
World world = new World(new Vector2(0, 0), true);
```

The first argument supplied is a 2D vector containing the gravity: 0 to indicate no gravity in the x direction and the other 0 is the force in the y direction. This means that there will essentially be no gravity, which is correct as the force of gravity will be facing downwards into the screen (the z-axis). If the game was a side view of the car, then gravity would be acting down on the screen, so the force in the y direction would be something like 9.8, which is the force of gravity on Earth in real life. There is a convention, which I mentioned previously, which has to be kept to and that is in Box2D 1 unit = 1 metre.

The second value in the world creation is a Boolean value which tells the world if we want objects to sleep or not.

I will now talk a bit about the creating objects in Box2D, however I will not go into too much detail as it is very long, the Box2D documentation can be viewed for more information. In Box2D objects are referred to as bodies and each body is made up of one of more fixtures, that have a fixed position and orientation within the body. The fixtures can be

any shape or a variety of different shape fixtures can be combined to make more complex shapes.

A fixture has a shape, density, friction and restitution attached to it. Each one is explained below:

- Shape - the actual shape of the object;
- Density - the mass per square metre;
- Friction - the amount of opposing force when the object rubs against another object;
- Restitution - how elastic something is, a body with a restitution of 0 will stop as soon as it hits the ground (if the gravity is negative in the y direction), whereas a body with a restitution of 1 would bounce to the same height forever.

There two main types of bodies are below:

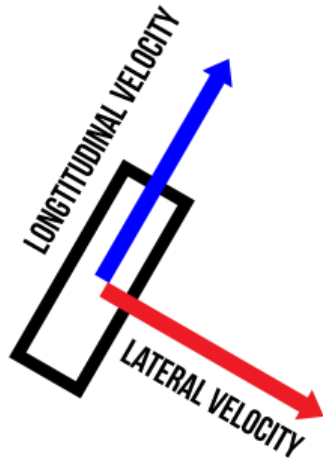
- Dynamic - objects which move around and are affected by forces and other dynamic, kinematic and static objects;
- Static bodies - objects which do not move and are not affected by forces. Dynamic bodies are affected by static bodies.

Impulses and Forces are used to move a body in addition to gravity and collision. Forces occur gradually over time to change the velocity of a body, whereas Impulses make immediate changes to the body's velocity (husainhz7, 2014).

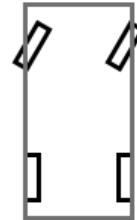
The first and most important object to display and interact with is the car. There are various ways of displaying a car in 2D. Whichever way I choose the parts of the car will always be dynamic.

My original plan was to have one body to represent the car and apply all the rotation and translation through the users input using the physics engine Box2D. However, Box2D is much more powerful than that; it allows you to create four separate wheels and attach them to the body. Forces can then be applied to these wheels to simulate the forces in real life. The front two wheels of the car are restricted to steering and depending on the drive of the car, power may be applied to the back two wheels (rear wheel drive), the front two wheels (front wheel drive) or all four wheels (four wheel drive).

WHEEL



CAR



The diagram above shows very simply how the physics works, the wheels are the most important part of the car. The Longitudinal velocity is the velocity that is experienced in the direction of the wheel, the lateral velocity is the velocity perpendicular to the wheel. Forces are applied in the direction the wheels are facing, impulses are also applied in the lateral direction to simulate turning. The code would be something similar to this:

1. Apply linear impulse (apply impulse at a point) and the centre of the wheel to cancel out lateral velocity;
2. Apply angular impulse to cancel angular velocity;
3. Apply force in direction wheel is heading in at the centre of the wheel;

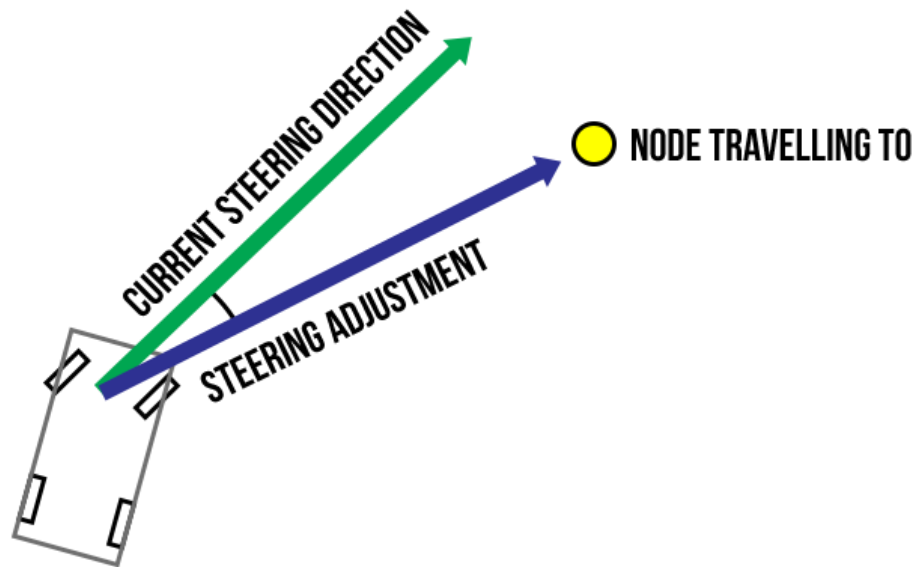
Four wheels are then connected to a car body and they work together to simulate movement of a real car (iforce2d, 2014).

The other objects that need physics are the buildings; however they are much simpler than the car. Once the data about their vertices has been retrieved from OpenStreetMap, they then need to be put in the game. They will be displayed in the game as static object with very high mass. This means that they will not move and be heavier than the cars.

Pavements is another object that could be considered. They are slightly more difficult to create than buildings, because with buildings when the player collides with them, they would bounce off which is simple to do in Box2D. However, with pavements when the player collides with them, they need to be able to drive on top of it; therefore there needs to be a collision detection mechanism, which checks whether the player has collided with the car and then the appropriate action needs to be taken, for example the traction of the wheels could be reduced.

4.3.7. Artificial Intelligence

The Artificial Intelligence is the next aspect of the game that needs to be considered, because of the time, it is very simple. It works in the following way; the AI is given the shortest path to the goal as queue of vertexes sorted in the order that they are encountered. The AI will then be accelerated to the goal, adjusting the steering if necessary.



The diagram above shows how the steering works; the idea is to keep the angle between the current steering direction and the steering adjustment as small as possible. Once the AI has reached within a certain distance to the node, for example 5m, it will accelerate towards the next node in the queue instead.

4.3.8. Rendering

As mentioned previously I have decided to use a MVC pattern for my game, which means that my rendering will be separate from the rest of the code. The actual process of rendering will be done using the methods available in LibGDX, however as the map could be large and the viewing frustum is small, it is important that I do not render the whole map, as it would waste a lot of CPU cycles, because it would be rendering objects that are not in view. There are various data structures that can be used to only render objects that are in view. To keep it simple, the data structure will only have to store static objects, such as roads and buildings. All the dynamic objects (player car and AI cars) will be rendered regardless if they are in view or not, as there are not many of them (Game Rendering, n.d.).

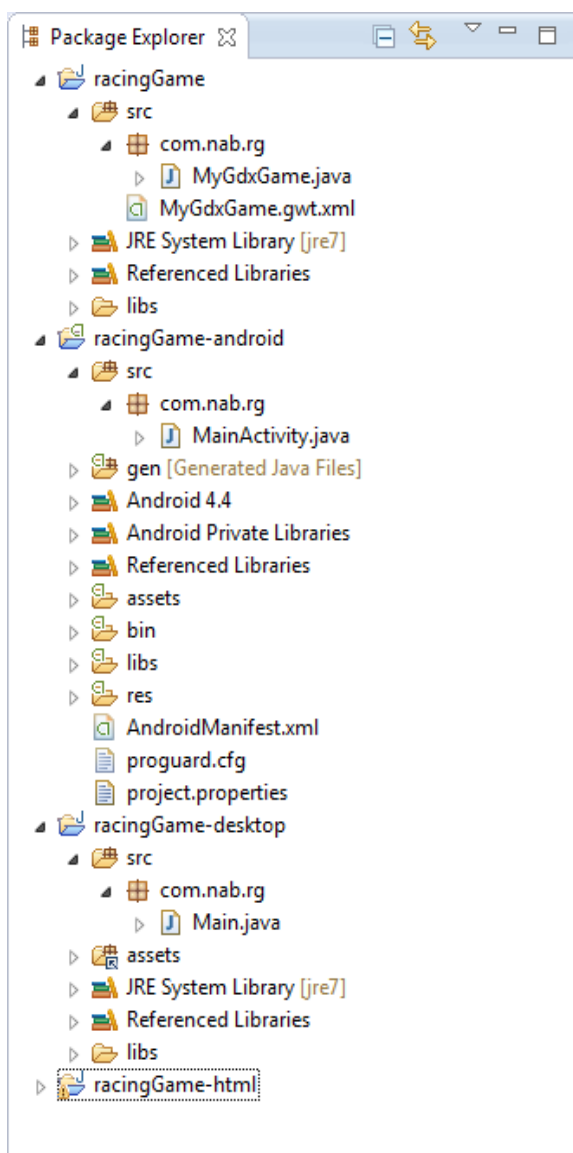
One such data structure is a uniform grid; it is perhaps the simplest way to manage the objects in a game. It's a simple 2D grid with equally sized cells, each object is placed into one of these cells, if there is an overlap then they can be put in more than one. When rendering all that is needed for culling is to check the view frustum against these cells to determine which ones are visible. This spatial data structure works well with both static and dynamic objects. However the main problem is that it does not work well when the objects

are of different size. In my game there is likely to be huge contrasts in the size of buildings and roads, so this option is not ideal.

A better solution, which caters for different sized object is a quadtree; this is a tree data structure in which each parent node has four children. It works by recursively subdividing it into four square regions, each cell having a maximum capacity set by the programmer. When maximum capacity is reached, the cell splits. This is perhaps the best data structure for my purposes, to keep it simple, it will start of as a single node. When more objects are added to the quadtree, it will eventually split into four sub-nodes. Each object will then be put into one of these sub-nodes according to where it lies in the 2D space. Any object that cannot fully fit inside a node's boundary will be placed in the parent node (Lambert, 2012).

5. Implementation

The implementation section looks at the various classes at more detail. The full code for the classes described are included separately to this report. Before I discuss this I would just like to talk about the process of setting up the project. Getting the project setup is trivial; the ADT Bundle for Windows is simple to install, although some additional add-ons are needed, it is still quick and easy. The LibGDX Project Setup application gets all the necessary project files, with a couple of clicks. This setup can be done manually however, configuring each project by hand (Desktop, Android, HTML, iOS etc.) can become a bit tedious and is often error-prone. So this tool places the files in the right folders and correctly links them to the Eclipse projects. Below is a snapshot of what the project looks like once imported into Eclipse:



The folder racingGame is where the main Java code goes; the Java file called MyGdxGame.java is invoked by the various platforms when it is run. I am only interacting with the Desktop and Android versions of the project, so the rest can be ignored.

The folder racingGame-android contains all the files necessary for the application to work on Android. AndroidManifest.xml is something all android applications must have. It presents essential information about the game to the Android system, such as components of the application - the activities, services, broadcast receivers and also the permissions the application must have in order to access protected parts of the API and interact with other applications (Android, n.d.). I have edited this file to give my application permission to use the internet and allow the device to stay on. The images and other files the game uses are stored in the assets folder located in racingGame-android. The other platforms also use this folder.

The racingGame-desktop contains the files to run on the desktop. I have used this a lot when testing my application as it compiles and runs the code much quicker than on Android.

5.1. Game Implementation

5.1.1. Structure

The classes have been split up into different packages to make the structure of the code more organised. The root package contains the Assets class that contain references to the textures to be drawn, a Settings class and a class called MyGdxGame, which is the main class called by each individual platform (Desktop, Android, iOS etc.). Below is a list of the additional packages used, with a brief description of the contents:

- Controller - contains various classes to control the input of the game, the class to control the AI players is also in here;
- GameUtils - has a variety of classes that help with the game;
- Internet - classes that use the internet to download data;
- Models - all the models that are used in the game, such as cars, buildings and wheels;
 - VectorData - has classes that store all the GeoJSON data once it has been downloaded
- Screens - classes to represent the screens used, such as the main menu and game screen. They all extend the AbstractScreen class, which has common variable and methods that each class uses.
- Search - contains the various pathfinding methods used such as A* and Breadth First Search. It has a class called AbstractSearch, which each search method extends. Similarly to the AbstractScreen class, it has common methods and variables that each class uses. There are also additional classes that extend the Node class found in the Model package to make the pathfinding work;
- View - has a class that renders the models on screen.

5.1.2. Game Description

To describe the implementation of the game, I will go through each class in the order that it is encountered from when the application starts, adding additional information if necessary.

The `MyGdxGame` class is invoked by the platforms; it contains code to load all the files in the `Assets` class. The `Assets` class itself contains images and fonts that are all stored and accessed as static variables, for example the image of a wheel can be accessed using `Assets.wheel1`. The LibGDX class called `AssetManager` stores and administers the assets in the game. Once they have all been loaded, it goes to the `MainMenuScreen` class.

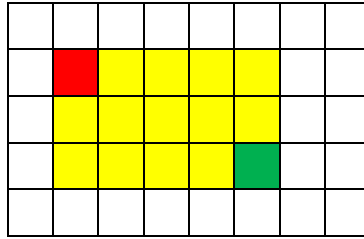
`MainMenuScreen` is the first visual output shown, it displays the title of the game, an image of a car and three buttons that allow access to the other screens of the game. The class makes use of LibGDX's `Scene2D` class to construct the user interface and provide clean and animated transitions when switching screens. I also used something called `Skins`, which is a collection of files that go together to make up the user interface. There is a JSON file which describes the various properties of the Skin such as how the widgets should look. There is a `TextureAtlas` that describes the layout of all the images that make up the user interface and there is a `fnt` file and the associated image, which has information about the fonts (Game From Scratch, 2013). To generate the font a tool called `Bitmap Font Generator` can be used (Jönsson, n.d.).

The `MainMenuScreen` extends the `AbstractScreen` class, which has various components used by all the screens such as the initialisation of the `SpriteBatch` class, creation of the `Stage` for `Scene2D` and creation and positioning of the camera. It also has code to render the background, which is the same across all screens (except the `GameScreen`) and also to update the camera. In addition it has code to deal with instances where the screen is resized.

From the `MainMenuScreen` the user can reach the `SettingsScreen` to change various parameters of the game and the `AboutScreen` class, which just has some general information about the game. Again both classes make use of `Scene2D` to make the user interface more appealing. Each screen also has a `BACK` button, which returns them to the `MainMenuScreen`.

From the `MainMenuScreen` class, the user can also go to the `RaceSelectScreen`, by pressing `PLAY`. This screen perhaps has the biggest change from the initial design. Originally, I wanted to create an interface that allowed the user to interact with a map and choose where they want to race by placing markers at the start and end locations. However, due to time, I was unable to complete this; instead, the user is required to input the start and end latitude and longitude coordinates manually. There are four input boxes to do this, one box for the starting latitude, one for the starting longitude, one for the end latitude and one for the end longitude. There is also a `BACK` button to return to the `MainMenu` and a `GO` button which takes them to the `LoadingGameScreen`. The values from the input boxes are sent to the `LoadingGameScreen` for processing.

`LoadingGameScreen` is crucial for the creation of the maps. It takes the start and end latitude and longitude coordinates of the race, then converts them into tile numbers at zoom level 16 (x tile number, y tile number and zoom, the format specified by Mapnik Vector Tiles) using the `MercatorProjection` class, located in the `GameUtils` package. It then fetches those tiles and the tiles that are within that region. For example:



If the red square is the starting tile and the green square is the finish tile, the program will also fetch all the tiles in yellow. These tiles are then downloaded, using `JsonTileDownloader` class, located in the `Internet` package. This class takes several parameters; the first is the array that stores the data, the x and y tile numbers, the zoom and the type of object to fetch (building or road). The class makes a HTTP request to fetch this data in JSON format, if there is any sort of error, the class responds with an error message and fetches map data stored locally. If the data for each type is downloaded successfully, the JSON data (in string format) is converted to the corresponding fields in the `Models.VectorData` package. The root of this data is called `Build` and contains data of one tile, so an array of `Build` is used to store multiple tiles (there are two separate `Build` arrays, one to store the buildings and one to store the roads). The bottom left corner of the map and top right corner are converted from x and y tile coordinates to longitude and latitude coordinates. All this data is passed to the `GameScreen` for further processing. The `LoadingGameScreen` also extends the `AbstractScreen`, so has the same background as it; it also features an image that rotates to indicate that the game is loading.

The `GameScreen` class contains the basic components needed for the actual game to work. It initialises a lot of components such as the `Box2D` world, the `WorldController` class, the camera and the various user interface components, such as the clock, that shows how long the user has been in a race for and a number indicating the position of the user in the race. This is calculated simply by working out who (player or AI) is currently closest to the finish line. These features were not in my original design, however I decided to implement them to improve the gameplay. If the platform is Android or iOS it will display on screen touch controls to steer the car, if it is not then these controls are not displayed. A pause button is displayed in the left hand corner of the screen regardless of the operating system; this pauses the current game state and provides options to allow the user to resume the game or return to the main menu. Again these user interface features make use of `Scene2D`. A quick note about this, in my design I said I wanted to adopt a MVC pattern, I have with the every object that I have used, except the user interface, the reason being is the `Scene2D` couple model and view. They store data that is often considered model data in games, such as their size and position, but they also have the view, as they know how to draw themselves. This coupling makes MVC separation difficult. Since I have `Scene2D` in the majority of screens I have not been consistent with the MVC pattern, however I have been consistent with the MVC pattern in the place where it matters the most and that is in the actual game.

The `WorldController` class is at the core of the game, it instantiates all the models that are needed for it to function. The other models are not accessed directly, but in fact through the `WorldController`. I had originally planned on calling it `World.java`, however the name conflicts with the `Box2D` world and can cause confusion. There is a lot going on in the class, when it is instantiated in the `GameScreen`, parameters are passed to it, such as the `Box2D` world, the array containing the data about the building and roads, the bottom left

and top right corner of the map and the starting and end locations of the actual race (all given as longitude and latitude coordinates). The bottom left hand corner is converted to metres (the longitude and latitude are converted to x and y world coordinates) using the static methods available in the MercatorProjection class. This position is considered the origin (point 0,0 in the world), then all other objects in the world are set relative to this. The reason this is done is because Box2D cannot deal with extremely large figures, so essentially a seed value is chosen and the other objects are set relative to that. For example the width of map is calculated by getting the top right hand corner, converting it to metres and then it is subtracted from the bottom left corner to get its value relative to the bottom left position. The start position and end positions of the race are also found in metres relative to the bottom left corner.

The crucial part of the map is then created, first the buildings are generated, there is a class called GetVectorData, which has static methods to convert buildings and road data into world coordinates so that they can be displayed in the game. The method buildBuildings takes an instance of the world, the array of Build and the bottom left corner of the map. It loops through all the buildings in the array (ignoring duplicates), extracting the coordinates of the polygon, relative to the origin. It then creates these buildings by creating an instance of the Building class, which takes the world and the shape of the building. The Building class generates each building using Box2D so that collision detection can be applied to it.

There are several checks in place, the reason for them is explained later; it checks if the polygon is convex (Bourke, 1998) (Fisica, 2011). If it is then the building is created, if it is not then the convex hull of the points is taken using the class PolygonConvexHull in the GameUtils package. These buildings are all stored in an Array in the WorldController class. Another thing to note is that the Array used is a special array provided by LibGDX, it is similar to the ArrayList class in the java.util package, but it is more efficient.

Creating the roads uses a similar process to creating the buildings, however it is much more complex. The road data is stored in a class called Graph; the graph has two private variables, one called Nodes, which is of type Node and one called edges of type Edge. The class contains getters and setters and other methods to adjust the values of the variables. It also has an important method called getAdjacencyList, which will be explained later. The Node class represents points as x and y world coordinates, the Edge class represents two Nodes that are joined together. The Edge class extends the StaticObject class which essentially creates a bounding box around it (this is used by the quadtree). The Edge also has various other variables such as whether the road is a bridge, the angle between the first node and the second node (used in rendering), the width of the road and the type of the road (path, highway, etc.). It also has various getters and setters.

In the WorldController class, the roads are actually generated by using the static method getGraph in the GetVectorData class. The code loops through the Build array, extracting the coordinates and other parameters and initialising the nodes and corresponding edges (very simplified explanation). Once they have all been created, any points that are not listed in the edges and nodes classes are found using a line intersection algorithm, using the method bruteForce, which resides in the class called BentleyOttaman in the GameUtils package (the reason for this is explained later). Once all intersections are found the points and edges are converted into an adjacency list using the method getAdjacencyList in the Graph class. The adjacency list is stored as a TreeMap, the first parameter is the Node and the next parameter is an array of all the Nodes adjacent to that

first Node. In order to sort the points in the TreeMap a class called `CoordinateComparator` is used which exists in the `GameUtils` class. The class sorts the points according to the distance from the origin. This class is the basis of pathfinding.

To make the pathfinding easier, the game finds the nodes that are nearest the start and end positions of the race. Since the nodes are given as a `TreeMap`, the nearest node can easily be found by finding the upper and lower bound using `treemap.floorKey(node)` and `treemap.ceilingKey(pointNode)`, then whichever node is closest to the original is chosen.

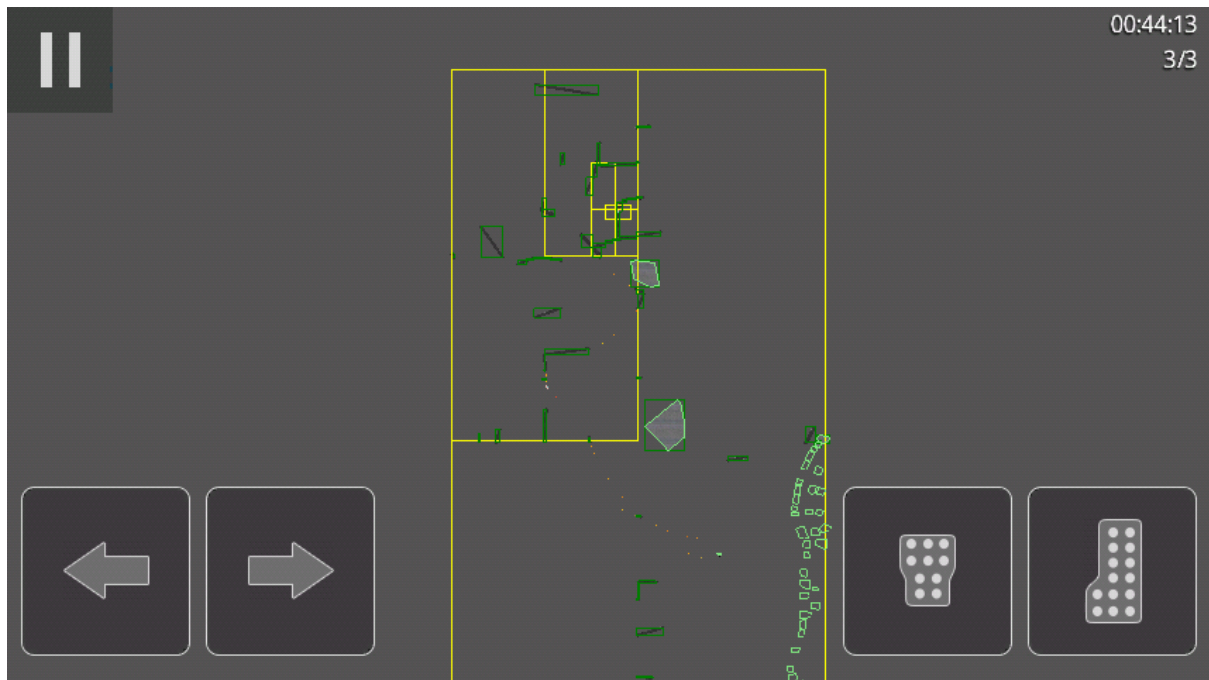
After this the pathfinding begins, for testing purposes there is an implementation of breadth first search, however the pathfinding algorithm that the game actually uses is A*. Each pathfinding algorithm extends a class called `AbstractSearch`, which contains the necessary components for any pathfinding algorithm to have, such as a frontier, which contains the nodes that need to be expanded and a closed data structure which has the nodes that have already been expanded. There are also methods to add data to those data structures. The `Node` class has also been extended for the various search methods, to allow more functions to be implemented, for example, the A* Node has additional functions to calculate the cost and heuristic estimates.

The car model is fairly simple; there is a constructor that takes parameters such as the width, the name of the car (player or AI) and the texture used for rendering. The `Car` class instantiates four of the `Wheels` class; it has several parameters, such as whether power is given to the wheels and if they can be steered. The main physics of the car (mentioned in the design) is also in here, it contains methods to apply forces and impulses. I made use of the physics engine, `Box2D` to simulate the actual physics of the objects.

In order to make the car move, there are various controller classes available. On desktop there is a class that takes the keyboard inputs to move the car, whereas on Android devices there are controls on screen to do this (these controls are hidden on desktop). In addition to these controls, there are also debugging controls for the camera, which are only used when the debug mode is on in the `Settings` class, the variables in the class can be changed by interacting with the settings screen in the game. On desktop the camera is controlled using the mouse, whereas on Android, the camera is controlled using gestures.

The AI is simple as well, it has not changed from the design stage. The AI cars are created, then each one is given a controller called `SimpleAIController`, this component of the game did not change as well from the initial design. The class takes the AI car and the path to the goal in the constructor, power is applied constantly, only the steering is altered, until it is aligned with the node it is travelling to. It does this by finding the angle between the current wheel position and the node using the method `getAngle` in the `AngleUtils` class, which is located on the `GameUtils` package. The problem with this implementation is that it is not perfect, as the cars do not brake when they approach bends, which causes it to “wobble” a lot.

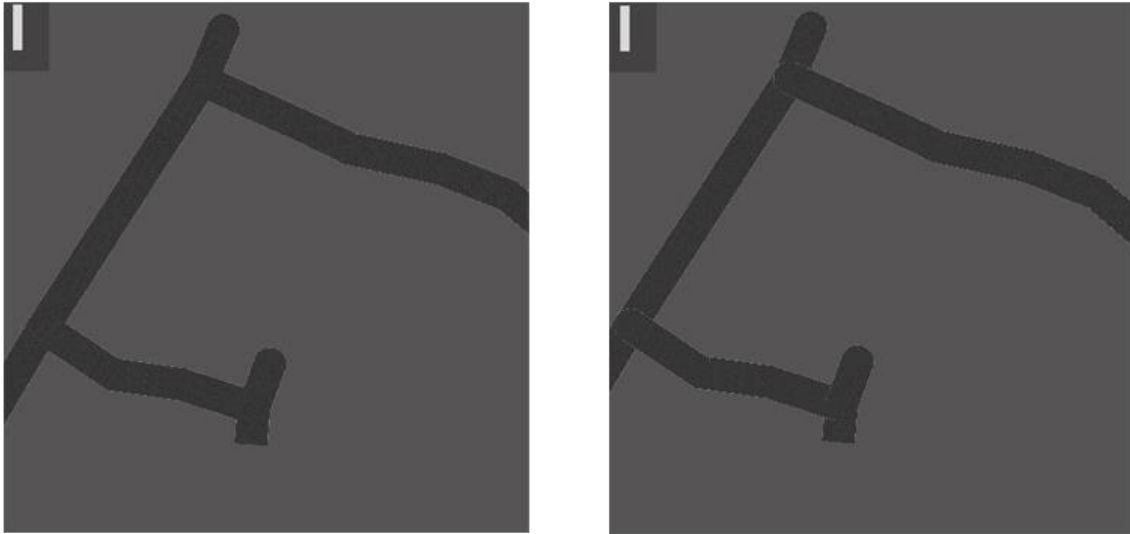
The quadtree is an attempt of optimisation; it is instantiated in the `WorldController` class, but the `QuadTree` class resides in the `GameUtils` package. Admittedly I had more trouble with it than I should have (explained later), however it works fine, again it isn't perfect. Before objects are added to the quadtree they must have a bounding box in a shape of a rectangle. The only objects that are added to the quadtree are buildings and roads, they extend a class called `StaticObject`, which deals with the bounding boxes, but the objects are required to specify an x and y coordinate as well as a width and a height. Below is a screenshot showing it in action.



It is quite difficult to see, but the dark green rectangles represent the bounding boxes of the objects that are currently being rendered, the light green are the objects that can be collided with. The yellow rectangles represent the boundaries of the quads currently in view.

In the `WorldController` class there is a method called `update` which is called in `GameScreen`'s render method. This means it is called constantly. In this method various objects are updated such as the timer, the `Box2D` world, the player's car, the AI and the position the user is in the race. There is also a method that checks whether either of the cars have reached the finish line. Once the player has completed the race, the `GameScreen` displays the final position of the race.

In the `View` package dwells the class `WorldRenderer`; it is responsible for drawing the majority of the objects drawn on screen, by interacting with the `WorldController` class to fetch the objects. The main thing that is of interest here is the way roads are rendered, at first a layer of road textures with a border and a large width is put down, then a road texture with no border and a smaller width is put down to give the illusion that all roads are linked correctly. Below is a screenshot with and without this method:



On the left is with the algorithm and on the right is without. The left is much smoother and connected, whereas the right has unwanted artefacts.

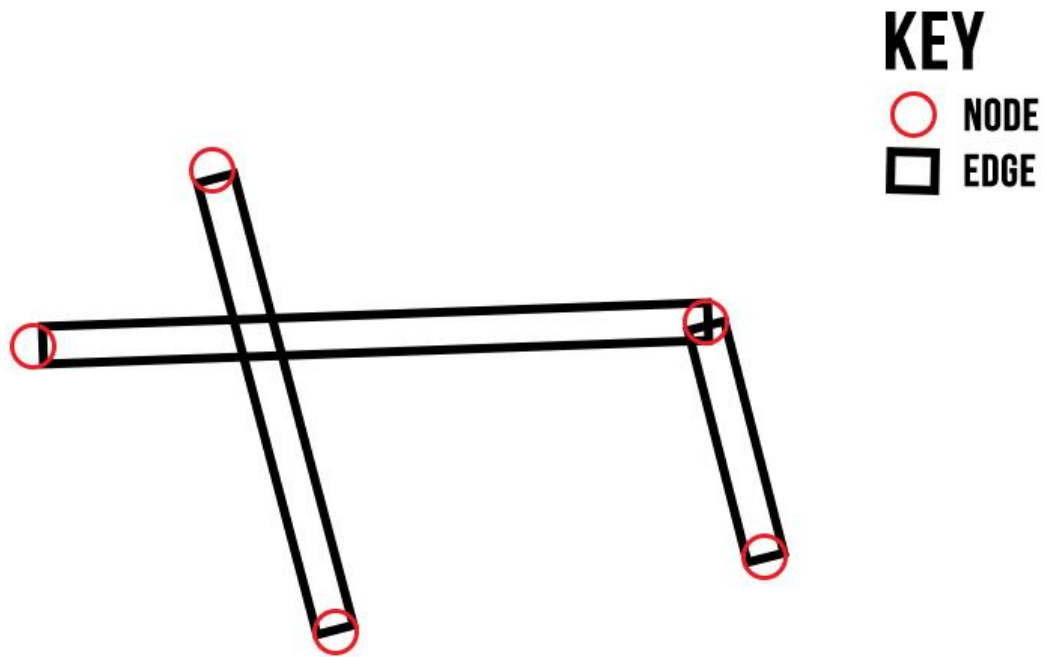
There is a class called Constants, which has various variables that remain constant throughout the game such as `PIXELS_TO_METRES`, which is the number of pixels on screen that equals 1 metre in the game.

5.2. Problems

Creating the game was difficult and I was faced with numerous problems, some of which are described in this section.

Getting the physics of the car and wheels right was quite difficult. I followed some theory I found online, but was not completely happy with the outcome; I tweaked it slightly to make the physics more fun.

Interacting with Mapnik Vector Tiles proved to be a challenge. Downloading the road and building data was not too much of an issue, as LibGDX has classes to deal with this, however getting the actual JSON data into a format that works well with my game was a huge problem. One such problem is that there are many intersections that are not listed in the file. This is because the Mapnik Vector Tiles were mainly used for rendering maps (it made sense not to include all these extra intersections to minimise the file size and reduce the number of roads to render). So I had to find all the intersections myself, so that the pathfinding would find the most efficient path to the goal. Below is a diagram showing a typical scenario:



The red circles are nodes and the black rectangles are the edges. As you can see there is a node that is supposed to be in the centre however, it is not listed. There is therefore some code that finds all the intersections, once found it performs “surgery”. It finds the two lines where there should be an intersection, it then creates a new node where the lines intersect and adds it to the array of nodes. It then adds four new edges to the edges array representing the new points and removes the two original lines.

The intersection algorithm used to find all the crossings in the road is not optimal. In fact to find the intersections, it compares all the lines with each other, which is $O(n^2)$. There are other, more efficient algorithms that can be used instead, such as Bentley Ottaman. I began implementing it, but due to the time constraint, I was unable to finish.

Another issue with roads, was in order to minimise the number of HTTP requests and avoid writing code that would somehow stitch the map tiles together, I decided to fetch one tile with the roads and one for the buildings at low zooms, such as 12 or 14. This worked reasonably well for a while, however I found that at these low zooms, it would produce lines that were not correct. This was discovered when testing the game; my pathfinding algorithm was not finding the shortest path. After some investigation, I found that the data was not right. When I zoomed out of the game the roads appeared to intersect, however, when I zoomed in I found that some of these road did not actually intersect. My code was not picking them up and therefore was not finding the shortest path. My solution was to stitch tiles at higher zooms together (zoom level 16). Originally I created a PHP script to do this and uploaded it to the Universities web server, however, I realised that this was not such a good idea as it is not easily cacheable, so I decided to do the stitching within my application.

The adjacency list that represents the roads were originally stored as a HashMap. This was actually fine for the majority of the coding, however, I eventually needed a function that finds the node nearest to a certain node; the HashMap was not suited for this as I would have to convert all the values in there to an array and check every node until I find the one which is closest to a certain point. So instead I used a TreeMap, which is much more

efficient, however, there have had to be performance compromises elsewhere, for example, HashMap is $O(1)$ for access and a TreeMap is $O(\log n)$.

The buildings also gave me some unforeseen problems. They are “unclipped” in each tile when fetched from Mapnik Vector Tiles, and this means that the same building can appear in more than one tile, which does not play nice with Box2D. This error was difficult to pinpoint as I was receiving a runtime error, however all it stated was that there was an issue with Box2D. After a long time I figured out the error and came up with a solution; since each building is given a unique id, there is a HashMap that stores them. If my game encounters a building that has already been created it ignores it. Another problem I had with the building was the fact that Box2D does not support concave shapes natively. There are various ways of solving this problem such as triangulation, however I decided to adopt a simpler method which takes the convex hull, which is the shape that completely encloses a set of points with the fewest number of nodes. There are various convex hull algorithms available, however thankfully LibGDX has this built-in, which uses Andrew’s algorithm to compute it.

Something which I completely missed in the design is a situation where the user has no internet connection. I found this relatively early on in the coding process. If the user has no internet connection or there is an error when downloading the JSON files, then it automatically loads a map stored in the Assets folder.

Rendering the roads correctly was also difficult, previously I specified a technique which does it in layers. Although it is simple, it took several attempts to get it working correctly.

I had minor issues with A* for path finding, it was not finding the shortest path. Originally to improve performance, the cost function I used was overestimating; I was using something similar to this: $a*a+b*b$ (given a is the node you are currently on and b the node you are trying to get to). I completely forgot one of the main rules with an A* algorithm, that it should never overestimate the true cost. With my mind set on improving performance, I completely overlooked it and it took me a while to discover the problem. It also reminded me of another rule, always optimise at the end. The final cost function used in the game is $\sqrt{a*a+b*b}$.

As I said before I had some issues getting the quadtree to work properly, this was down to lack of preparation. I had an idea in my head of how it was going to work and I followed some tutorials, but I was getting errors and it took me a while to pinpoint them. I was getting a situation where I was passing over a road, however it was not being rendered. After a lot of debugging, I found that the roads were not being inserted into the right quad in the quadtree. I resolved it and it now works fine.

6. Testing

6.1. Software Verification

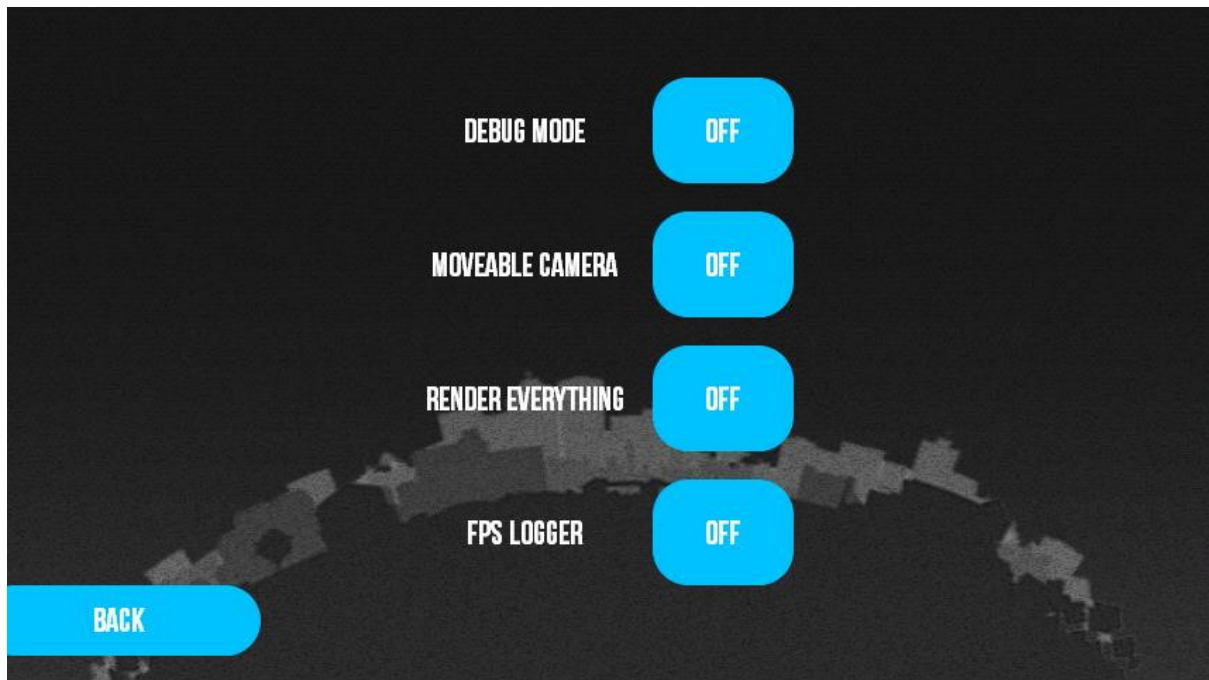
This section will look at testing of the game to firstly see if it met the original requirements. As the application is cross-platform, I can test it on multiple different platforms, however since my original idea was to make the game on Android platform, I will do all the testing on that platform as well. The phone I will be using is the Xperia Play, which is a handheld game console smartphone, currently running Android 2.3.7 (Gingerbread - API level 10). It was released in 2011 and has the following major specifications (Sony, n.d.):

- CPU - 1 GHz Qualcomm Snapdragon S2 MSM8255 (Single Core);
- GPU - Adreno 205;
- Memory - 512 MB.

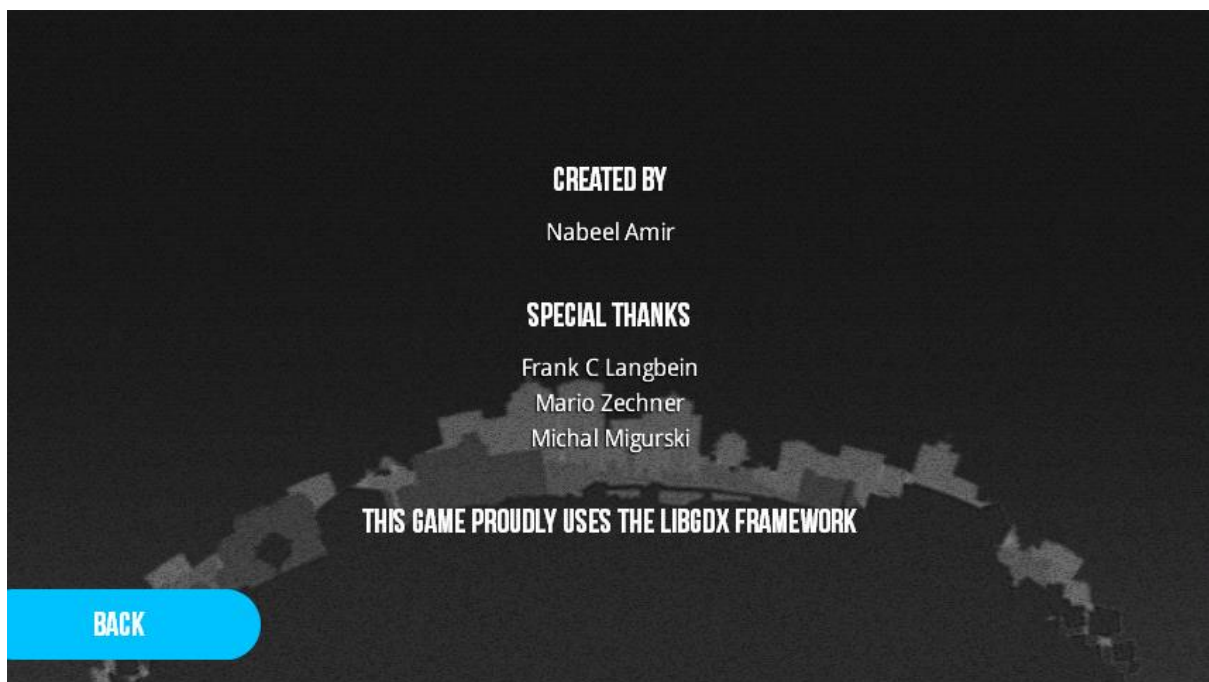
Below are various screen shots from the game. I used the screen capture function in Eclipse, provided by the Android SDK, to get these images:



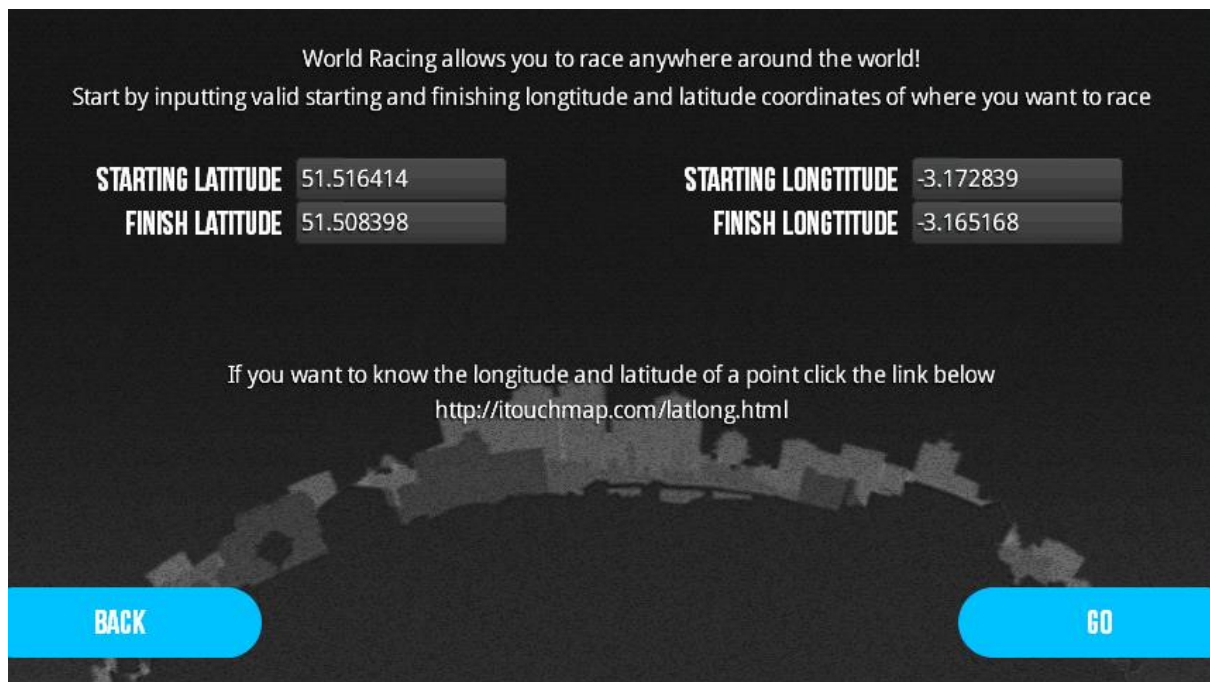
Above is the main menu which has not changed at all from the original design. One of my requirements was to have a nice user interface, which I believe I have met. It looks neat and it is also animated. What each button does is clear and this is consistent across all the screens.



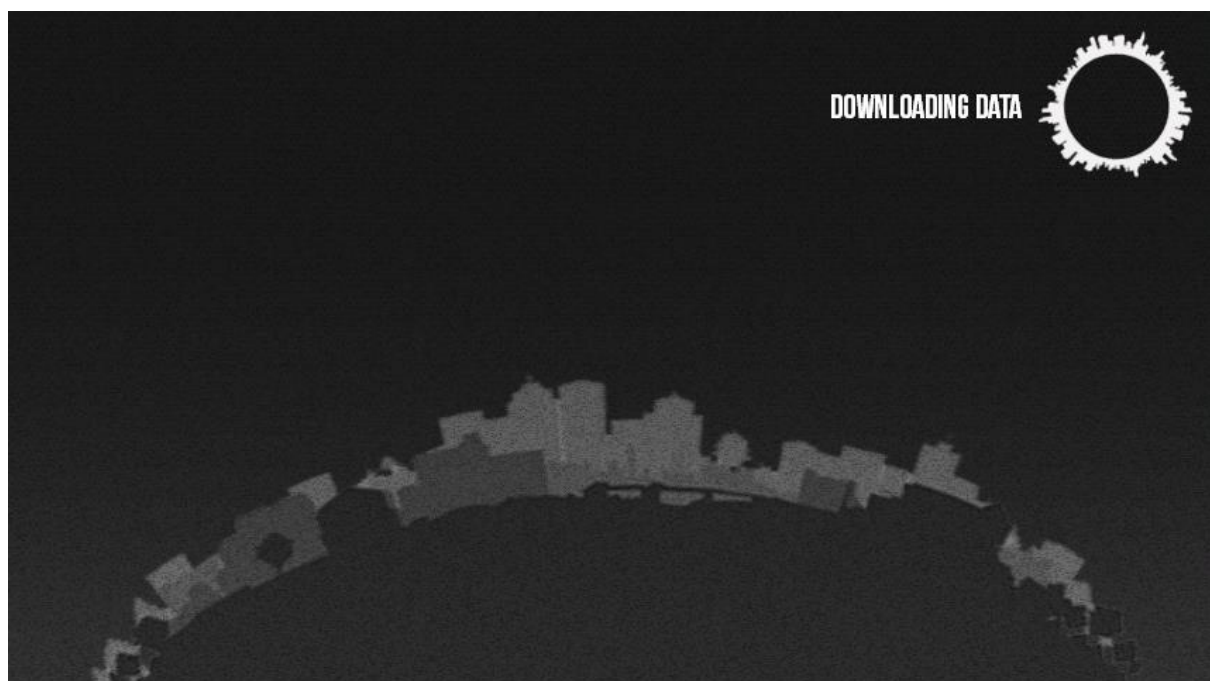
Above is the setting screen; this is here purely for testing purposes and if the game was being published, the above buttons would definitely not appear in the final release.



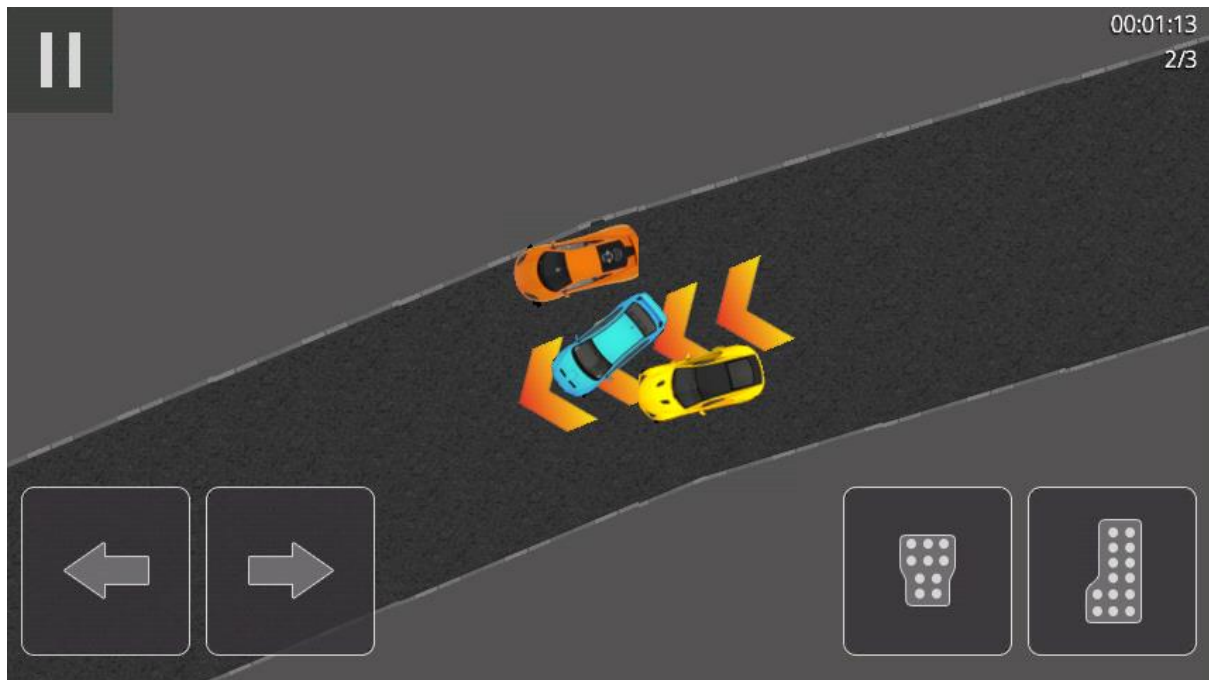
This is the about screen; it just contains some information about the creation of the game.



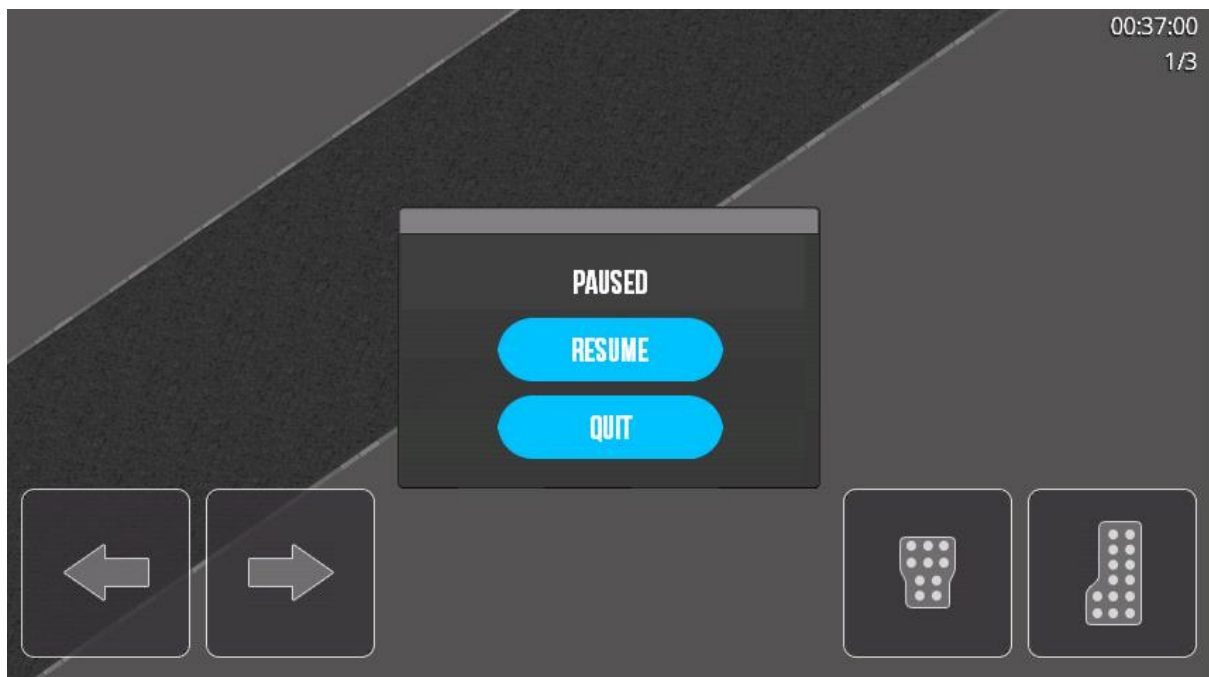
When the user wishes to play the game, they are greeted with this screen. Here the user can input the starting and finishing latitude and longitude. This is not ideal, as previously stated it would be much better if there was a map that the user can interact with to choose these locations. Once they have selected the race, they can press go and then the loading screen is displayed.



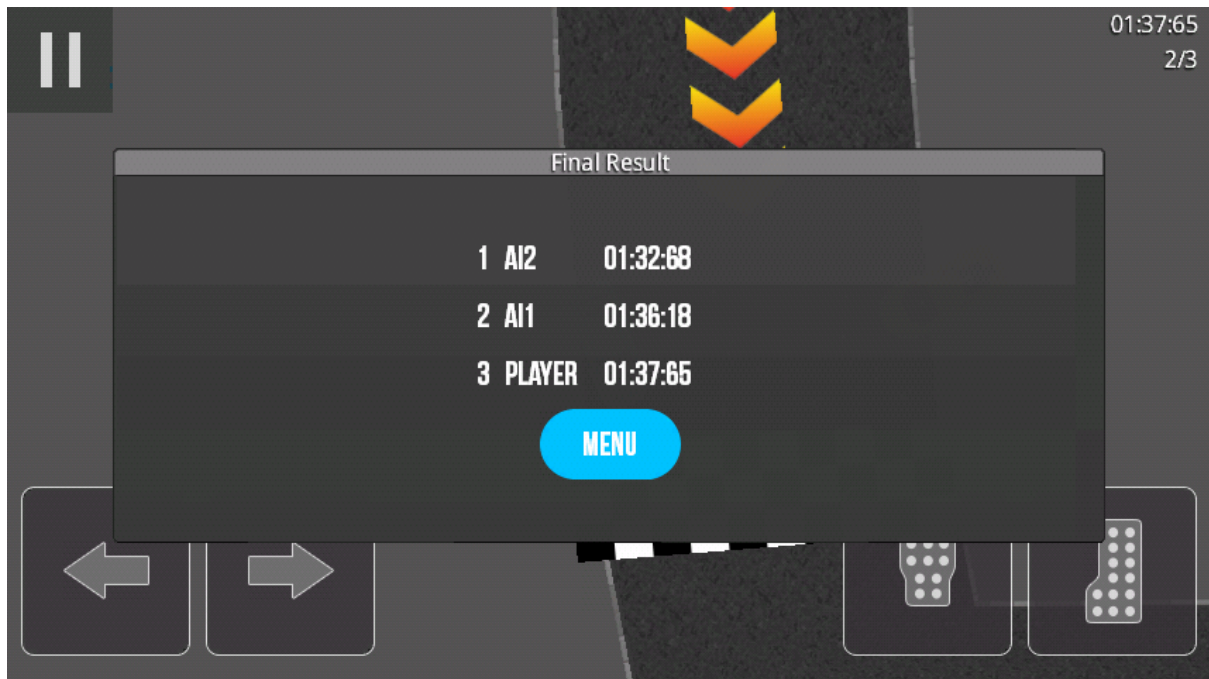
The loading screen provides visual feedback about what the game is currently doing; the circle structure in the top right hand corner rotates.



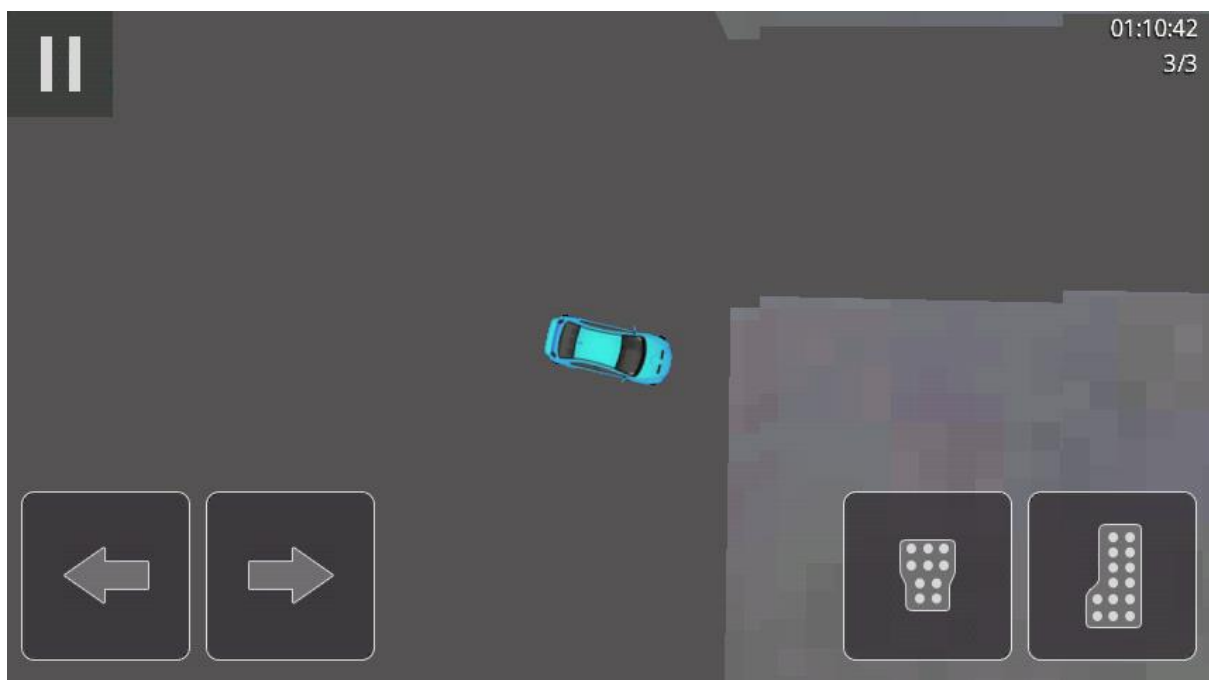
Once the game is loaded, the user is instantly put into the game. Again the user interface is neat here. The touch screen buttons have icons on them to indicate their action, they are also slightly transparent so that they do not completely obstruct the view of the game. The orange and yellow chevrons represent the path that should be followed, it is calculated using A*. The blue car is the player car and the orange and yellow cars are the AI, which meets some more of my requirements, which was to have AI and pathfinding. The camera angle and size has stayed true to the top down genre and it is not too big that the car cannot be seen and not too small so that the rest of the world cannot be seen.



When the pause button is pressed, the user can either resume the game or quit, which takes them back to the main menu.



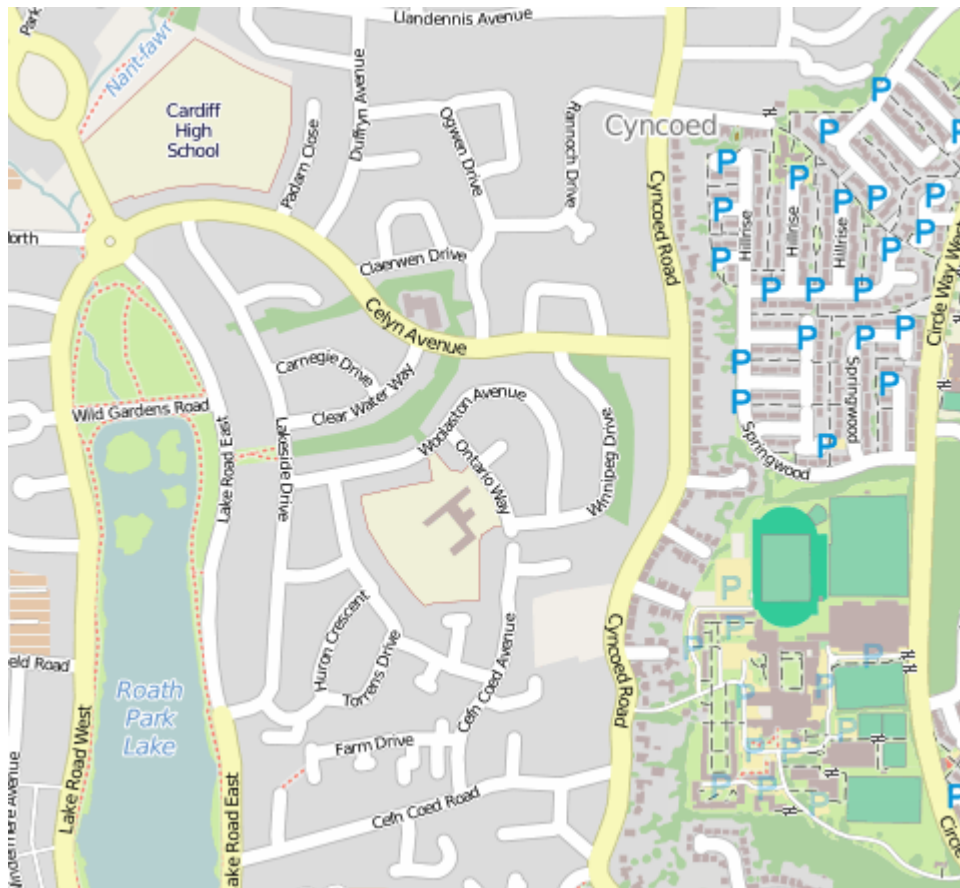
When the user has finished the race, they are presented with the above screen indicating the position they have come and the time they have completed it.



One of the requirements was to get the physics working and it does so reasonably well. The driving is fun and quite realistic; collisions with other players and buildings work fine, no damage is applied to objects when they collide, they just bounce off each other.



One of my main requirements was to have a map that uses OpenStreetMap to fetch the data. Above is a screen showing the map; the light grey blocks are buildings and the thin dark grey lines are the roads. To get this screenshot I made sure that the quadtree was no longer used and the camera no longer followed the player (render everything - on and moveable camera - on in the settings screens), then started the game and pinched the screen to zoom out. Below is a screenshot of what the intended map looks like from OpenStreetMap site (OpenStreetMap, n.d.).



As you can see the geometry of the objects are similar (with the exception of the buildings) as well as the relative location. Some features that appear on the map do not appear in the game such as grass, water areas and street names.

The size of the file is around 5MB, which is under the file size requirement I originally set.



Above is the main menu on a tablet and a phone. The user interface looks fine on the phone, however on the tablet it seems very small, however it is fine for its purposes. If I wanted to support the tablet properly, I would have to scale the images up and make sure the resolution of them is not affected.



Above is the actual game running on both devices; on the tablet, more of the screen is shown.

6.2. User Testing

Now that I have tested it on my device, it is important to test it on other people's devices. This will help to identify the following (Sloper, 2004):

- User's view of the system - see what the user thinks of my game, by playing it on their device;
- Compatibility testing - I have one Android device. Testing it on a range of devices with different processor speeds and screen resolutions will help identify if it works on various Android devices;
- Identify bugs - There may be some bugs that I did not identify by testing on my device, there may also be bugs associated with specific devices. This is quite common when developing on Android. For example, my game may register two finger input on the touch screen on one device, but may not have been picked up by another for whatever reason.

I will also have to make sure that I get the right people to play the game, meaning that they fit the criteria of the target audience I set earlier.

I gathered five users who had Android devices and uploaded the game to their device. I let them play the game for a while, and then gave them a questionnaire to fill in. On the whole the response was fairly positive; the majority of the users enjoyed playing the game, however, some of them indicated issues with the user interface. The actual responses from the users are in the appendix and below are a select few of the things that the users did not like about the game:

- Controls were too small - the solution to this would be to make the buttons bigger and then test it further on various devices;
- Game was too easy - this can be adjusted by changing the power given to the AI
- Track was always the same - I can see why the user said this, as the roads are all the same texture; so a good thing to do would be to vary the textures and lighting effect to distinguish between different places;
- Did not like inputting the lon and lat manually - this is understandable, as selecting where you want to race is a pain;
- I didn't recognise the buildings - this is due to the fact that the convex hull of the buildings is taken, so the true shape of the building is not preserved.

The users suggested the following improvements for the game:

- Make controls less sensitive and enlarge;
- Ability to choose what car you could be;
- One user suggested I include more tracks; this is quite alarming as the point of the games is that you can race anywhere around the world, so one thing that needs to be addressed is that this feature is made clear in the game;
- Use map or something to choose where you want to race.

6.3. Performance Testing

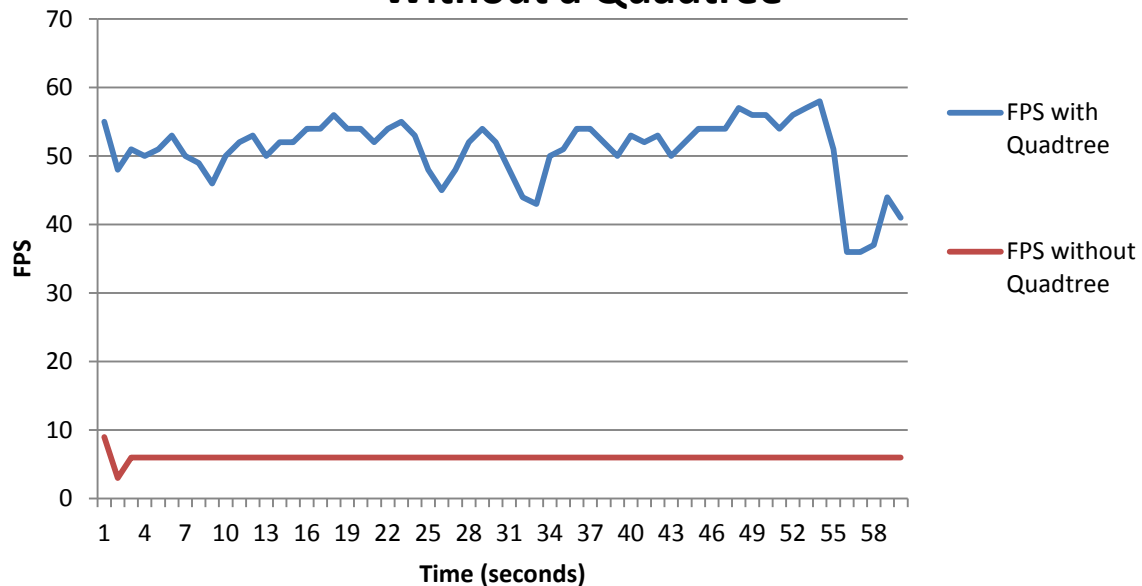
In order to improve performance, I have used various methods and data structures so that the game experiences no frame rate or other issues.

6.3.1. Quadtree

One such data structure is the quadtree; I implemented this so that the game only renders objects that appear on screen, beforehand I was rendering everything. I will now test to see if this optimisation is worth it by measuring the frames per second achieved by the two methods over 60 seconds.

To test it without the quadtree, I went to the settings screen and set render everything to on and show fps logger to on. The average fps is shown in the console every second, when the Android device is plugged into the computer. To keep it fair, both tests were given the same place to render. I also played the game as if I was an actual user.

Graph to Show the Game Running With and Without a Quadtree

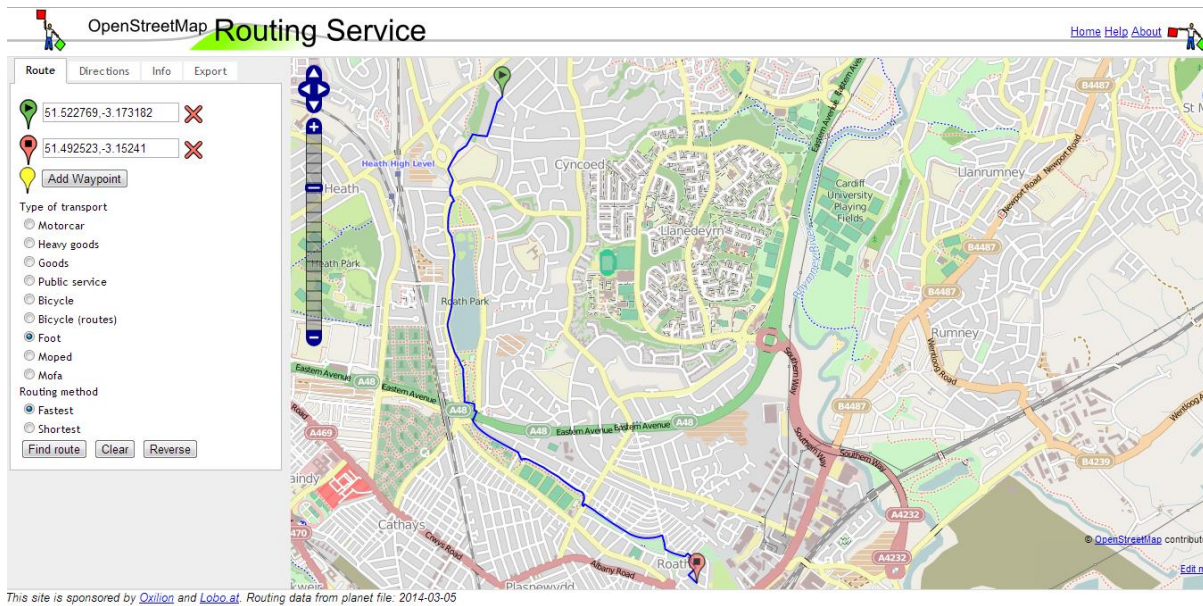


From the graph you can see that there is a huge improvement with the FPS when using the quadtree, however there is a problem, when everything is rendered; the FPS over 60 seconds remains fairly constant (average is 6 FPS), however the FPS with the quadtree, although higher, fluctuates drastically (average is 50.9 FPS). A possible reason for this is that when it renders too many objects, its FPS drops, but when there are few objects the fps increases. So although the quadtree has done its job and improved its performance, it will have to be tuned or changed to stop this fluctuation. However it does meet my initial requirement, which was to get the game running at a minimum of 30 FPS.

6.3.2. Pathfinding

Another thing to test is the pathfinding, I implemented two of these algorithms, although the A* is the one that is actually used, it would be interesting to see how both perform under certain circumstances. It would be quite difficult to test this, however, I will give each algorithm three different start and end positions. I will check if the algorithm has found the optimal path and measure the time that the algorithm found it. I will have to make various changes to the game to cater for this; I will first have to create a timer to measure how long it takes to find the path, once found all the roads have to be rendered and the path to the goal highlighted. To find the routes I used <http://www.yournavigation.org> (YourNavigation, 2014) which can be given as longitude and latitude coordinates. Because my game allows cars to use paths for pedestrians, I will find the shortest path achievable by foot.

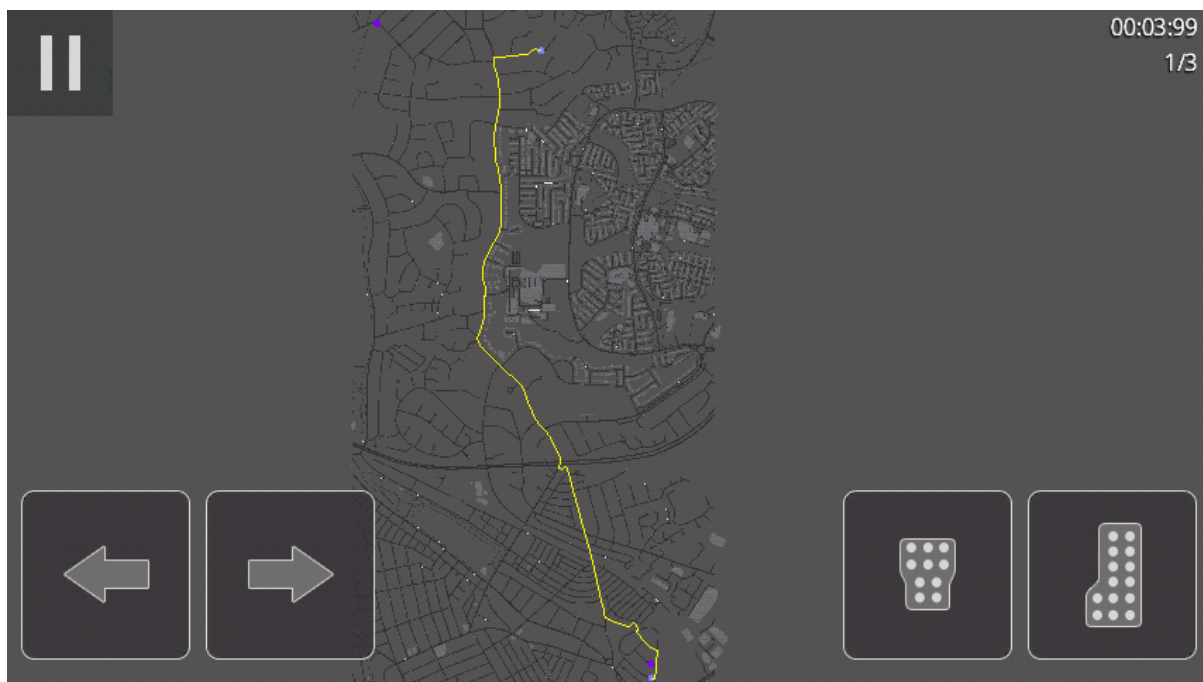
The first route was the following:



Starting point - 51.522769,-3.173182

End Point - 51.492523,-3.152412

After testing this I uncovered a problem with my code. Below is a screenshot of what was being displayed:

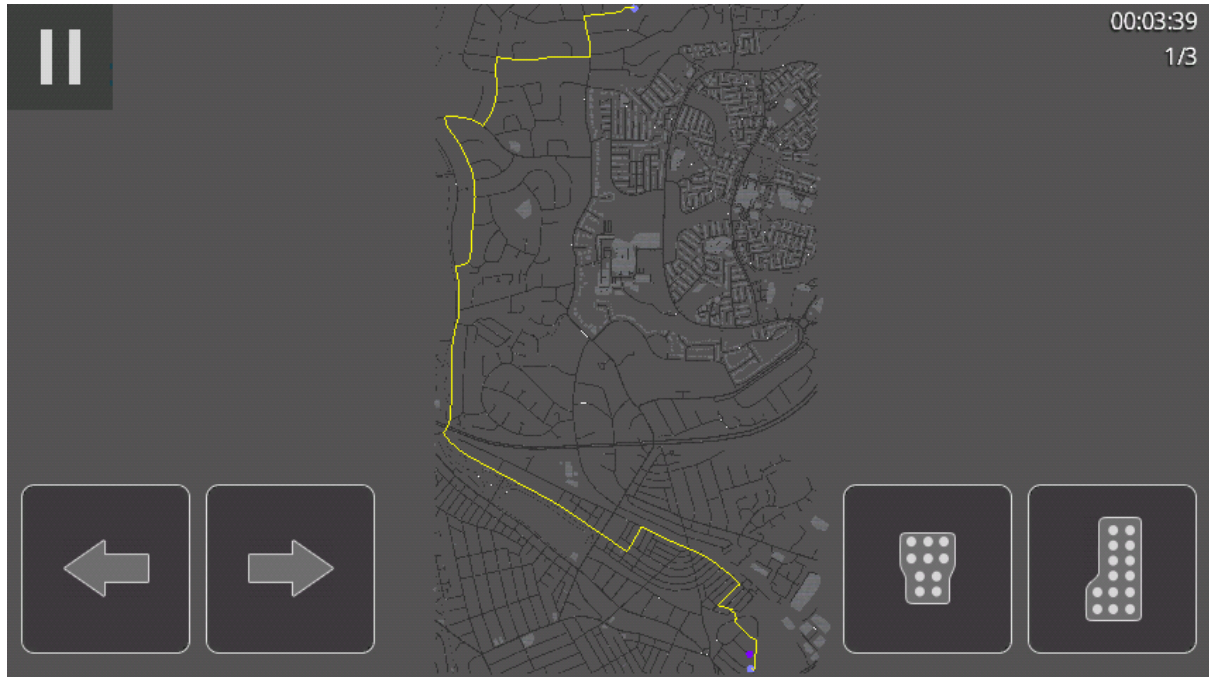


Although it may be difficult to see, the dark purple spot are the start and finish locations given by the user, the light purple is the closest node. That is a slight problem as it does not find the closest node, but a node that is near enough. The alarming problem is the distance at which the user inputted node appeared from the original. If you compare the games map to the one above, the start and end locations are very different.

I set about finding out what caused this and I narrowed it down to the precision that the latitude and longitude coordinates are stored as. They are stored as floats and they

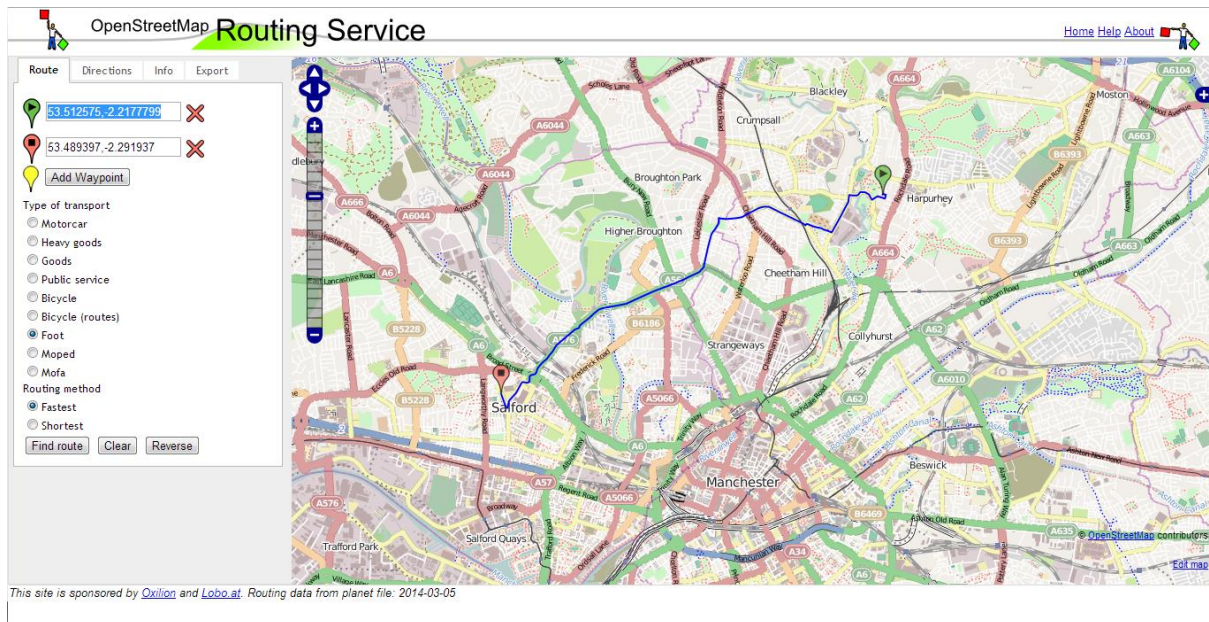
should really be doubles, which does cause precision errors, for example the starting latitude is 51.522769, but it is stored as - 51.52277, this may seem ok, but causes drastic changes in the world. This is something that should be fixed in the next update of the game. However, I will continue to assess the pathfinding algorithms. Moving on, the A* algorithm took 0.118286131 seconds to run on my phone.

The breadth first search result is below:



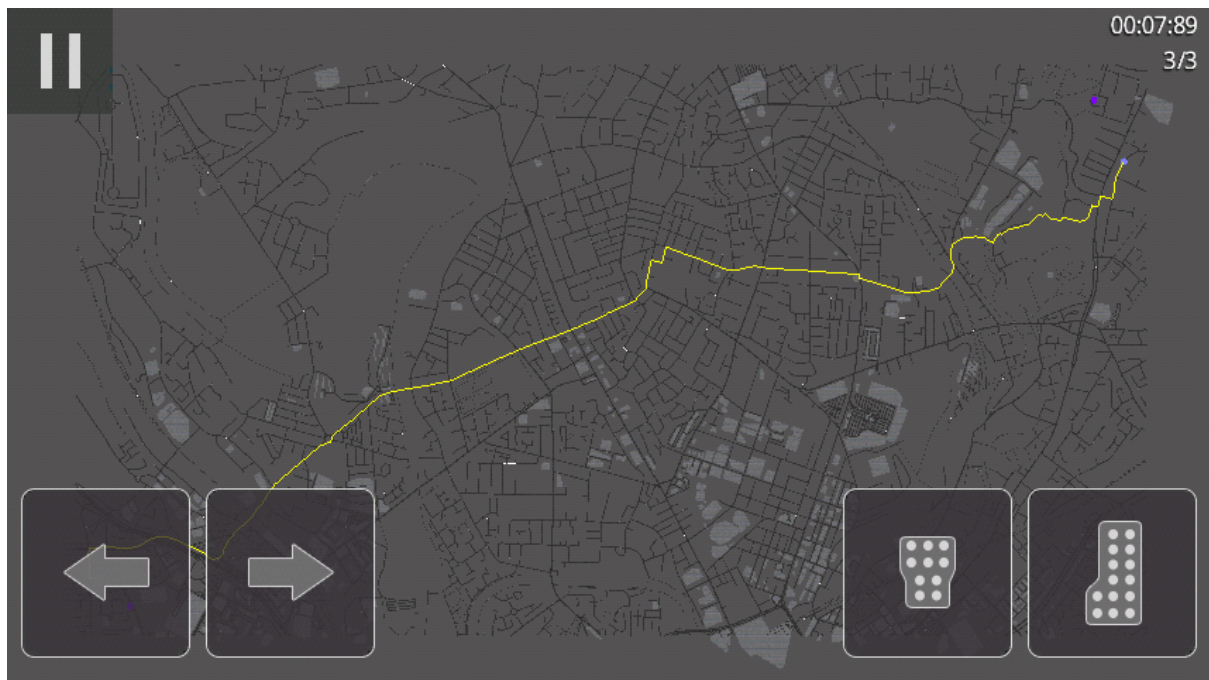
The algorithm took 2.412994385 seconds, which is significantly longer than the A* algorithm. Looking at the result it is obvious to see that the algorithm did not find the fastest path.

The second route is much larger and will put the algorithms through its paces, below is a screenshot of the location:



Starting point: 53.512575, -2.2177799

Finishing point: 53.489397, -2.291937

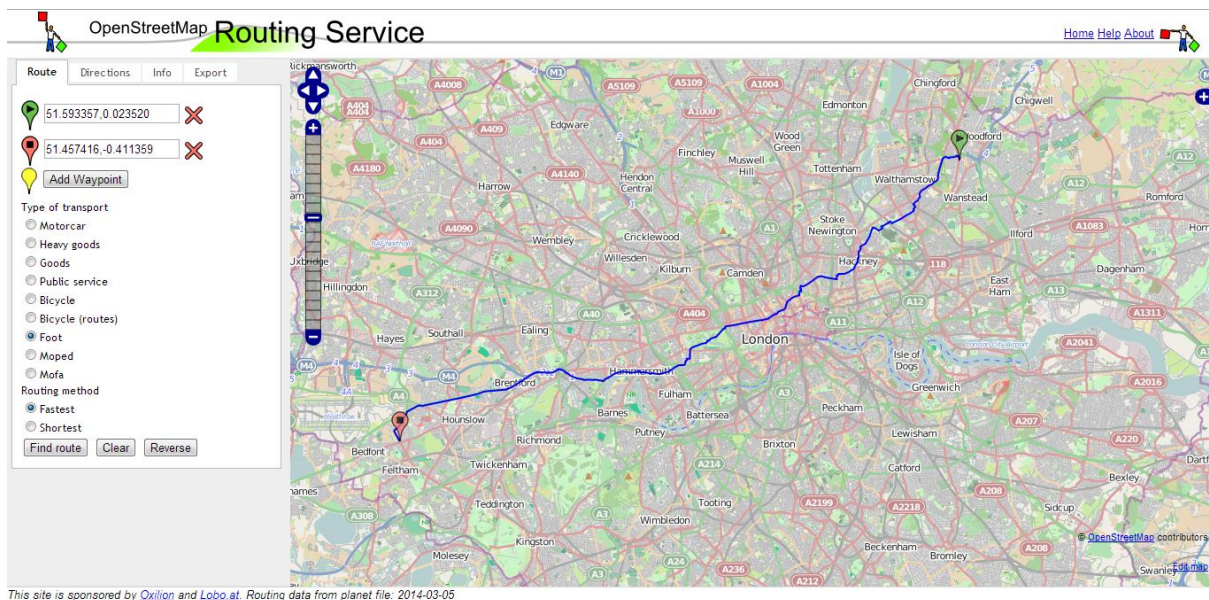


Above is the result for A*. It took 0.549377441 seconds, which as expected took longer than the first test. The route used is more or less the same.



Above is the result for the breadth first search. It looks as though it has found a reasonable route, however the algorithm took significantly longer, 3.339599605 seconds in fact.

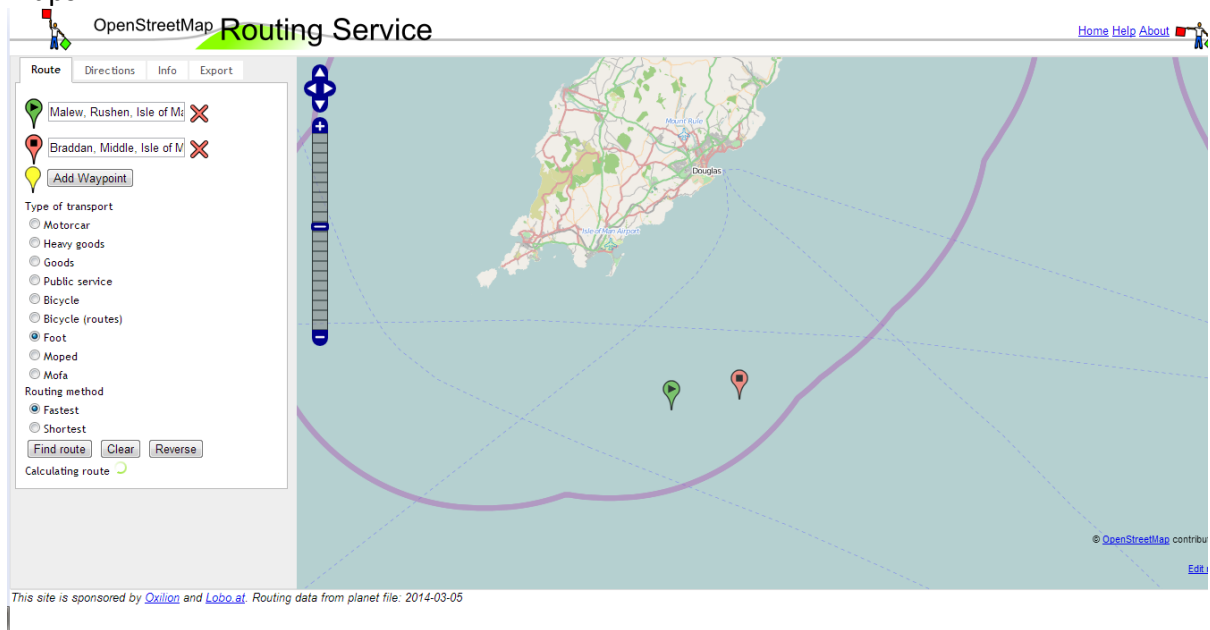
The next test is very extreme, which will fully test the capabilities of the device. It is effectively a race across London.



Starting point: 51.593357, 0.023520
Finishing point: 51.457416, -0.411359

I attempted this test several times and the game more or less crashed. There were some errors with the net interface (probably due to the vast amount of http requests) and some

issues with the memory. I guess it is fair to say that the game does not work with large maps.



Starting point: 53.964064, -4.612885

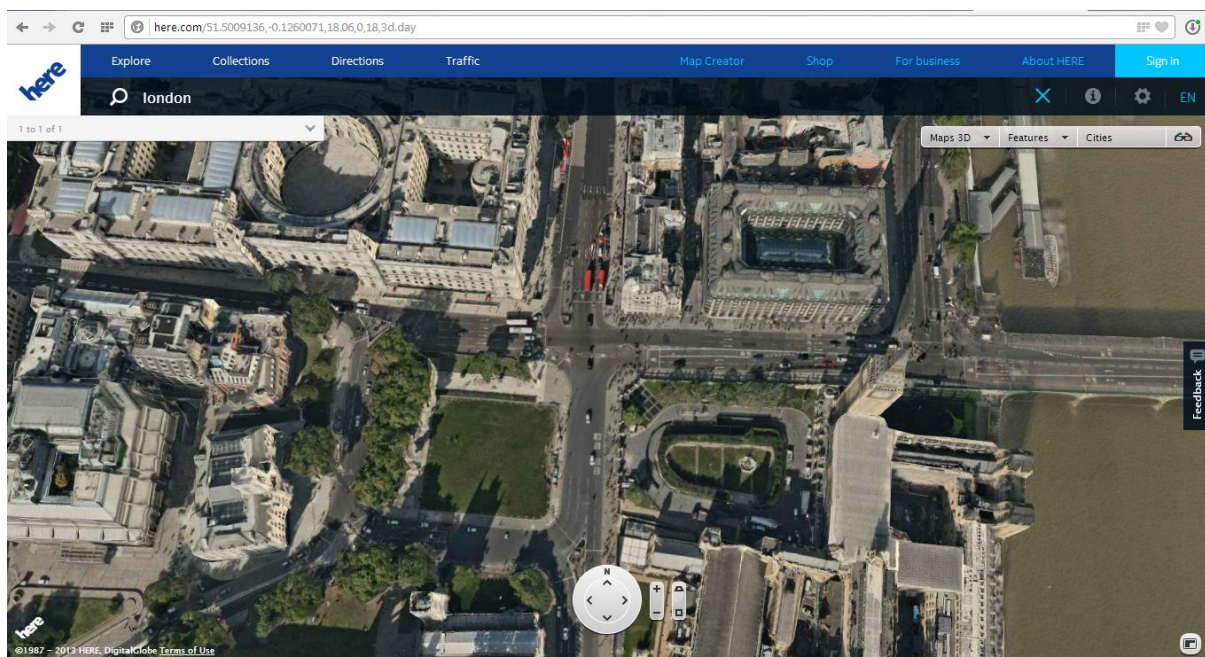
Finishing point: 53.972193, -4.524994

Unfortunately with this test the game crashed. In future there really should be an error message indicating a route was not found.

7. Future Work

There are numerous things that can be improved and added to the game to make it better. Some I have already mentioned at Implementation stage, others have also been identified in the Testing stage by myself and the users that tested my game. Below is a list of these things:

- Improve user interface - the user interface looks pretty good at the moment, however certain screens do not scale properly on larger devices. Despite this the actual game screen looks good on tablets. However the size of the buttons need to be bigger or adjustable depending on the users preference;
- Add textures for specific platforms - at the moment the textures only officially support one resolution. This is more clear in the background in the main menu and other screens, the image does not stretch properly on different screens;
- Improve Artificial Intelligence - the AI is very simple at the moment; in the future I would like to make them more intelligent. They could potentially have a better understanding of how to deal with certain situations such as braking when approaching corners. At the moment it is not aware of its surroundings, it only knows about the path it should follow. In the future it could be given a line of sight which will allow it to see what is ahead and beside it and react accordingly, for example if there is a car next to it, it may decide to crash into it;
- Could make it 3D - I could actually give some depth to the cars and buildings to make them look more realistic. Nokia maps has a 3D map feature that looks pretty good; below is a screenshot of Westminster in central London. You can see the depth of the buildings really well (especially Big Ben). I would however have to do something to remove the cars off the road, perhaps by overlaying another texture (HERE, n.d.);



- Improve physics - in the future I would like to be able to add damage to cars when they crash into objects;
- Multiplayer feature - adding a network features would make the game much more fun; I could add local multiplayer over Wi-Fi or Bluetooth or worldwide multiplayer over the internet;
- Settings - at the moment the settings are only there for developer use. I could add settings to control say the zoom of the camera, the difficulty of the AI players or the graphics settings;
- Take into account bridges, lakes, greenery and other features - Mapnik Vector tiles does have an API to access the water areas and land usages and they could be used to display more features on the map. With regards to bridges, I could have some collision detection that finds if the player is on a bridge. If they are, then they cannot fall off the side, if they are not on the bridge, they will go underneath it. I would also need to add some lighting effects to give the illusion that the bridge is raised, if the game was in 3D, it would be slightly easier to do this;
- Add traffic - adding other cars that are not part of the race would make the game more challenging, not just for the player and the AI, if they are spawned at random places and have random specifications it would also ensure that no two games would be the same, making it more interesting. Another potential idea would be to add police, to chase down the racing players;
- Add sound - sound really enhances the game experience. I would have to take into account various variables to make the engine sound realistic, such as current speed, acceleration etc. I could also add sounds of players crashing and skidding;
- Skid marks - in real life, when a car is travelling fast and it changes direction quickly. Marks are usually left on the grounds, incorporating this in my game would make it more realistic;
- Interactive map - the interactive map to select where the user wants to race is something I really wanted to include, but because of time I couldn't; this would be my top priority when continuing this project;
- Solve frame rate issues - I believe there are issues with the quadtree and they need to be solved to improve the frame rate. At the moment, if it cannot decide which quadrant to put an object in, it puts it in the root quadrant, which causes unnecessary rendering of objects that cannot be seen. Another thing that may need looking at is the physics engine, to see if there are any issues there;
- Put on other platforms - LibGDX is already cross platform, so the application could be run on Windows, Linux, Mac OS X, BlackBerry, iOS, Java Applet or browsers. However, it would be good to also port the game to standalone game consoles such as Xbox 360, PS3, Nintendo DS, PS Vita etc.;
- Ability to choose car - at the moment the user is not allowed to choose what car they wish to race with, they are given the same one each time they race. A good feature to introduce would be the ability to choose a variety of different cars to race with.

8. Conclusion

The aim of this project was to create a racing game for Android devices that utilises OpenStreetMap to generate the map. As a result of this project I have learnt a lot about a number of topics: I feel that my programming knowledge and practice has been massively improved; I now have a much wider knowledge of the features Java offers and put into practice various object oriented methods that I learnt last year such as the MVC pattern, inheritance and abstraction.

The game overall functions correctly, there are a few bugs such as the accuracy of the longitude and latitude, but the game does function as intended. There is a neat user interface, with colours and graphics that would appeal to my target audience. There is a screen that allows users to choose where they want to race, although it was not what I had envisioned, it is still a good compromise. The user can control a car in a top-down perspective, with relatively realistic physics, which allows the player to simulate actions such as acceleration, braking, steering and collisions with objects. The map generation turned out pretty good, much better than I first anticipated, although it was a pain to get it working, the results are good. The pathfinding is also suitable and the AI, although simple, still make good opponents. The fact that the game works well on Android means that I have accomplished my main goal, which was to make the game compatible on Android. There are some issues with the frame rate, which would need to be solved in the future.

In conclusion, I believe that my final year project was a success, I achieved everything I set out to do from the beginning. Although in my initial plan I was uncertain, I eventually decided what I wanted to do and did it. I learnt a lot throughout the process, but I only scratched the surface of the project - there is still a lot more to do in the future to make it better.

9. Reflection

I have learnt a lot by doing this project and am glad I chose the project that I did. My initial plan for the map was to use procedural map generation, although this would have been challenging, it has already been done. Using data from real locations anywhere around the world to generate a map in a game is something that has not been done before (not to my knowledge); it was difficult but I managed to get it working. This area of map generation introduced me to a whole new topic known as cartography, which is making maps, using a combination of science and aesthetics. I learnt a lot about maps that I did not know before such as Mercator projection and in-depth details about longitude and latitude.

I am also glad I also used LibGDX to ease the development process. The ability to deploy the application on the Desktop meant I could test and debug the game quickly, if it did not have that, I would have to debug solely on an Android device, which would take longer. Also the fact that I only had to write my code once and then LibGDX would sort everything else out on the various platforms was a huge bonus. The other features that LibGDX had also helped me a lot, such as the SpriteBatch class to render objects and the input classes to take input from various platforms.

The Box2D physics engine was extremely useful and made my job a lot easier. Without these tools the job of creating a game would be much more difficult, as I would have to create my own framework and physics engine and it would take much longer to do this. So I am very much grateful for these tools. I am also grateful to the Mapnik Vector Tiles, which provided the map data of locations in JSON format. Without it I would have probably have to use the vector data provided in OSM format, which is bulky and would require a lot more processing power to create the maps.

I learnt a lot of other things too; a major one is how significant time management skills are, it is vital that you keep to them. I actually finished the coding for the game a little later than I anticipated, which meant that I spent long hours during the remaining days writing the report and continuing to carry out testing.

I also learnt how important it is to keep a record or diary of the actions I took to completing a task, although I had made notes, I missed out crucial aspects of the development and it took me a while to remember these when writing the report.

I learnt how important design is; going head first into a problem without any preparation is likely to give poor results. I found this particularly when I was implementing the quadtree. I had a brief idea in my head of how it was going to be, coded it and found it did not work correctly. It wasn't until a meeting with my supervisor and a short session of designing the algorithm on paper I fully understood how the quadtree was going to be.

Overall I thoroughly enjoyed making this game, I developed my programming skills, used data structures and methods that I had never used before such as a quadtree. The project has provided me with a solid foundation for a game that has great potential in the future.

10. References

Algorithm and Data Structures, n.d. *Undirected graphs representation - Algorithm and Data Structures*. [Online]

Available at: http://www.algolist.net/Data_structures/Graph/Internal_representation
[Accessed 27 February 2014].

All Android Game Engines, n.d. *All Android Game Engines*. [Online]

Available at: http://mobilegameengines.com/android/game_engines
[Accessed 18 April 2014].

Alpcentauri, n.d. *Pseudocode and Algorithms*. [Online]

Available at: http://www.alpcentauri.info/pseudocode_and_algorithms.htm
[Accessed 28 April 2014].

Amadeo, R., 2013. *Google's iron grip on Android: Controlling open source by any means necessary*. [Online]

Available at: <http://arstechnica.com/gadgets/2013/10/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/>
[Accessed 27 April 2014].

AndEngine, n.d. *AndEngine - Free Android 2D OpenGL Game Engine*. [Online]

Available at: <http://www.andengine.org>
[Accessed 18 April 2014].

Android, n.d. *Android - Meet Android*. [Online]

Available at: <http://www.android.com/meet-android/>
[Accessed 10 April 2014].

Android, n.d. *Android SDK - Get the Android SDK*. [Online]

Available at: <http://developer.android.com/sdk/index.html>
[Accessed 11 February 2014].

Android, n.d. *App Manifest | Android Developers*. [Online]

Available at: <http://developer.android.com/guide/topics/manifest/manifest-intro.html>
[Accessed 2014 April 21].

Android, n.d. *OpenGL ES*. [Online]

Available at: <http://developer.android.com/guide/topics/graphics/opengl.html>
[Accessed 27 April 2014].

Anon., 2013. *Mercator - OpenStreetMap Wiki*. [Online]

Available at: <http://wiki.openstreetmap.org/wiki/Mercator>
[Accessed 13 February 2014].

Anon., n.d. s.l.:s.n.

Anon., n.d. *BFS and DFS*. [Online]

Available at: <https://www.ics.uci.edu/~eppstein/161/960215.html>
[Accessed 28 April 2014].

Anon., n.d. *What is Artificial Intelligence (AI)?*. [Online]

Available at: http://www.webopedia.com/TERM/A/artificial_intelligence.html
[Accessed 28 April 2014].

Association, The Entertainment Software, n.d. *The Entertainment Software Association - Industry Facts*. [Online]

Available at: <http://www.theesa.com/facts/index.asp>
[Accessed 10 April 2014].

Atwood, J., 2008. *Understanding Model-View-Controller*. [Online]
Available at: <http://blog.codinghorror.com/understanding-model-view-controller/>
[Accessed 03 May 2014].

badlogic, 2013. *The life cycle - libgdx/libgdx Wiki - GitHub*. [Online]
Available at: <https://github.com/libgdx/libgdx/wiki/The-life-cycle>
[Accessed 23 April 2014].

BatteryTech, n.d. *BatteryTech - Mobile Game Development Platform*. [Online]
Available at: <http://www.batterytechsdk.com>
[Accessed 18 April 2014].

Bourke, P., 1998. *Clockwise or counterclockwise polygon in a plane*. [Online]
Available at: <http://debian.fmi.uni-sofia.bg/~sergei/cgsr/docs/clockwise.htm>
[Accessed 20 February 2014].

brutalbutler, 2014. *The application framework · libgdx/libgdx Wiki · GitHub*. [Online]
Available at: <https://github.com/libgdx/libgdx/wiki/The-application-framework>
[Accessed 02 May 2014].

Emo, n.d. *emo-framework - open source game engine for Android and iOS*. [Online]
Available at: <http://www.emo-framework.com/index.html>
[Accessed 18 April 2014].

Fisica, 2011. *src/fisica/util/nonconvex/Polygon.java in fisica/fisica:master - Gitorious*. [Online]
Available at:
<https://gitorious.org/fisica/fisica/source/258aadf2c06fadd64ff73ac7b777b6a790573424:src/fisica/util/nonconvex/Polygon.java>
[Accessed 08 April 2014].

Game From Scratch, 2013. *LibGDX Tutorial 9: Scene2D Part 4–UI Skins*. [Online]
Available at: <http://www.gamefromscratch.com/post/2013/12/18/LibGDX-Tutorial-9-Scene2D-Part-3-UI-Skins.aspx>
[Accessed 22 April 2014].

Game Rendering, n.d. *Spatial Data Structure - Game Rendering*. [Online]
Available at: <http://www.gamerendering.com/category/scene-management/spatial-data-structure-scene-management/>
[Accessed 21 April 2014].

GeoJSON, 2014. *GeoJSON*. [Online]
Available at: <http://geojson.org>
[Accessed 2014 February 2014].

HERE, n.d. *HERE - City and Country Maps*. [Online]
Available at: <http://here.com/51.5009136,-0.1260071,18.06,0,18,3d.day>
[Accessed 24 April 2014].

husainhz7, 2014. *Box2d - libgdx/libgdx Wiki - GitHub*. [Online]
Available at: <https://github.com/libgdx/libgdx/wiki/Box2d>
[Accessed 01 May 2014].

iforce2d, 2014. *Box2D C++ tutorials - Top-down car physics*. [Online]
Available at: <https://www.iforce2d.net/b2dtut/top-down-car>
[Accessed 08 February 2014].

ISTQB Exam Certification, n.d. *What are the Software Development Models*. [Online]
Available at: <http://istqbexamcertification.com/what-are-the-software-development->

models/

[Accessed 17 April 2014].

Jönsson, A., n.d. *Bitmap Font Generator*. [Online]

Available at: <http://www.angelcode.com/products/bmfont/>

[Accessed 25 February 2014].

Lambert, S., 2012. *Quick Tip: Use Quadrees to Detect Likely Collisions in 2D Space*. [Online]

Available at: <http://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space--gamedev-374>

[Accessed 27 March 2014].

Lester, P., 2005. *A* Pathfinding for Beginners*. [Online]

Available at: <http://www.policyalmanac.org/games/aStarTutorial.htm>

[Accessed 06 April 2014].

LibGDX, 2014. *LibGDX*. [Online]

Available at: <http://libgdx.badlogicgames.com>

[Accessed 08 February 2014].

LibGDX, 2014. *LibGDX - Goals and Features*. [Online]

Available at: <http://libgdx.badlogicgames.com/features.html>

[Accessed 08 February 2014].

Migurski, M., 2014. *Mapnik Vector Tiles*. [Online]

Available at: <http://openstreetmap.us/~migurski/vector-datasource/>

[Accessed 13 February 2014].

Mr Qwak, 2014. *Retro Racing - Android Apps on Google Play*. [Online]

Available at: <https://play.google.com/store/apps/details?id=com.mrqwak.retroracing>

[Accessed 16 April 2014].

Ninja Cave, n.d. *Slick2D / 2D JAVA Game Library*. [Online]

Available at: <http://slick.ninjacave.com>

[Accessed 18 April 2014].

OpenStreetMap, 2013. *API v0.6 - OpenStreetMap Wiki*. [Online]

Available at: http://wiki.openstreetmap.org/wiki/API_v0.6

[Accessed 13 February 2014].

OpenStreetMap, 2014. *Slippy map tilenames - OpenStreetMap Wiki*. [Online]

Available at: http://wiki.openstreetmap.org/wiki/Slippy_map_tilenames

[Accessed 19 April 2014].

OpenStreetMap, 2014. *Vector tiles - OpenStreetMap Wiki*. [Online]

Available at: http://wiki.openstreetmap.org/wiki/Vector_tiles

[Accessed 13 February 2014].

OpenStreetMap, n.d. *OpenStreetMap*. [Online]

Available at: <http://www.openstreetmap.org/#map=15/51.5114/-3.1666>

[Accessed 23 April 2014].

Playstos, n.d. *Real World Racing - Home page*. [Online]

Available at: <http://www.realworldracing.com>

[Accessed 16 April 2014].

Ribon, A., 2012. *LibGDX Project Setup v3.0.0! | Aurelien Ribon's Dev Blog*. [Online]

Available at: <http://www.aurelienribon.com/blog/2012/09/libgdx-project-setup-v3-0-0/>

[Accessed 11 February 2014].

Ribon, A., 2012. *libgdx-texturepacker-gui*. [Online]
 Available at: <https://code.google.com/p/libgdx-texturepacker-gui/>
 [Accessed 20 February 2014].

Rosenberg, M., n.d. *Latitude and Longitude - Learn Latitude and Longitude*. [Online]
 Available at: <http://geography.about.com/cs/latitudelongitude/a/latlong.htm>
 [Accessed 19 April 2014].

Rouse III, R., 2001. *Gamasutra - Game Design - Theory And Practice: The Elements of Gameplay*. [Online]
 Available at:
[http://www.gamasutra.com/view/feature/131472/game_design_theory_and_practice .php](http://www.gamasutra.com/view/feature/131472/game_design_theory_and_practice.php)
 [Accessed 28 April 2014].

Sedgewick, R. & Wayne, K., 2013. *Undirected Graphs*. [Online]
 Available at: <http://algs4.cs.princeton.edu/41undirected/>
 [Accessed 27 February 2014].

Servo, 2002. *Micro Machines for NES*. [Online]
 Available at: <http://www.mobygames.com/game/nas/micro-machines>
 [Accessed 16 April 2014].

Sheard, T., n.d. *Graphs in Computer Science*. [Online]
 Available at: <http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/Graphs.html>
 [Accessed 27 April 2014].

Sloper, T., 2004. *Sloperama FAQ 5 - Testers -- The Unsung Heroes of Games*. [Online]
 Available at: <http://www.sloperama.com/advice/lesson5.htm>
 [Accessed 23 April 2014].

Software Developer's Journal, 2013. *Artificial Intelligence in Games - CodeProject*. [Online]
 Available at: <http://www.codeproject.com/Articles/14840/Artificial-Intelligence-in-Games>
 [Accessed 28 April 2014].

Sony, n.d. *Xperia Play Specifications*. [Online]
 Available at: <http://www.sonymobile.com/gb/products/phones/xperia-play/specifications/>
 [Accessed 23 April 2014].

Warren, C., 2013. *Google Play Hits 1 Million Apps*. [Online]
 Available at: <http://mashable.com/2013/07/24/google-play-1-million/>
 [Accessed 27 April 2014].

YourNavigation, 2014. *YourNavigation - Worldwide routing on OpenStreetMap data*. [Online]
 Available at: <http://www.yournavigation.org>
 [Accessed 01 May 2014].

Zechner, M., 2012. *Beginning Android Games, Second Edition*. 2nd ed. New York: APRESS.