Joseph Morgan | vt011265 - Individual Project Report

by Joseph Morgan-Rees

Submission date: 26-Apr-2017 01:19PM (UTC+0100)
Submission ID: 71166203
File name: Individual_Project_report.pdf (3.1M)
Word count: 25994
Character count: 134671

School of Mathematical, Physical and Computational Sciences

Individual Project – CS3IP16

Blueprint – An Interactive Indoor Route Planning Application based on the University of Reading Campus

Student:	Joseph Morgan
Student Number:	22011265
Supervisor:	John Roberts
Submission date:	2nd May 2017
Word Count	22,048

Abstract

This report outlines the steps taken towards improving the inaccurate, time-consuming and outdated navigational techniques currently implemented inside large public buildings and establishments, including universities, hospitals and shopping centres. The report describes the design, development and testing processes undertaken to create a brand-new application, Blueprint, which allows its users to view 3-dimensional, interactive models of the buildings. Within Blueprint, users can search for a room by specifying the room number and an entrance; the application then plots the quickest route from the entrance to the room, using the A* search algorithm. Blueprint can display models on all modern devices, including computers, tablets and mobiles, across a range of operating systems.

Acknowledgments

Foremost, I would like to thank John Roberts for his continued enthusiasm and support throughout the project. His eagerness to not only help but steer the project in the right direction has been invaluable.

Secondly, I'd like to give my thanks to the 89 respondents who completed my online survey; the data collected from the results was incredibly beneficial to the project, as it enabled me to scope the future of my application with a much higher degree of accuracy.

Contents

Glossary of Terms & Abbreviations

2D – 2-Dimensional

3D – 3-Dimensional

API – Application Programming Interface

CAD – Computer Aided Design

CPU – Central Processing Unit

CSS – Cascading Style Sheets

FPS – Frames per Second

GIS – Geographic Information Systems

GPS – Global Positioning System

GPU – Graphics Processing Unit

GUI – Graphical User Interface

HTML – HyperText Markup Language

IDE – Integrated Development Environment

IPS – Indoor Positioning System

JS – JavaScript

JSON – JavaScript Object Notation

PID – Project Initiation Document

UI – User Interface

UoR – University of Reading

URL – Uniform Resource Locator

1. Introduction

The sole aim of this project is to improve current navigation techniques implemented inside large buildings and establishments, more specifically the buildings at the University of Reading. Current techniques are inaccurate, time consuming and outdated, with very little changes being made since the buildings were first erected. The most common way of finding a specific room inside a building is to study the building's floor plan and memorise your route from the entrance to your room, using waypoint markers along the way. In a similar scenario, Global Positioning Systems (GPS's) have revolutionised travel, and have minimised the need to use a physical map for directions, another technique that has been in use for hundreds of years.

The motivation for the project stems from constantly overhearing how students have got lost whilst trying to find specific rooms on campus. It was also glaringly obvious how prospective and new students would struggle to navigate the more complex campus buildings, such as the HumSS building (soon to become the Edith Morley building). An application of this type would also be great for open days. In the future, this application could even be used in other large, public establishments, such as hospitals, shopping centres, libraries and airports.

To improve upon the current methods used, the system developed by this project must surpass current methods in terms of accuracy, speed and ease of use. If it does not meet these expectations, then it will be regarded as a failure, as it cannot be deemed as a solution to the existing problem. This report outlines a technological solution; the development of an application that allows users to search for a specific room inside a 3D model of a building, using their phone, tablet or computer. This method of indoor navigation has not been attempted before and therefore very little research exists.

To achieve the objectives specified in the PID (see appendix 1), the project must be split into two parts; the first is to develop or modify an existing application to build the 3D model buildings, whilst being able to specify key details about the building such as room numbers, toilet locations etc. The models created must also be aesthetically pleasing, by ensuring the modeller can create wall and floor coverings, for example. The second part is to develop an application that will allow users to interact with the 3D models. The users will be able to search for particular rooms inside the models, whilst also being able to rotate and enlarge them. The users will be able to interact with the models on computers, tablets and mobile devices, across a range of operating systems. The combination of these two applications will structure the output for this project.

This report will guide the reader through the existing problem, how the solution was conceived and implemented, and how the final application can be applied to many real-life situations, as an exciting, innovative product.

2. Problem Articulation & Technical Specification

2.1 Problem statement

Outdoor navigation has changed dramatically since the beginning of the new millennium, with many technological solutions working in harmony with existing paper maps and road signs. However, the world of indoor mapping remains stagnant and unevolved, which is a problem that needs to be solved. Many large establishments are still relying on outdated physical wall maps and verbal communication for effective indoor navigation, both of which present their own problems.

Physical signs are common place; they are used to direct people in almost every public building, from libraries to airports. While many signs are clear and detailed, others can be vague and poorly placed, which can lead to confusion amongst the many that try to use them. Another huge problem with physical signage is scalability; if the rooms inside the building are rearranged, or the building is extended, many of the signs will become redundant, or will have to be modified. Also, having too many signs can often lead to contradictory, which will undoubtedly lead to misunderstanding when being used by visitors. A further issue with physical floor plans is that they are often only located near entrances to the building, meaning they are effectively useless for visitors elsewhere in the building.

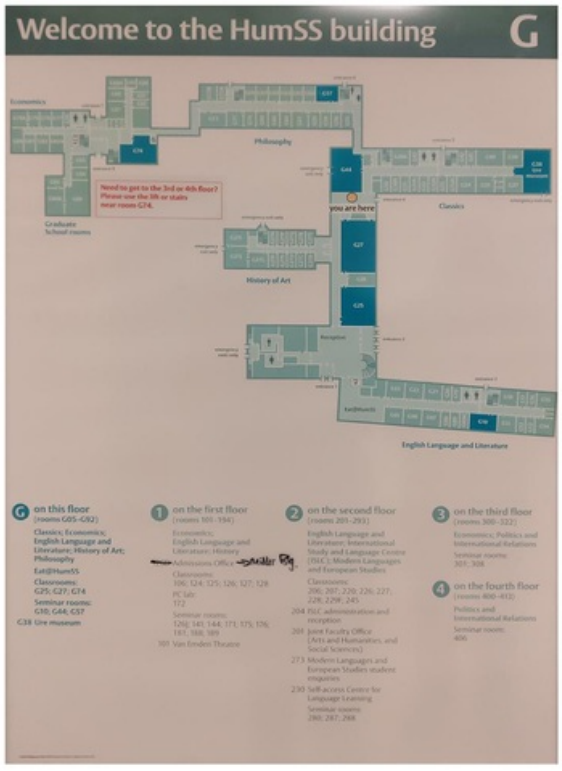


Figure 2.1.1 – The HumSS building floor plan

Verbal directions pose their own issues; when asking someone for directions inside a building a visitor must first find somebody that will know the location of the room they are searching for. The visitor must then rely on the clarity of the given directions as well as the director having recalled the directions correctly. As with signage, once the visitor has received the directions, they must rely on their own memory and ability to follow directions, in order to find the room.

In very large establishments, such as airports or hospitals, visitors can easily get lost within a building, which is shocking when considering the recent advancements in technology and the far superior technology harnessed by outdoor GPS navigation. This problem is also evident throughout the campus of the University of Reading. Many buildings have a very complicated layout, with insufficient floor plans and directions within the buildings. Figure 2.1.1 above shows the HumSS building (soon to be the Edith Morley building), one the most frequently complained about buildings on the Reading University campus, due to its complicated layout and lack of clear directions.

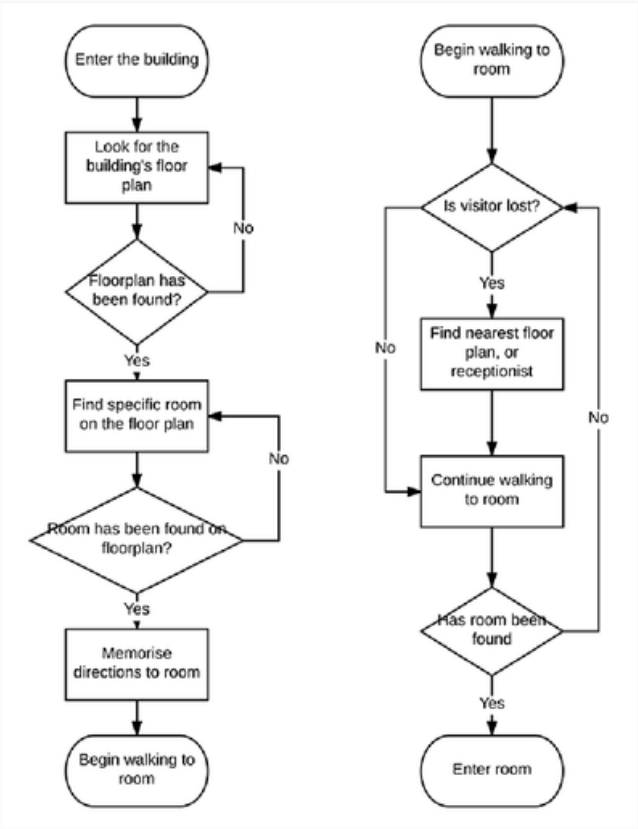


Figure 2.1.2– Situation-as-is (Entering a building/finding a room)

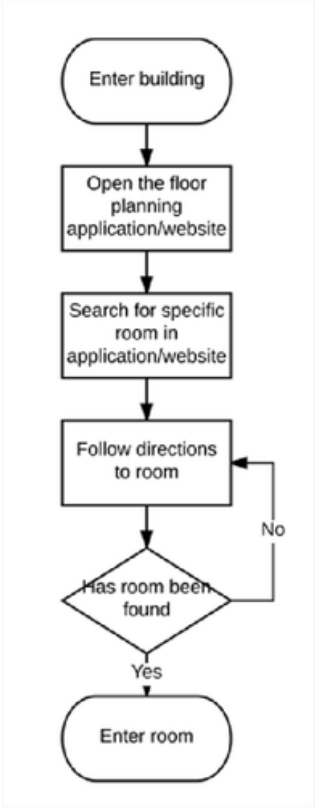


Figure 2.1.3 – Situation-to-be

Figure 2.1.2 above shows the current, time consuming situation a new visitor must face when entering a building they are unfamiliar with. Figure 2.1.3 shows the proposed solution, with the use of a room finding application or website to guide the visitor, removing confusion and saving time. The visitor would enter a source and destination into the application, i.e. the entrance they are located at and the room they wish to find. The application will then display a 3D model of the building, highlight the specified room and plot to quickest route to the room from the specified entrance. The application would not replace physical signs; it would rather work with them, to *enhance* the visitors experience.

2.2 Technical specification

The application will be hosted on a website, with the ability to convert it into a pair of native applications for both iOS and Android. As previously mentioned, the application will be split into 2 parts; firstly, a basic 3D modelling suite used to create 3D buildings that can be viewed in the web browser, and secondly, a room finding application that allows users to search for rooms inside the 3D model buildings. Proposed and existing solutions and coding languages are addressed in section 3.

Adhering to the Project Initiation Document (PID), the project should satisfy a number of objectives. The objectives listed below are revised versions of those taken from the original PID, updated to align with the realistic scope of the project:

Primary output – A room finding application for the Reading University Campus buildings:

- Must display 3D models of the main campus buildings
- 3D models must show the rooms in the buildings clearly
- 3D models must be interactive i.e. be able to be rotated, enlarged etc.
- 3D models must highlight the particular room that is searched for
- Application must allow users to search for a particular room inside the model of the building
- 3D models should highlight staircases and main building entrances, to provide a basic route from the entrance to the chosen room
- The application should be user friendly and have an aesthetically pleasing user interface
- Application should work on all recent versions of iOS and Android operating systems
- Application could include basic information about the building e.g. opening times
- Application could be opened from other apps e.g. calendar app on iOS

Secondary output – A basic modelling suite used to create 3D buildings:

- The suite must allow users to create individual rooms of a building
- The suite must allow each room to be given a unique identifier, used by the search function
- The suite should allow users to add extra floors to existing floors, rooms can then be added to these extra floors
- The suite should allow users to add basic styling to the 3D models i.e. doors, brickwork etc.

2.3 Stakeholders

There are several different people that could present an interest in this type of application, ranging from those who visit the University on a very regular basis, to those who many only ever visit it once.

2.3.1 The developer (Joe Morgan)

The developer is concerned with ensuring the project runs smoothly, the application is developed on time, and that all requirements have been met.

2.3.2 The University of Reading (UoR)

The University of Reading can be compared to a living and breathing organism, due to its regular change in students and the constant expansion of its campus. Each year thousands of people visit the campus, from existing students, to prospective students with their parents, to guest speakers. The University would not be able to monetise an application such as the one proposed in this report, however it would greatly improve the user experience of navigating around the campus. As there are no other universities in the United Kingdom (or potentially the world) that harness such a useful application, this could set Reading University as a leader in the rapidly evolving field of indoor navigation.

2.3.3 Existing students and members of staff at UoR

It's estimated that the University of Reading is home to around 15000 students (2015/16) [1], and a very large staff force, most of which will visit the campus on a regular basis. Regardless of the frequency these people visit the campus, there are, without a doubt, areas and buildings that they will be unfamiliar with. An application such as the one proposed in this report could greatly improve their experience at Reading University, especially if the application was extended in the future to included important extra details about that buildings, that the staff and students could benefit from.

2.3.4 Parents and prospective students of UoR

When visiting the campus for the first time, such as on an open day, it is possible that parents and prospective students could find themselves lost. If there were an application that could help them find where they are supposed to be, it would greatly improve their experience of the University of Reading and could even play a key role for the student when choosing their university. Similarly, newly enrolled students, who have not had experience in navigating the university's campus or potentially any large establishment, would benefit hugely from this application, as it would save them a great amount of time and confusion when trying to find a specific room inside a building they have never visited.

2.4 Project motivation

The motivation for this project has evolved over the past 4 years. The developer initially proposed a PID describing a Google Maps like navigation application for the University of Reading, however after extensive research it was deemed that such a project would be too unchallenging. The original PID described the idea of allowing users to search for rooms inside the buildings on campus, and after considerable research in this area it was decided that an application of this type *could* be developed.

It' is well know that certain buildings at the University of Reading are difficult to navigate, such as the HumSS (Edith Morley) and Systems Engineering (Polly Vacher) buildings, so the introduction of an application such the one described in this report could be hugely beneficial, to staff, students, parents and other visitors that may need help with navigation.

The application also has a huge amount of scope for the future, as certain details about each building can be added, such as disability information, opening times and other points of interest. This type of application could also be incredibly useful in other large establishments, such as airports and hospitals.

2.5 Project constraints

Perhaps the most obvious constraint on this project will be gaining access to the floor plans to the buildings on the University of Reading campus. Suitable emails will have to be sent the Estates and Facilities department at the University of Reading, in an attempt to gain access to full scale, PDF versions of the floor plans. If this is not possible, then the images of the floor plans located inside the buildings will have to suffice.

Another major constraint will be the ability to created detailed 3-dimensional models that can be viewed inside a web browser, using existing technology. A lot of research will have to be conducted into this area to ensure the correct solution is developed.

3. Literature Review

3.1 Existing technologies – Model viewers

3.1.1 2-dimensional floor plans

Online digital mapping technologies have been around for the best part of 20 years, thanks to Geographic Information Systems (GIS). Expanding on this technology, the invention of online route planning services can be pin pointed to around 1996, with the invention of MultiMap, which progressed to become Bing Maps in 2010 [2]. Towards the end of the 20th century there was a noticeable increase in the creation of these route planning services, such as the AA Route Planner in 1999 [3], most of which were based on the GIS technology. Proceeding to the mid 2000's saw the invention of Google Maps, created by Where 2 Technologies, before being acquired by Google in 2004.

The invention of these technologies has transformed the way people travel, with Google Maps having reportedly over 1 billion users in 2014 [4], however indoor route planning is still a rather undocumented subject. Despite this, there are many applications that are currently attempting to solve the problem of indoor navigation. The most evident solution is Google's 'Indoor Maps' integrated with Google Maps. Indoor Maps allows its users to zoom in to buildings to see the associated floor plan, from here the user can search for specific rooms, switch between different floors and even get directions between 2 places on the map. The idea was first published in 2011 [5], however since then very little has changed; its flooring system feels clunky, the rooms are virtually indistinguishable from one another and it's 2-dimensional appearance makes it feel very outdated.

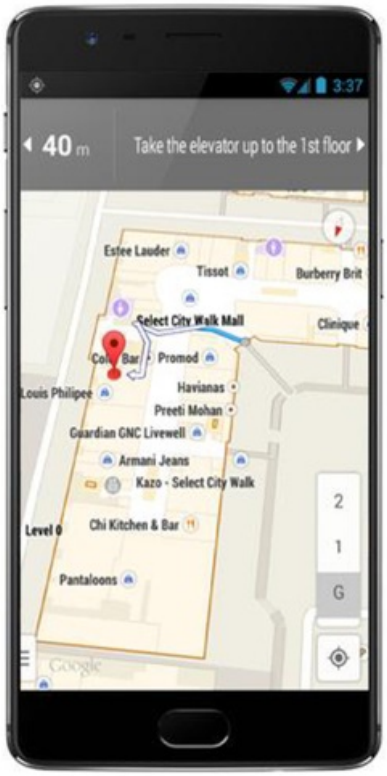


Figure 3.1.1 – Google Indoor Maps

Other companies, such as MapsPeople [6] have built on top of this technology to create a slick route finding algorithm, with an improved user interface (UI), when compared to the standard Google Indoor Maps UI. However, the application's UI still makes it surprisingly difficult to distinguish between different rooms and corridors when looking at a building from a bird's eye view.

Mapwize is another 2D indoor navigation application, claiming to be 'the multi-purpose indoor mapping platform for smart buildings' [7]. It allows its users to upload floor plans and add different rooms as points of interests. Once published, the floor plans can then be downloaded and used for navigation. However, its pitfalls are similar to those of Google's Indoor Maps; a poor UI coupled with the need to download a native application, which makes for a poor overall user experience.

3.1.2 3-dimensional floor plans

Evolving from 2D floor plans, 3D floor plans make for a much greater user experience, however they can often suffer from having an over complicated UI. One of the only examples available is the Reece app created by eeGeo [8].

Figure 3.1.2 – eeGeo's Reece application displaying 3D indoor mapping

Reece has a remarkable UI that displays a huge number of buildings in 3D, which looks extraordinary when browsing the application's map. The app also allows its users to zoom further into certain buildings, displaying a 3D floor plan with the option to switch between floors. The 3D floor plans are powered by Micello, who claim to have mapped over 15,000 venues [9], however the models appear undeveloped and almost childlike. The overlapping text overcomplicates the models and there appears to be no consistency with the colouring. Furthermore, there is no ability to plan routes between the rooms, making it purely useful for visual purposes.

3.2 Existing technologies – Modelling suites

3.2.1 Existing Modelling Suites

For the proposed project the modelling software chosen must not only allow the modeller to build the 3D models, but also allow them to specify the different rooms inside the models. Finding a modelling suite that can do both these things will be the biggest challenge when choosing a suitable software package. That being said, there are many desktop programs that can be used to build the 3D models.

Both Blender and SketchUp are great examples of 3D modelling suites. They contain all the tools that would be needed to design and create a model building, as well as having very lenient licences, allowing the user to use the models they've created to generate money.

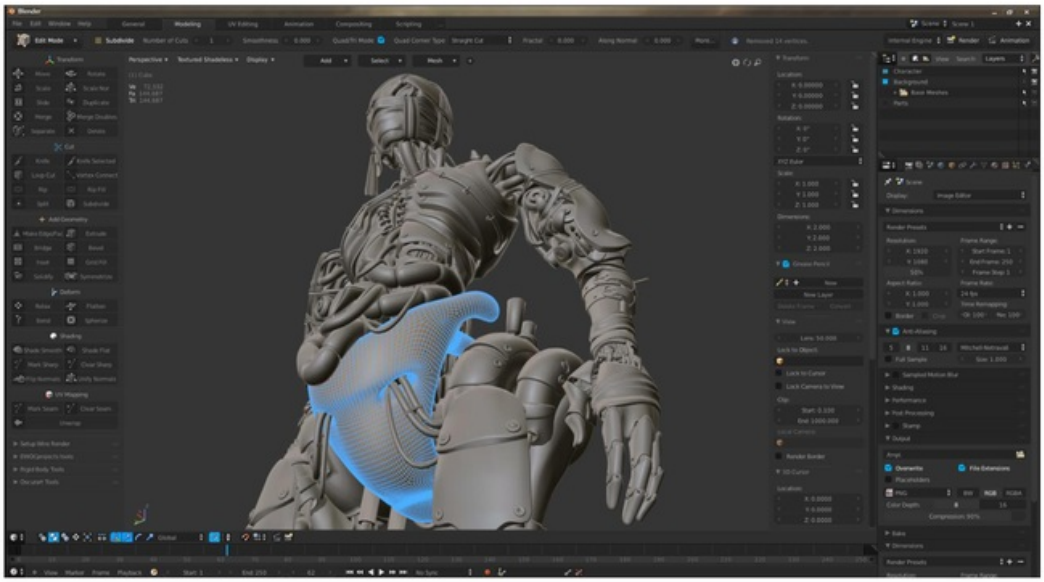


Figure 3.2.1 – Blender's complex graphical user interface

The major disadvantage of using a pre-existing modelling suite is that the modeller would be unable to specify the room number, when a room has been created. The project is based upon a finding a particular room inside a 3D model, so without being able to determine which room is which, the 3D model would effectively be useless. There are also a few more trivial drawbacks to using an existing modelling suite; firstly, there is scope for the application developed in this project to allow the modeller to auto generate their own floor plans, something that would be difficult in an application like Blender without developing an extra plugin. Secondly, it's a well-known fact that learning to use complex programs like Blender can take many, many years [10].

3.2.2 Bespoke Modelling Suite

An alternative way of creating the 3D model buildings would be to develop a bespoke 3D modelling suite for the *exact* purpose of creating the 3D buildings. As the finished models will be viewed in a web browser, JavaScript would be the most obvious language of choice to create the 3D modelling environment.

WebGL is a JavaScript API used to render 3D graphics within a web browser, without the use of external plugins. Perhaps the most important feature of WebGL is its ability to utilise the machine's GPU (Graphics Processing Unit) [11], allowing the CPU (Central Processing Unit) to be used concurrently for other tasks. WebGL works on many handheld devices, with the first Apple device supporting WebGL being shipped with iOS8 in September 2014 [12]. Many programming frameworks are exploiting the capabilities of WebGL, allowing their users to create powerful 3D applications, executable entirely in the web browser.

Three.js, developed in 2009, is viewed by some as the most established WebGL framework, with extensive documentation and a large collection of example projects. Three.js has been used on a number of projects, from interactive music videos [13] to 3-dimensional vehicle modelling [14]. It's important to note that Three.js is mostly used for 3D graphical rendering, such as 3D model animation, as opposed to game development [15]. Three.js has been developed under the MIT licence, which allows its users to monetise the models and applications they create with the framework.

Figure 3.2.2 – Ford's 3D Mustang visualizer built in Three.js

Babylon.js was created more recently by an employee at Microsoft and is very similar to Three.js. It is primarily used for gaming purposes, as it focuses on the traditional game engine requirements, such as engines and custom lighting. Like Three.js, it boats a great documentation database and an active developer community. Babylon.js is developed under the Apache licence, meaning the users can export and sell the models and games they create, like in Three.js. Many large companies, such as Xbox [16] use Babylon.js to promote their products in 3 dimensions.

PlayCanvas.js can be defined as an "enterprise grade open source JavaScript based WebGL game Engine" [17]. It can be used to develop much larger, more details projects, than Babylon.js or Three.js. However, a premium subscription is needed to create models and applications privately.

3.2.3 Indoor Positioning Systems

An Indoor Positioning System (IPS) is a system used to locate devices inside a building, using sensory information collected by the device. The systems are usually very complicated to implement, however if done correctly they can be an incredibly powerful and useful tool. The fundamental problem with IPS's is the technology they use to determine a device's position, as this hinders the accuracy of the IPS. Most solutions use Wi-Fi or Bluetooth, which have an accuracy of around 5-10 meters, which is not always enough for a precise position.

Some companies have moved away from Wi-Fi and Bluetooth onto more advances solutions; IndoorAtlas use geomagnetic technology (the earth's magnetic field) to determine a device's position, providing an accuracy of 1-2 meters [18]. It is a software only solution, meaning no additional hardware is required to map a building, as well as being a cross platform solution.

FIND (The Framework for Internal Navigation and Discovery) is a framework that allows indoor positioning on an Android device. The system works by sending specific data to a machine learning server, which in turn analyses the data. The device's location is calculated before being sent back to the device and displayed on the screen. This approach to indoor navigation works great in theory, however, as it uses Wi-Fi to determine location, it is not the most accurate solution available.

A 2016 paper written by Kaifei Chen and Karthik Vadde at the University of California proposes an IPS framework that combines multiple localisation systems, as opposed to simply using an individual one. Having deployed the solution in their campus buildings, they found that it generated suitable results; however, according to the paper, old indoor positions systems still have "huge potential" [19].

3.3 Creating native applications

To achieve the best user experience, the application created to view the 3D models will be executable across a range of devices and operating systems. Ideally, the application will be native to Android and iOS devices, however it should also work in the devices web browser, depending on which framework is used to write the application. There are a number of methods that can be used to write a native application, most applications are written in the device's native language; Swift for iOS and Java for Android. It would be impossible to develop these 2 applications native applications in the timeframe allowed for this project, however alternatives are available.

PhoneGap, based on Apache Cordova, is an open source library that allows developers to create native mobile applications using HTML, CSS and JavaScript, meaning applications can be run on an abundance of devices with the same code. "The UI layer of a PhoneGap application is a web browser view that takes up 100% of the device width and 100% of the device height" [20]; as PhoneGap applications use the devices web browser, they can utilise WebGL technologies and frameworks, such as Three.js or Babylon.js. PhoneGap also allows the code to access native devices features, such as the camera or digital compass. There could however be some disadvantages to using PhoneGap; applications packaged for Android devices using KitKat (or any previous version) will use the standard Android browser. The Android browser does not support WebGL technologies, so the application will not load. Furthermore, the JIT complier is not supported in iOS, which means the JS code cannot exploit the full performance optimisations of the browser engine [21].

An alternative to developing two native applications would be to ensure the web application is mobile friendly and works as well as an application, but in the web browser. If done correctly, this would have the advantage that users will not have to download the application to their device, therefore saving space on their device. It would also allow for 'one time use', for buildings such as airports or museums. Certain frameworks, such as jQuery mobile allow its user to create responsive web sites and applications that are accessible in a wide variety of devices; jQuery code can also be used by PhoneGap, allowing the potential of creating native applications in the future.

3.4 Searching algorithms

To find the shortest path between a specified entrance and room, a shortest path algorithm will be need to be implemented into the application. The algorithm should calculate the shortest path and clearly display is on the device for the user. Obviously, the algorithm should also work when there is more than one path available. After conducting research into the different types of shortest path algorithms, two were clearly dominant, in terms of speed and ease of implementation.

3.4.1 Dijkstra's algorithm

Dijkstra's algorithm finds the optimal path between two vertices on a graph, or for this project, from a building's entrance to the specific room's door. In simple terms, starting at a source (the entrance), the algorithm will calculate the cost of visiting every neighbouring vertex (the points in the corridors), until it has visited every vertex. From here it will calculate the shortest path from the starting vertex, to the ending vertex.

$$f(x) = g(x)$$

Figure 3.4.1 – Dijkstra's algorithm where g(x) is the cost to reach vertex x

In terms of speed, Dijkstra's algorithm isn't one of the fasted, clocking in at O(n^2), where n is the number of vertices; however, with the size of the data set that will be traversed (the entire floor plan), this speed is more than enough (for a computer). There are also many libraries that already exist in JavaScript that could be used to implement Dijkstra's algorithm, if JavaScript was chosen to be used.

3.4.2 A Algorithm*

"The A* algorithm is a modification of Dijkstra's algorithm that uses a heuristic to determine which vertices to search" [22]. The heuristic is effectively an estimated cost from the vertex you're searching to the target vertex. A* will only search the neighbours of a vertex if it seems likely they could be part of the shortest path. Like Dijkstra's algorithm, there are many libraries that currently exist in JavaScript, however when using the optimal heuristic, the time complexity is O(n), where n is the number of vertices, making it the optimal algorithm to use, when compared with Dijkstra's [23].

$$f(x) = g(x) + h(x)$$

Figure 3.4.2 – A algorithm where h(x) is the approximate cost from vertex x to goal vertex*

3.5 Market research

3.5.1 Survey results

To investigate the need for indoor navigation, a survey was conducted regarding the possible improvements that could be made, from the public's point of view (see appendix 3 for full survey results). 8 questions regarding indoor and outdoor navigation were asked to 89 members of the public, of different ages, and a range of answers were recorded.

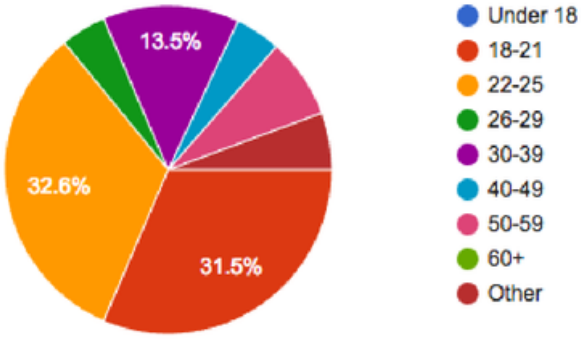


Figure 3.5.1 – The age span of the respondents

Firstly, a very interesting difference could be drawn between indoor and outdoor navigation techniques. When asked how they would most likely plan their route if travelling (by foot/car/bike etc.) to a place they had not been before, 96% of the respondents said via GPS (Google Maps/Apple Maps/Waze etc.). Whereas when asked how they would find a specific room inside a building/establishment that they were unfamiliar with, 78% said by using the floor plan inside the building. Other answers for this question included guesswork, asking for directions or by downloading the floor plan from the internet. What's interesting is that the general public appear to rely heavily on GPS for outdoor navigation, whereas they still rely on outdated, vague physical maps for indoor navigation.

Unsurprisingly, when asked if they had ever had trouble finding a specific room inside a building/establishment, 96% agreed that they had. Digging deeper it was found that university buildings and hospitals were the main culprits, scoring 77% and 80% respectively, with airports, shopping centres and workplace buildings scoring a lower 30%.

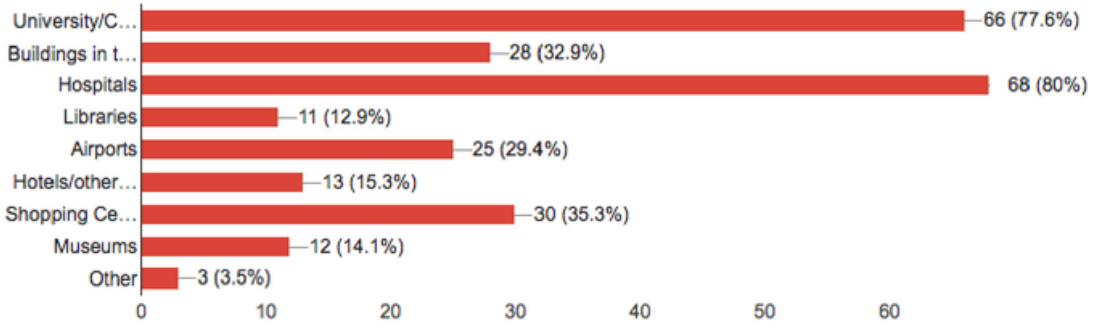


Figure 3.5.2 – Results when asked the type of buildings that people had trouble navigating

To try and predict the scope of the application, respondents were asked whether they would use a room finding application, if one existed. Similarly, to the results collected by previous questions, 66% said they would, with 32% stating they would consider it, depending on certain functionality.

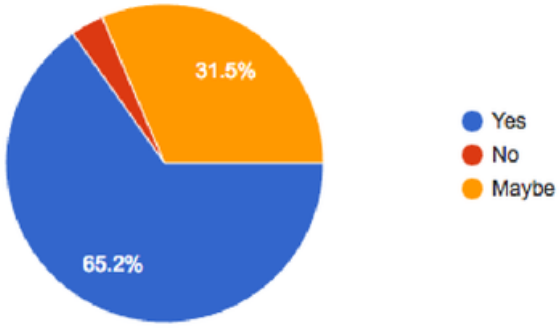


Figure 3.5.3 – Results when asked if they would use a room finding application

58 respondents gave examples of specific features they believed should be present in the application; by far the most requested feature was the ability to find the shortest route between an entrance and a room. Expanding on that was the potential to include a guidance system, similar to that of a GPS, as well as an estimated time of arrival. It was also mentioned that the models should include certain points of interest, such as toilets, stairways, disabled entrances and water fountains.

With regards to the application itself, many respondents stated that it would *need* to be an incredibly easy application to use; one respondent said that "the app would have to be efficient enough to compete with the ease of simply asking a receptionist". Another respondent agreed, stating that developing the application could potentially involve "more effort than just increasing physical navigation aids present in buildings (signs, maps, arrows)." Many people also said they would not be keen to download a native application, and would prefer the application to work in a web browser, as well as having the ability to search the 3D models when no internet connection was available.

Finally, there was heavy interest in the ability to interact with the rooms themselves, as opposed to just having them as static objects in a 3D model. Ideas such as displaying the times the room is in use, being able to book the room for a meeting and integrating the application with the device's calendar were all mentioned as being features that could improve the application.

As this project is focusing on The University of Reading, the survey asked which buildings on campus were most difficult to navigate, if applicable to the respondent. The results were unsurprising, with 39% stating the HumSS building, and a following 19% stating the URS building. 18% of the respondents also chose The Systems Engineering building.

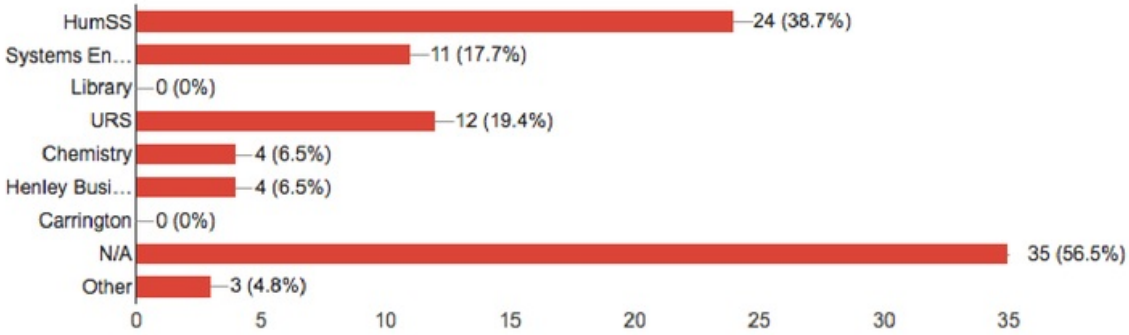


Figure 3.5.4 – Results when asked about the difficulty of navigating The University of Reading buildings.

Overall, the data collected from the survey has reinforced the findings of the initial research conducted by the developer; there is a very large market for an application like this, but only if it is developed correctly.

3.5.2 Oxford University Hospital

Leading on from the results collected from the survey (listed above), the complications presented with trying to find a specific department inside a hospital were later reinforced; after making a visit to The Churchill Hospital, one of Oxford University's many hospitals, it was noticed that one of the pieces of feedback stated that the department was "tricky to find". The hospital responded with "new signs are on their way" – this feedback isn't very useful; what is the hospital going to do in the meantime? What happens to the signs if the department changes locations? What if the patients cannot read the new signs? It's also worth noting that this is data collected from one department in one hospital, meaning it's very likely to be happening elsewhere.

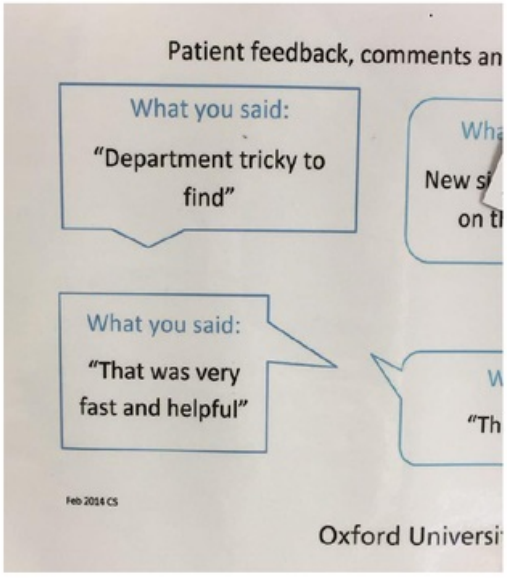


Figure 3.5.5 Oxford University Hospital feedback

4. The Solution Approach

This section will present a suitable development process that will be used throughout the implementation of this project. The tools and technologies used to develop the application will also be explained and justified. The issues and solutions raised in the problem statement, technical specification and literature review will all considered when developing an appropriate final solution, as well as the time allowed for the project and the experience of the developer.

4.1 Coding languages & frameworks

As previously mentioned, the 3D models will be displayed in the user's web browser, meaning the WebGL JavaScript API will have to be utilised. Because of this, JavaScript will obviously be the most appropriate coding language to use for this project. There are several WebGL frameworks, as described in section 3.2.2, that would allow for the viewing and interaction of 3D models in the web browser; Table 4.1 below compares and contrasts these frameworks, highlighting their advantages and disadvantages:

Framework	Advantages	Disadvantages
Three.js	- The most established WebGL framework - Many online demos/examples - Comprehensive documentation - MIT licence - Suitable for small projects - Active developer community - Free to use	- Limited backwards version compatibilities - Maybe be hard to set up initially
Babylon.js	- Many online demos/examples - Comprehensive documentation - Apache 2.0 licence - Very active developer community - Free to use	- Primarily used for large projects and developing games
PlayCanvas.js	- Networking and multi-user capabilities - Many online demos/examples - Comprehensive documentation - MIT licence	- Primarily used for large projects and developing games - Not free to use - Active but potentially unhelpful developer community

Table 4.1 - Comparison of WebGL frameworks

When comparing the 3 possible frameworks, Three.js appears to be the most appropriate one to use for this project. While Three.js and Babylon.js offer almost identical opportunities, research suggests that Three.js is more suited to smaller projects while Babylon.js appears to be more appropriate for larger, game like applications [24]. Fortunately, Three.js has a very active developer community and a comprehensive library of documentation. The developer also has previous experience with JavaScript, making it an even more appropriate choice, however the key features and intricacies of the Three.js framework will have to be self-taught throughout the project.

Using Three.js will solve the first problem outlined in the PID (see appendix 1); the ability for the user to view and interact with 3D models in the web browser. The PID also defines the need for the user to be able to search for specific rooms inside the building, as well as for the buildings being visually pleasing. If used correctly, Three.js will also be able to provide an appropriate solution to both of these issues. If this project was developed further in the future, it may be more appropriate to use Babylon.js, however Three.js will be suitable for now.

HTML5 and CSS3 will be used in conjunction with Three.js to display the models in the browser. A separate website will also be created to allow users to choose which models they wish to view, again written in HTML and CSS. The application will be hosted on a web server, and will be given a specific URL for access.

4.2 Development process

As the developer has previous experience with the agile methodology, a few of it' major principles will be adopted during this project. The Kanban approach will primarily be used, as it is more appropriate than Scrum for this type of project [24]. The work items, such as features to be implemented, or bug fixes, will be split into tickets. Each ticket will be given a progress state, such as 'to do', 'in progress', 'ready for testing', or 'complete'. This will allow the developer to keep track of the features and bugs within the project and ensure they stick to the allocated timeframe. The key features of agile/Kanban that will be implemented are as follows; visualising the workflow, allowing the requirements to change throughout the development of the application and ensuring that outstanding features are completed before work on another feature is started. It's hoped that this method of development will produce the highest quality project, within the timeframe.

As mentioned in previous sections, the application will be split into two sections, a modelling suite and a model viewer. For obvious reasons, the modelling suite will have to be developed prior to the model viewer; however, the solution for the model viewer is far easier to depict, due to the limited technologies available to use. For this reason, a reverse programming type of method will be adopted; the solution for the model viewer will be devised first, with the solution for the modelling suite being based upon this solution.

4.2.1 The model viewer

The process of how the user will interact with the model viewer is outlined below:

- The user will select the correct model
- The application will load and display the chosen the model
- The user will search for a room within the model, specifying the room and entrance number
- The application will display the shortest route from the specified entrance to the specified room

There are many implementation issues that will need be addressed in the above sequence. Firstly, the model created by the modelling suite will need to be in an appropriate format, for it to be loaded into the model viewer. JSON (JavaScript Object Notation) will most likely be used for this; if the modelling suite allowed for the model to be saved as a JSON file, the entire process could in theory be reversed when the model is loaded into the viewer. The link between the modelling suite and the model viewer will be described in greater detail in section 4.2.3. Secondly, all models will be different sizes, meaning the model viewer will need to calculate the dimensions of each model before it is displayed in the web browser, to ensure that each model is displayed correctly when loaded.

Thirdly, and perhaps most importantly, the application must clearly display the room specified by the user, as well as a route between the specified entrance and the door to the room. Because of its O(n) time complexity [25], and its ease of implementation, the A* search algorithm will be used to perform this calculation. The A* algorithm, like many other search algorithms, works on graphs, in the mathematical sense, i.e. a set of vertices with edges connecting them. A 2-dimensional grid can be considered a graph, with each cell representing a vertex, and edges (grid lines) being drawn between cells that are adjacent to each other [25].

Taking this approach, the entire model would need to be created on top of a 2D grid; certain edges of the grid will represent the walls of the buildings, with the vertices (the grid cells) representing the floor space. This method would allow for the rooms and corridors of the buildings to be separated with ease; for example, the floor area of each room would be defined by a collection of adjacent cells within the grid, and the surrounding edges of the outermost cells will form the walls of the room. The corridors of the building would also be formed of a collection of adjacent cells. Entrances to the building will be composed of 2 cells, with one cell overlapping a corridor cell (inside the building), and the other being placed outside the building. Doors to rooms will be implemented in a similar fashion, with one cell overlapping a cell belonging to the room (inside the room), and the other overlapping a corridor cell (inside the corridor).

Using this method, every entrance and doorway will be connected to a corridor inside the model. This will allow the A* algorithm to use the specified entrance as a starting position, and the door to the specified room as the target to reach. The corridor cells will be used as vertices available to the algorithm, as a possible pathway. Figure 4.2.1 below visualises how this method will be implemented. It's also clear to visualise where the walls of the building will be located, i.e. the entire perimeter of the model, as well as the edges between the room and the corridor cells.

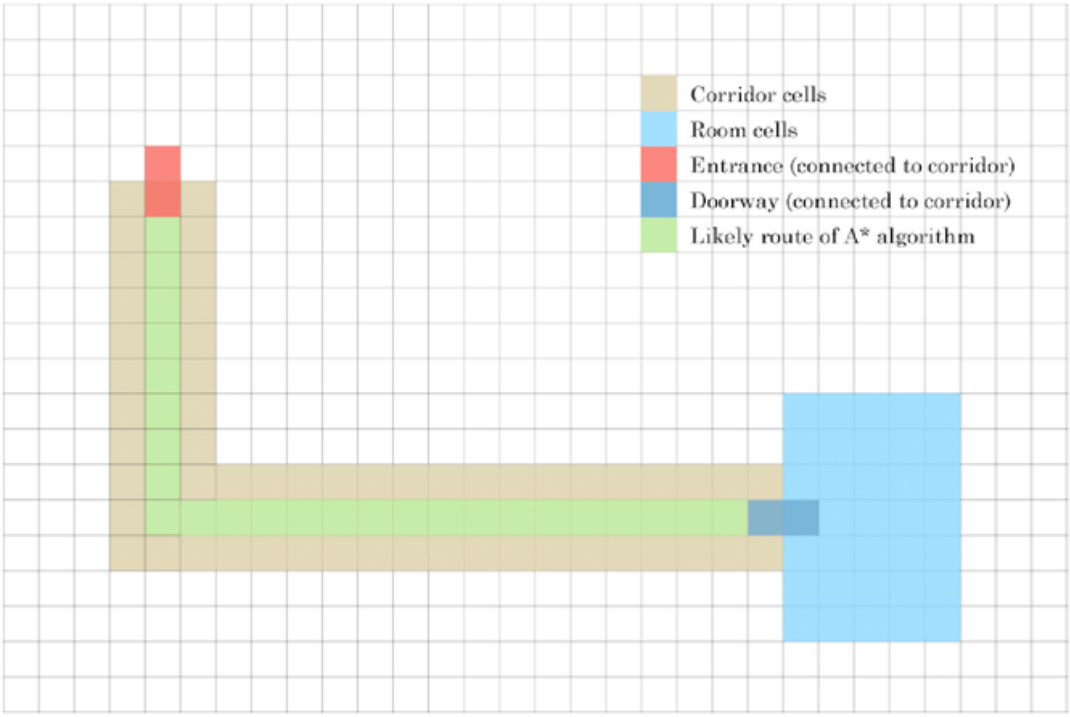


Figure 4.2.1 - The 2D grid solution for A search algorithm*

When the user searches for a room, the model viewer will highlight all the grid cells belonging to that room, as well as the corridor cells that belong to the shortest path between the entrance and doorway, calculated by the A* algorithm. In the event of there being more than one possible pathway, i.e. when a room has more than one doorway, the A* algorithm will calculate all the potential pathways, and display the shortest pathway on the model. This will be explained in greater detail in section 5.2 of this report. There are many A* algorithm JavaScript libraries that currently exist that can be modified for this project, to save the developer time.

4.2.2 The modelling suite

Since it is now understood how the models will be displayed by the model viewer, it's easier to define a solution for the modelling suite. The models created in the modelling suite must be based upon a 2D grid, each room and entrance must have a unique identifier, and the entrances and room doorways must be attached to the corridors. The process of how the modelling suite will be used to create the models is outlined below:

- Load the correct floorplan
- Define the buildings corridors
- Define the buildings entrances, with appropriate names
- Define the buildings rooms, with appropriate names and doorways
- Build the walls of the building, with appropriate walls coverings (interior/exterior)
- Export the model for displaying in the model viewer

At this stage, there are two obvious solutions that could be implemented to create a modelling suite, as outlined in section 3.2; the first is to use Blender, a comprehensive modelling tool, and the second is to use the Three.js framework to create a bespoke modelling suite, for the purposes of creating the 3D models used by the model viewer.

Using Blender would allow the modeller to use an array of existing tools and functionalities to build the 3D models, based on the building's floor plans. There are many tutorials available that provide step by step instructions, describing this process [26]. The fundamental issue with this solution is that the modeller will not easily be able to define which parts of the model are associated with the building's rooms, corridors, entrances and doorways. Therefore, when the model is opened in the model viewer, the user will not be able to use the A* algorithm to plan routes. However, an intermediate stage could be introduced to solve this problem, as Blender allows its users to export models in JSON format, meaning they can be imported into a Three.js application [27]. Using this method, the imported models would have to be analysed to determine the different parts of the building, i.e. corridors, rooms etc., and place them into a 2D grid. Unique identifiers could then be given to each entrance and room, before the model is once again exported, to be displayed in the model viewer.

The other solution, creating a bespoke modelling suite using the Three.js framework, would allow the modeller to define exactly which functions and tools would be needed to create the models. The entire suite would be based upon a 2D grid, with the building's floor plan laid beneath, allowing the modeller to align the walls in the floor plan with the grid lines of the 2D grid. Fortunately, as the dimensions of the rooms inside the finalised models do not need to match the floor plan *exactly*, the grid and the floor plan do not need to be exactly aligned. Once roughly aligned, the modeller will use a

click-and-drag type tool to define which cells in the grid overlay the corridors and which overlay the rooms, naming the rooms appropriately. Doors and entrances could also be added with ease. Once the corridors and rooms have been defined, the modeller will be able to build the walls, using the perimeters of the defined cells as a guide. The model could then be exported as a JSON file, and imported directly into the model viewer with ease.

At this stage, it seems the most appropriate choice would be to develop a bespoke modelling suite in Three.js, therefore this is the solution that will be implemented. There are many reasons for this, partly so the developer/modeller won't have to learn the intricacies of Blender's toolkit, as well as the bespoke suite boasting the ability to import its models directly into the model viewer, removing the need for an intermediate stage. As mentioned before, there are several examples of Three.js projects available online, such as the Voxel Painter [28], that can be used by the developer to learn about the framework's coding structure and style.

4.2.3 Exporting/Importing the models

It's clear that the simplest method of importing the models into the model viewer is via a JSON file. When buildings the models in the suite, each cell within the 2D grid will be given a certain type, be it corridor, room, entrance etc. When the model is complete, this information can be exported and stored in a JSON file, using an appropriate structure. This JSON file can then be imported into the model viewer and the entire process can be reversed; the information in the JSON file will be loaded into the appropriate variables in the model viewer, therefore mimicking the exact model that was built using the suite.

4.3 Tools & IDE's

An appropriate set of tools will be required to develop the application. Firstly, a suitable IDE (Integrated Development Environment) will need to be selected. There are a number of IDE's that support JavaScript development and can therefore be used, however the JetBrains WebStorm IDE has been chosen, due to its intelligent coding assistance, powerful debugging tools and familiar UI.

To ensure the project runs as smoothly as possible, a project management tool will be used. Taiga.io has been chosen for this project, as it's free, incredibly simple to use and has a built-in Kanban style dashboard. Taiga.io will allow the developer to keep track of the progress of features when developing the application. It will also allow the developer to maintain a list of bugs and issues that exist within the application, giving each issue a corresponding severity and priority. The use of a project management platform will help the project run with as few problems as possible, as well as ensuring the developer completes the work within the defined timeframe.

4.4 Definition of solution approach

The solution above describes the project being composed of two separate applications, a modelling suite and a model viewer. Both applications will be accessible in the web browser, utilising the WebGL JavaScript API and will be built using the Three.js framework, using the JavaScript coding language. The modelling suite will allow the modeller to create 3D interactive models of buildings, while the model viewer will allow users to view and interact with the said models. JSON will be used to export the models from the suite and import them into the viewer. The most challenging aspect of this project will most likely be creating an application that doesn't currently exist, even in theory. As there are no current applications like the one proposed in this report, the developer will have very limited resources/examples to help guide the project.

When development is complete, the application will be tested against the problem articulation, technical specification and solution approach to ensure it meets the minimum requirements. As a whole, the indoor navigation solution will be deemed acceptable if it's more efficient to use when compared to asking a receptionist, or simply reading a sign.

5. Implementation

As stated in section 4.2, the modelling suite will be developer prior to the model viewer. Before development can begin, the developer must learn the fundamentals of the Three.js framework. There are several base variables that should be present in every Three.js application, as displayed in Table 5.1 below.

Variable	Purpose	Type used
Scene	The scene is used to store objects /lights/cameras etc. for rendering	N/A
Renderer	Renders the objects present on the scene using WebGL	WebGLRenderer
Camera	Used to project the rendered objects to the web browser/users screen	PerspectiveCamera
Controls	Allows the user to rotate, zoom and pan the camera around the scene	OrbitControls
Lights	Used to add brightness and shadows to the objects present on the scene	DirectionalLight
Raycaster	Allows the mouse/cursor to interact with the objects on the scene	N/A

Table 5.1 - Fundamental Three.js variables used in the application

The above variables will be used by both the modelling suite and the model viewer, to allow both the modeller and the user to build, view and interact with the 3D models.

5.1 Developing the modelling suite

5.1.1 The foundation layers

The foundation of the modelling suite will be composed of 3 flat layers; a plain white backdrop, the image of the floor plan and a 2D grid, as shown in Figure 5.1.2 below. Note that the spacing added between the layers in Figure 5.1.2, as well as the grey tint of the backdrop, have been added purely to highlight the separation of the layers. There are 2 attributes associated with the grid that are very important, as shown in Table 5.2. The values stored in these variables are used by many different functions throughout the application.

Variable	Purpose	Value
Size	Used to set the width and height of the grid. The (0,0) coordinate lies directly in the centre of the grid, meaning the x and z axes* range from -150 – +150	150
Step	Used to set the width and height of each cell present in the grid	2

Table 5.2 – Two global variables used throughout the application

* When modelling in 3 dimensions, 3 axes are used; x, y and z. Their relationship with one another is shown in Figure 5.1.1. In some 3D engines, the z axis represents the depth/'up and down' axis, however, in Three.js, the y axis represents this axis. The z axis represents the horizontal axis, with x representing the vertical axis. The apparent ambiguity of the axes is due to the rotation of the layers shown in Figure 5.2.1.

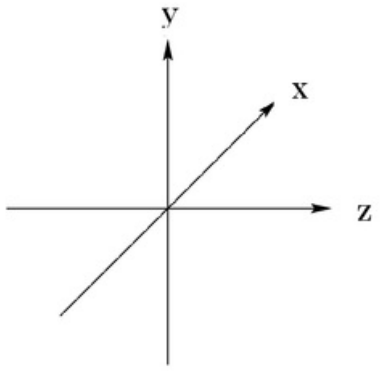


Figure 5.1.1 – Axis representation in Three.js

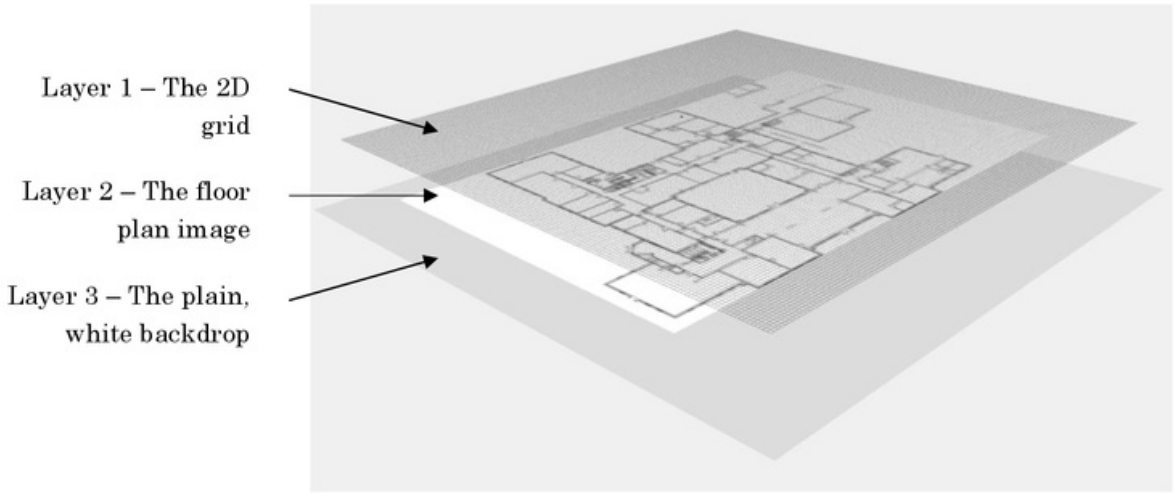


Figure 5.1.2 - The 3 exaggerated layers of the suite foundation

The background layer (layer 3), will always be 0.2x greater than the size of the grid, or more specifically, 0.2x greater than the value in the 'size' variable. This ensures that the backdrop always stretches beyond the perimeter of the grid, making for a more visually pleasing foundation to build models upon.

At this stage of development, the exact floorplan image used as a foundation is irrelevant; what's important is that the image is scaled to fit inside the grid. The application automatically resizes every floor plan when it's loaded, to ensure it will fit perfectly underneath the grid. This is to save the modeller from having to resize every floor plan manually, before loading it into the application.

Firstly, a texture loader is used to load the image into the application. The image is then analysed to find which dimension, width or height, is larger, from which a scale factor is calculated. The image's height and width are then multiplied by the scale factor, fitting the image perfectly within the grid above. The code snippet below shows this process.

```
if (texture.image.height > texture.image.width) {
    scaleFactorHeight = texture.image.height / (size * 2);
    planeImage.scale.z = texture.image.height / scaleFactorHeight;
    planeImage.scale.x = texture.image.width / scaleFactorHeight;
} else {
    scaleFactorWidth = texture.image.width / (size * 2);
    planeImage.scale.z = texture.image.height / scaleFactorWidth;
    planeImage.scale.x = texture.image.width / scaleFactorWidth;
}
```

Figure 5.1.3 shows the relationship between all 3 layers, once they have been compressed, finalised and added to the scene. Notice how the grid and the floorplan layer now exist on the same plane and how the floor plan image has been scaled to fit within the grid.

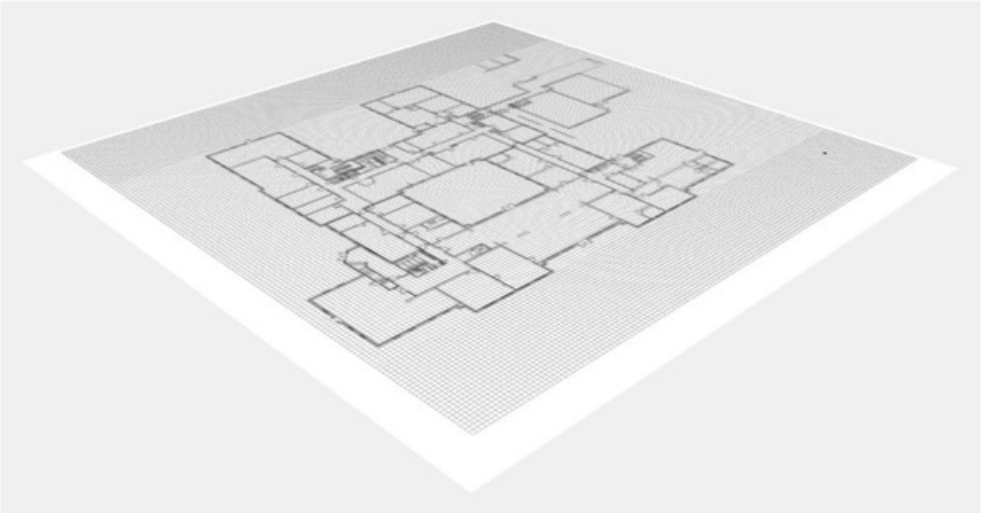


Figure 5.1.3 – The relationship between each of the 3 layers

While the grid lines do not need to overlap the walls of the floorplan exactly, it's important that they are as closely matched as possible. A GUI (Graphical User Interface) has been implemented using the 'dat.GUI' JavaScript library [30] to allow the modeller to scale and position the image of the floorplan, as seen in Figure 5.1.4. The buttons allow the user to increase the size of the image on each axis, as well as move the image along each of the axes, to align the walls with the grid lines. The GUI will also be used at a later stage to allow the modeller to save and export the models.

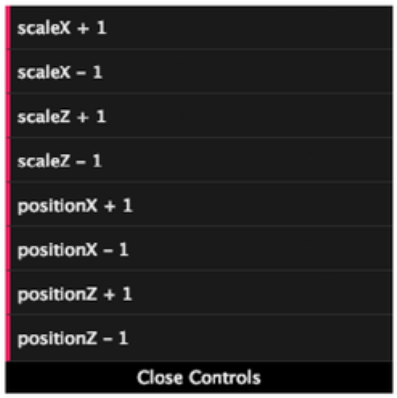


Figure 5.1.4 - The GUI allowing the modeller to position and scale the floorplan image

5.1.2 Meshes, Geometries & Materials

A 3-dimensional object in Three.js is known as a mesh; a mesh is composed of a geometry and a material. Unsurprising, as the objects appear in 3D, a geometry is composed of three dimensions, a height, a width and a depth. An object's material is the colour/texture that is displayed on each face, be it a solid block colour, an image, or a texture etc.

Within the application, the 'materials.js' class is used to define the materials displayed by each object/mesh. The materials defined in the class can be seen below in Table 5.3. There are two important types of material, temporary and permanent. Temporary materials are used when 'rolling over' the cells in the grid, i.e. using the cursor to decide where objects should be placed. Permanent materials are used when the modeller has finalised the placement of an object. This process is defined in greater detail in later sections.

Material	Purpose	Colour/image
corridorColour	The permanent grid cells defined as a corridor	Light brown (0xDCBFAB)
floorColourTemp	The temporary grid cells inside a room (used by the modelling suite)	Light blue (0x6A766A)
floorColour	The permanent grid cells inside a room (used by the model viewer)	Beige (0x89D8FF)
entranceColour	The temporary cell colour used to display where an entrance will be located	Red (0xFE6762)

Material	Purpose	Colour/image
entranceMag	The image used to display an entrance from the interior of the building	Image of an entrance on a magnolia background
entranceBrick	The image used to display an entrance from the exterior of the building	Image of an entrance on a brick background
doorColour	The temporary cell colour used to display where a door will be located	Blue (0x6EADCC)
door	The image used to display a door between a room and a corridor	Image of a door on a magnolia background
brickColour	The permanent colour of an exterior wall	Red/brown (0x8C3D30)
magnoliaColour	The permanent colour of an interior wall	Magnolia (0xF9F6E5)
tempMaterial	The temporary grid cell colour of a corridor/floor/entrance/door grid cell	Grey (0x455455)
routeMaterial	The colour used show the route from point a to point b (used by model viewer)	Green (0x66B266)

Table 5.3 – The main materials used by the application

In the earlier versions of the application, the material used by the exterior walls of the model was defined as an image of brickwork, with the interior walls using a magnolia wallpaper image. These materials were depreciated in later versions of the application to increase the performance. This is outlined in more detail in section 5.1.4.

5.1.3 Build modes & main application functionality

To allow the modeller to create a model with ease and precision, a set of tools have been developed. Each tool is used at a specific stage of the modelling process and is defined in the code as a 'build mode'. Originally there were 5 build modes; 'Corridor', 'Floor, 'Entrance', 'Irregular Wall' and 'Regular Wall'. Their uses are self-explanatory, however they will each be explained in their respective sections below. The 'Irregular Wall' and 'Regular Wall' build modes were later depreciated, as described in 5.1.9. A button was added to the GUI, allowing the modeller to toggle the build mode with ease.

Throughout the application, two crucial global even handlers are used, onMouseMove and onMouseDown. As the name suggests, the onMouseMove event handler defines what happens when the modeller moves the cursor around the application, using the Raycaster variable defined in Table 5.2. The Raycaster will only ever interact with the intersections (the corners) of the grid cells. The onMouseMove method has two primary uses; firstly, it allows the modeller to 'roll over' the grid cells, to decide where objects should be placed. Secondly, it allows the modeller to use the 'click-and-drag' function to specify the width and height of a floor or corridor. Figure 5.1.5 shows the 'roll over' function; notice how the cell highlighted is below the cursor. Figure 5.1.6 shows the 'click-and-drag' function, activated when the modeller has clicked the mouse once, allowing them to specify the size of the floor/corridor. Both functions use the 'tempMaterial' as defined in Table 5.3.

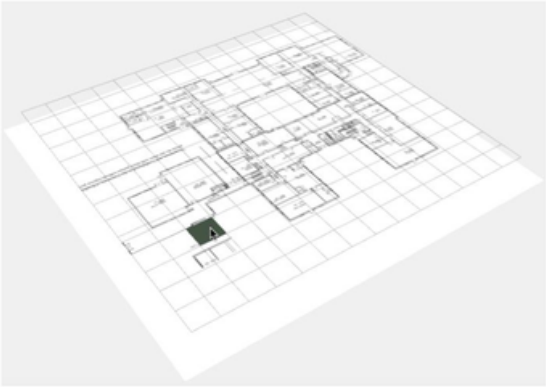


Figure 5.1.5 - The 'roll over' function

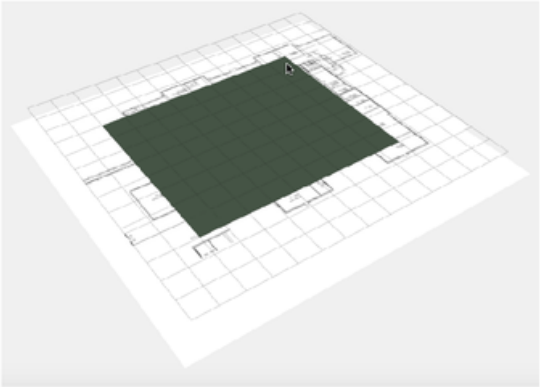


Figure 5.1.6 - The 'click-and-drag' function

The onMouseMove event handler defines what happens when the modeller clicks the mouse. Again, there are two primary functions of this handler; the first click stores the coordinate of the 'roll over' grid cell highlighted by the modeller, as shown in Figure 5.1.5. The second click stores the coordinates of each cell within the highlighted section of the 'click-and-drag' function, shown in grey in Figure 5.1.6, and places the temporary object onto the scene. When the 'click-and-drag' function is not used, for example, when placing an entrance or a doorway, the event handler simply places the temporary object onto the scene and stores its coordinates. This entire process is explained in greater detail in the sections following.

Once a temporary object has been added to the scene, the modeller can either add more (e.g. when creating each individual section of the corridor), or convert them into permanent objects using a key code. Pressing the 'b' key (keycode_B) will convert everything currently on the scene, associated with that build mode, from a temporary object into a permanent object, therefore finalising the placement of that object. For example, when defining the building's corridors using the 'click-and-drag' method, when

the modeller has finished placing each individual section of corridor, they will click the 'b' key, at which point each section of temporary corridor on the scene will be converted into permanent corridor. The grid coordinates for each cell will be transfer from a temporary array into a permanent array, for later use. The temporary array will also be emptied. Table 5.4 displays the different key codes that can be used by the modeller when creating a model. Note that some of the features were not implemented in early versions.

Key	Function name	Purpose
b	keycode_B()	Converts temporary objects in the current build mode into permanent objects
r	keycode_R()	Allows the modeller to rotate an entrance or doorway before being placed
w	keycode_W()	Automatically builds the of walls of the model (see section 5.1.9)
cmd	N/A	Used with the onMouseDown event handler to allow the modeller to rotate the model
cmd/ctrl+z	keycode_CMD_Z()	Allows the modeller to undo the placement of temporary objects
backspace	keycode_BACKSPACE()	Allows the modeller to cancel the 'click-and-drag' function

Table 5.4 – The key codes used throughout the application

5.1.4 Building the walls – version 1

The order in which models will be built has been specified in section 4.2.2, however the walls of the model can be can be built at either the beginning or the end of construction. Initially, it was thought it may have been easier to visualise the model if the walls were built prior to the corridors and room. This section outlines how walls were built manually in the earlier versions of the application.

As rooms within a building can either be rectangular or non-rectangular, each 'section' of wall in the application will span the width of one grid cell. This will allow the modeller to create both 'regularly' and 'irregularly' shaped rooms in the application, therefore matching the shape of the rooms in the floorplan. Each 'section' of wall can be positioned either vertically or horizontally; if positioned vertically, it's *depth* (along the x axis) will be the size of the grid cell (as defined by the 'step' variable in Table 5.2 above), with its width being set slightly wider than the width of the grid line. If positioned

horizontally, it's *width* will be the size of the grid cell, with its depth being slightly larger than the width of the grid line. The wall's height will always be constant, whether vertical or horizontal, as defined in the global variable 'wallHeight'. The code below shows this:

```
xWallGeometry = new THREE.BoxBufferGeometry(step, wallHeight, 0.5);
zWallGeometry = new THREE.BoxBufferGeometry(0.5, wallHeight, step);
```

Regular walls belong to rooms that are rectangular; that is, they have 4 sides/walls where every angle is a right angle, as depicted in Figure 5.1.7. Irregular walls belong to rooms that are non-rectangular, as seen in Figure 5.1.8.

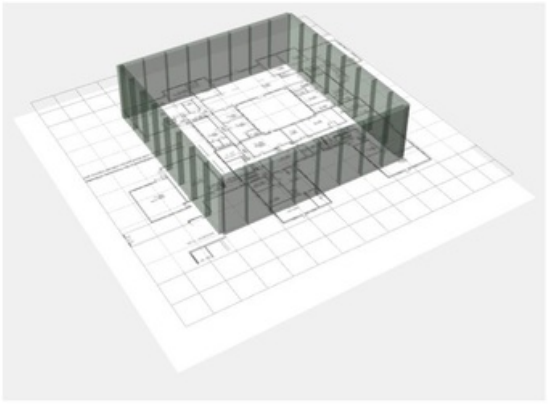


Figure 5.1.7 - Regular wall temporary placement

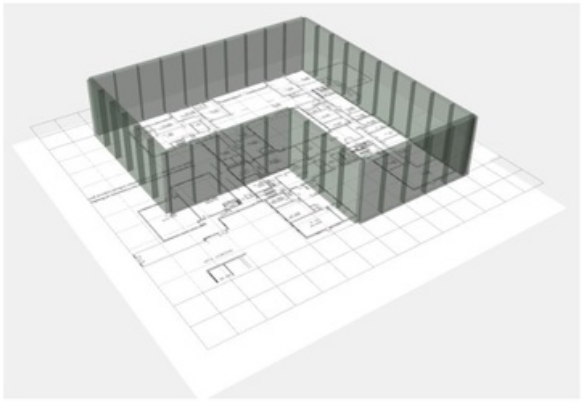


Figure 5.1.8 - Irregular wall temporary placement

To place a regular wall, the modeller would select the 'Regular Wall' build mode from the GUI. They would then click the corner of the grid cell overlaying the corner of the room or corridor in the floorplan below (Start-coordinate (S)). They would then move the mouse/cursor to the corner of the grid cell overlaying the *opposite* corner of the room/corridor of the floorplan below (End-coordinate (E)). The application would then perform a simple calculation to determine the 2 unknown coordinates, for example:

Known coordinate	Known value	Unknown coordinate	Calculated value
Start-coordinate (S)	x = 4	Unknown x	x = S(x) = 4
	z = 6		z = E(z) = 10
End-coordinate (E)	x = 12	Unknown z	x = S(z) = 6
	z = 10		z = E(x) = 12

Table 5.5 - Unknown coordinate calculations

When each of the 4 coordinates is known, the application would place several 'sections' of wall between each of the 4 coordinates, as seen in Figure 5.1.7. When the modeller clicked the mouse for a second time, the walls would be converted from temporary objects, into permanent objects, as seen in Figure 5.1.9, and thus a regular room is built.

To place an irregular wall, the modeller would select the 'Irregular Wall' build mode from the GUI. The modeller would firstly click the corner of the grid cell overlaying the corner of the room or corridor in the floorplan below. They would then click the corner of another grid cell, at which point the application will place a temporary wall along the axis between the two cells. The modeller would continue this process, changing the direction of the walls with each click, until they return to the starting coordinate, as seen in Figure 5.1.8. At this point, the application will finish the placement of the walls, converting the temporary objects to permanent objects, as seen in Figure 5.1.10.

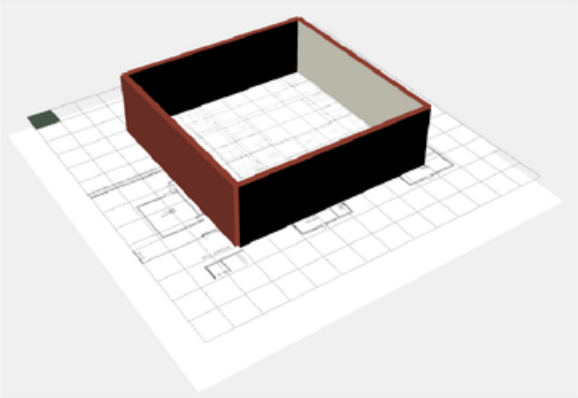


Figure 5.1.9 - Regular wall permanent placement

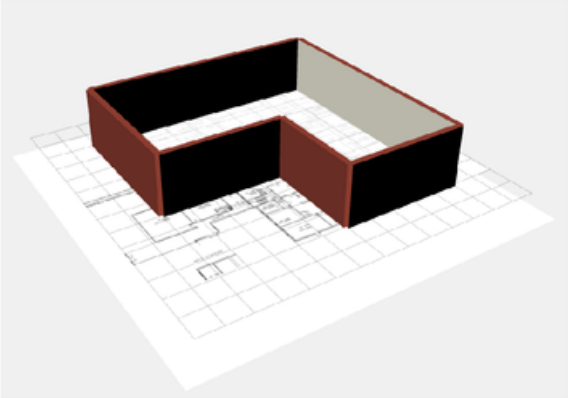


Figure 5.1.10 - Irregular wall permanent placement

There was a small bug that was found in this part of the application (before it was depreciated), that would have effected many areas of the modelling suite. Originally, the Raycaster would scan ever object on the scene, which included the 2D grid and each of the currently placed walls. It was noticed that the walls on the scene physically blocked the Raycaster/mouse cursor from intersecting the grid lines located directly behind them, meaning the modeller could not place certain walls. To fix this, the 2D grid was duplicated and placed in a separate array, away from the other objects on the scene, such as previously placed walls. Now, the Raycaster only interacts with objects in this array (i.e. only the 2D grid) when the modeller moves the cursor. This allows the Raycaster to 'see through' walls and other objects on the scene.

The models shown in Figures 5.1.9 and 5.1.10 are displaying different materials for the interior and exterior walls. Originally, 4 different materials were used to produce this effect. When converting a wall from temporary to permanent, the application would determine the walls relationship with the x and z axes. For example, if the wall was located along the x axis, and its location on the z axis was greater than 0 (i.e. it's in the top left quadrant of the grid, as (0,0) is the centre point of the grid), then the application would give the face of the wall facing the centre of the grid a magnolia material, and the opposite side a brick material. This method was sufficient for the earlier versions of the application, however it assumed that the centre of the grid, (0,0), resided *inside* every

room, which is obviously incorrect. An optimised version was developed and implemented in later versions. This has been described in further detail in section 5.1.9.

To visualise to progress currently made, the first model was built with the modelling suite at this stage of development. Building the first model highlighted many areas of the modelling suite that needed vast improvement, however this was expected, as development was still incomplete. The feedback from the first use of the modelling suite, along with images of the first model, can be seen in section 6.2.1.

5.1.5 Creating the rooms – version 1

Since the application now allows for the modeller to create rooms (multiple connected walls with no gaps), it should theoretically allow for the rooms to be utilised. Each room within the model must have a name and a set of grid coordinates that determines its location within the model. When building a regularly shaped room, the room coordinates are easy to calculate as the 4 corners of the room can be used. However, problems arise when trying to calculate the grid coordinates of irregularly shaped rooms; it was proposed that the room should be split into rectangles with the corners of each individual rectangle being used to determine the area of the entire of the room.

It was soon realised that developing an algorithm to accomplish this would be very complex and time consuming, and it couldn't be guaranteed to work every time. At this stage, the 'click-and-drag' function described in section 5.1.3 had not been developed. It was soon understood that the most important part of the models were the grid coordinates of the corridors, rooms and entrances, and that the walls are *purely* for visual purposes. The rooms therefore needed to be based on the grid cells, not the locations of the walls, i.e. the walls could be removed from the model, but the room would still exist. Knowing this, future development work focused on the fully utilising the grid to build the models and paid less attention to the construction of the walls.

5.1.6 Placing the corridors

As it is now understood how the modelling suite would be used to create models, as well as the order of the building process, work could begin on developing the corridor build mode. The 'click-and-drag' function specified in 5.1.3, and referenced throughout this report, was developed at this stage of development. It successfully allowed the modeller to select which cells in the 2D grid belonged to the corridors with ease, using the 'click-and-drag' function and the 'keycode_B' function to confirm the cell selection. The coordinates of the cells are placed into the 'corridorArray' for later use, with the application ensuring no cells are duplicated in the array. Figure 5.1.11 shows the cells belonging to the corridors of the model in light brown, using the 'corridorColour' material specified in Table 5.3.

5.1.7 Placing the entrances

The next step in development was to allow the modeller to add entrances to the building. Entrances are composed of 2 grid cells; one cell connected to/overlapping a corridor cell, inside the building, and other connected/lapping an empty cell, outside the building. The modeller can use the 'r' key (keycode_R) to rotate the entrance allowing for them to be placed on either axis. When the modeller places an entrance, the application asks them to specify its name. The application then calculates which of the two entrance cells lies within the corridor and this cell is marked with a Boolean value. An entrance object is then created, containing the name of the entrance, the positions of its two cells, and which cell is connected to the corridor. The modeller can repeat this process, adding every entrance to the building to the model, before using 'keycode_B' to confirm the locations of each entrance. The entrances are added to the 'entranceArray' for later use. Figure 5.1.11 shows the entrances in the model in red, using the 'entranceColour' material specified in Table 5.3.

5.1.8 Creating the rooms – version 2

At this point in development the word 'rooms' was replaced with the word 'floors'; the word 'rooms' suggests the need for a floor, walls and a ceiling, however the word 'floors' implies that the only important aspect of a room is its floor space. It also meant that different types of rooms (e.g. offices, toilets etc.) could be created, without there being confusion between them and their floor space (as the floor space is the only real important aspect). However, in this report, the words 'room' and 'floor' will be assumed to mean the same thing.

Using the 'click-and-drag' function, floors could also now be created in the same way as corridors; the modeller selects the appropriate cells, uses 'keycode_B' to confirm their selection, and enters a suitable name for the floor space (the name of the room). The application then allows the modeller to add a doorway to the room, in the same way that entrances are added to the model. The modeller can add multiple doorways to a room, ensuring that one cell of the doorway lies within the room and the other within the corridor. Again, the application determines which cell of the doorway is connected to the corridor and marks it with a Boolean value. The modeller then uses the 'keycode_B' function once again to complete the room creation. At this stage, a room object is created, containing the name of the room, the coordinates of the floor area, and the positions of its doorways. The object is added to the 'roomArray' for later use. Sometimes a room will not be directly attached to a corridor, but attached instead to another room (i.e. a room inside a room); in this scenario, the positions of the doorway stored against the room object will be inherited from the room it is attached to. This can be visualised in Figure 5.1.11; each room is coloured light blue, with doorways being coloured slightly darker.

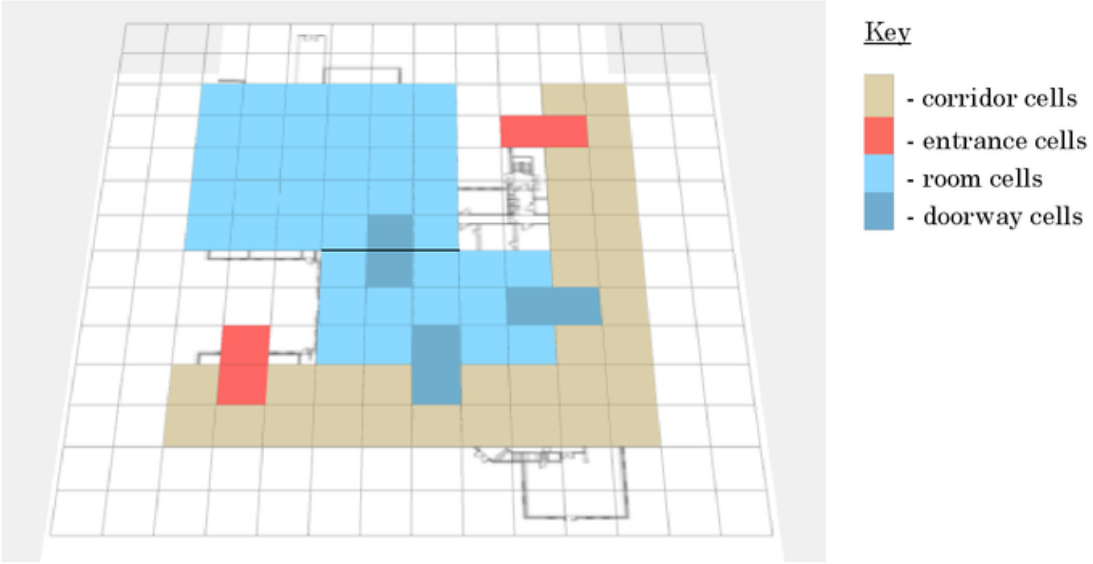


Figure 5.1.11 - The modelling suite displaying the different types of cells

Figure 5.1.11 shows how the different types of cell are displayed in the modelling suite, before the walls have been built. Note that the objects (rooms, corridors etc.) displayed are not based upon a floorplan. The Figure shows how entrances and doorways can be of either orientation and how rooms can be attached to other rooms (the black line dividing the rooms has been added to emphasise the division). Figure 6.2.3 in section 6.2.2 shows an entire floor plan being modelled using this technique.

5.1.9 Building the walls – version 2 (Automatic wall algorithm)

At this stage of development, the corridors, entrances, rooms (floors) and doorways in the model have all been defined, however the walls have not. Currently, the only way to create walls is the use the manual function outlined in section 5.1.4. Using this method is slow, unpredictable and frequently selects the incorrect materials for the walls. Automating this process would almost halve the time needed to create a model. It would also, in theory, ensure that the correct material is applied to every wall object. When studying Figure 5.1.11 it's easy to see where the walls should be placed, and which materials should be applied, however the application cannot interpret this for several reasons. Firstly, all the objects on the scene (corridors, floors etc.) are segregated, as they exist in separate arrays in code. Secondly, the grid is technically 3 dimensional, as each grid cell coordinate has 3 values, x, z and y. Thirdly, the origin (the (0,0) coordinate) of the grid is in the centre, as opposed to the corner of the bottom left quadrant. These factors make it difficult to use the 3D grid alone to determine the locations of the walls.

To solve these problems, a 2-dimensional, code based grid is used in conjunction with the 3-dimensional grid displayed on the screen. When the application is initialised, a 2D array is created to mimic a 2D grid. The 'size' variable is used to ensure that both the 2D and 3D grids are of equal size. The 2D array is stored in the 'grid2D' global variable, meaning it can be used throughout the application. Every value in the 2D array is initialised to 0.

```
var grid2D = [];
var gridSize = (size-1)*2;

for(var y = 0; y < gridSize; y++)
{
   grid2D.push([]);
   for(var x = 0; x < gridSize; x++)
   {
      grid2D[y].push(0);
   }
}
```

The key to this solution is to store each object (corridor, room, entrance etc.) in both the 3D grid and the 2D array. The first obstacle is to determine *how* the objects are going to be represented in the 2D array. It was decided that each object type would be given a unique identifier in the form of a number; these numbers could then be positioned in the 2D array in the same locations as the 3D grid, using the coordinates as a guide. This is explained in greater detail later. Table 5.6 indicates which number corresponds to which object type.

Unique ID	Object type
0	Ground/outdoor cells
1	Corridor cells
2	Floor cells
3	Door cells
4	Entrance cells

Table 5.6 - Unique ID's for each object type

The second obstacle was to calculate the offset between the coordinates in the 3D grid and the 2D array, as the (0,0) coordinate of the 3D grid is in the centre, whereas the (0,0) coordinate of the 2D array is the first element (the corner of the bottom left quadrant). Therefore, the application must calculate the offset between the two coordinates and use it to determine the correct locations of the objects when placing them in the 2D array. The following for loop was developed to carry out this calculation.

```
for (var i = 0; i < positionArray.length; i++) {

   xCoord = positionArray[i].x;
   yCoord = positionArray[i].z;

   if (xCoord > 0)
      xCoordGrid = ((size - 1) - xCoord) / 2;
   else if (xCoord == 0)
      xCoordGrid = size / 2;
   else if (xCoord < 0)
      xCoordGrid = ((size - 1) + (xCoord * -1)) / 2;


   if (yCoord > 0)
      yCoordGrid = ((size - 1) - yCoord) / 2;
   else if (yCoord == 0)
      yCoordGrid = size / 2;
   else if (yCoord < 0)
      yCoordGrid = ((size - 1) + (yCoord * -1)) / 2;

   grid2D[xCoordGrid][yCoordGrid] = positionArray[i].mode;
}
```

The 'positionArray' array contas the coordinates of all the cells belonging to a certain object (e.g. all the corridor cells) in the 3D grid. Each x and z coordinate is converted by the application using the size of the grid as a base value. The unique identifier corresponding to the object type (the 'mode') is then placed into the 2D array, using the newly calculated coordinates.

For example, if the corridor cell coordinate (-22, 14) was analysed in a 50x50 grid ((0,0) is the centre of the 3D grid and that cells are '2' wide (the value of the 'step' variable), meaning each axis ranges from -25–+25), the following calculation would occur:

```
xCoord = -22
yCoord = 14
size = 25
positionArray[i].mode = corridor = 1

if(xCoord < 0)
        xCoordGrid = ((25 - 1) + (-22 *-1)) / 2
        xCoordGrid = (24 + 22) / 2
        xCoordGrid = 23

if(yCoord > 0)
        yCoordGrid = ((25 - 1) - 14)) / 2
        yCoordGrid = (24 - 14) / 2
        yCoordGrid = 5

grid2D[23][5] = 1
```

The above example therefore states that the 3D coordinate (-22, 14) is equal to the 2D coordinate (23, 5). The entire process outlined above is repeated every time the modeller uses the 'keycode_B' function, i.e. every time they finalise an object placement. Figure 5.1.12 shows the relationship between the objects in the 3D grid and the 2D array.

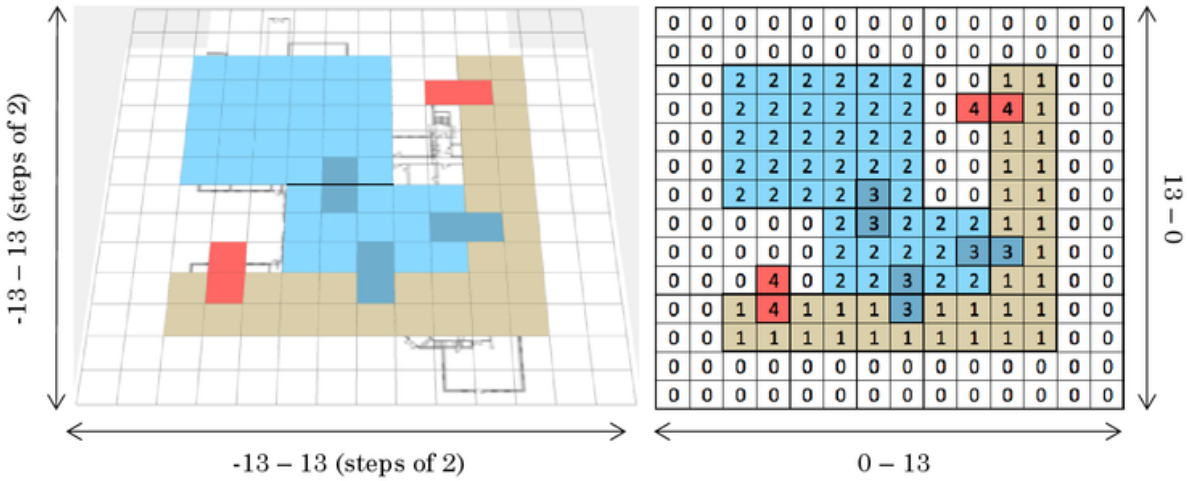


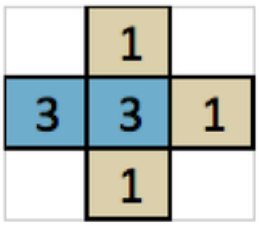
Figure 5.1.12 - Left: The 3D grid Right: The 2D array

There are two obvious errors with the values in the 2D array displayed in Figure 5.1.12. Firstly, it's clear that the model contains 2 separate rooms, however the values in the 2D array do not specify the border between the 2. Instead of being given the unique identifier '2', each room must have its own unique identifier. To solve this, each room object is given a number when it's placed into the 'roomArray'. This number is incremented every time a room object is created. Now, when placed into the 2D array, the room number will be appended onto the unique identifier, thus giving each room its own unique ID. Secondly, the 2D array does not know which corridor cell is outside the building and which is inside. To solve this, the Boolean value that was added to the entrance upon creation (specifying which cell overlaps the corridor) is evaluated. The value '0' is appended to any entrance value that lies within a corridor, and a '1' is appended to those that don't. Figure 5.1.13 shows the finalised 2D array values for the 3D grid displayed in Figure 5.1.12.

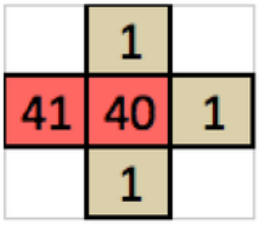
Once finalised, the 2D array can be used to automatically create the walls of the model, using a bespoke algorithm. The algorithm analyses each value of the 2D array in turn and determines which, if any walls need to be built around it, before proceeding to analyse the next cell. This entire process is repeated until ever cell has been analysed. The algorithm compares the value of the current cell with the values of its neighbouring cells. From this it can determine which wall should be built and which material should be applied. Below are 3 examples.

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	22	22	22	22	22	22	0	0	1	1	0	0
0	0	22	22	22	22	22	22	0	41	40	1	0	0
0	0	22	22	22	22	22	22	0	0	1	1	0	0
0	0	22	22	22	22	22	22	0	0	1	1	0	0
0	0	22	22	22	22	3	22	0	0	1	1	0	0
0	0	0	0	0	21	3	21	21	21	1	1	0	0
0	0	0	0	0	21	21	21	21	3	3	1	0	0
0	0	0	41	0	21	21	3	21	21	1	1	0	0
0	0	1	40	1	1	1	3	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

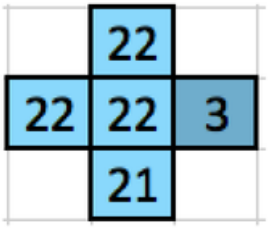
Figure 5.1.13 - The finalised 2D array values



Current:	3	N/A
Right:	1	No wall placed between a door and corridor cell
Left:	3	Wall placed between two door cells on the x axis Material = 'door' on both sides
Above:	1	No wall placed between a door and corridor cell
Below:	1	No wall placed between a door and corridor cell



Current:	40	N/A
Right:	1	No wall placed between entrance and corridor cells
Left:	41	Wall placed between two entrance cells on the x axis Interior = 'entranceMag', Exterior = 'entranceBrick'
Above:	1	No wall placed between entrance and corridor cells
Below:	1	No wall placed between entrance and corridor cells



Current:	22	N/A
Right:	3	No wall placed between a floor and door cell
Left:	22	No wall placed between two cells with same value
Above:	22	No wall placed between two cells with same value
Below:	21	Wall placed between two different floor cells on the z axis Material = 'magnoliaColour' on both sides

Once every cell within the 2D array has been analysed, the walls of the 3D model are erected on top of the 3D grid. This entire process occurs when the modeller presses the 'w' key, and takes less than a second to execute. Figure 5.1.14 shows the result of the algorithm automatically build the walls for the model. Notice how every material has been applied correctly. At this stage of development, a second model was created using the modelling suite; section 6.2.2 provides details of this.

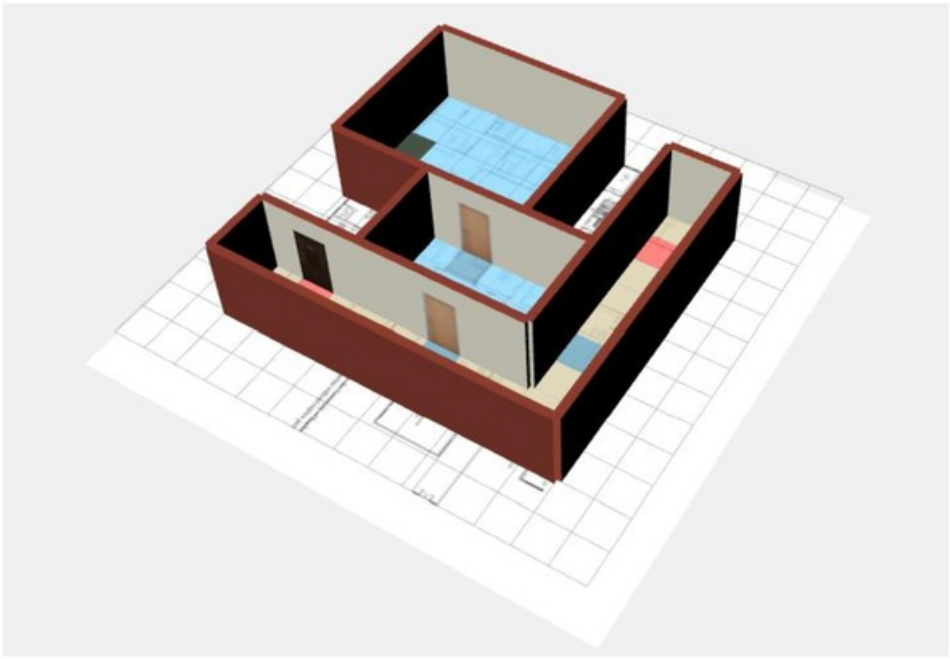


Figure 5.1.14 - The 'automatic walls' algorithm creating the walls for the model

5.1.10 Exporting the models

The final challenge was to allow the modeller to export the models once they had been created. Within the modelling suite, the objects, i.e. entrances, rooms, and corridors, are stored in separate arrays, with the walls stored in the scene array. It was deemed appropriate that the contents of these arrays should be converted to JSON and loaded into the model viewer, as a way of saving and loading the models.

A button was added to the GUI that allows the modeller to save the model. When they click the button, the application converts the contents inside each object array, as well as the walls within the scene array, into JSON objects, using the built in '.toJSON' function. These JSON objects are then placed into an encompassing array, which is again converted to a JSON string, using the 'stringify' function. The application then asks the modeller to enter a name for a model. Once the name has been entered, the application creates a new JavaScript Blob file, renames the file and automatically downloads it to the

modellers machine, in the JSON format. The file contains every individual object within the model, including the names of the objects and their coordinates within the grid. Each object is then grouped in an array with other objects of that type. The basic structure of the JSON file is depicted below:

```
modelname.json[
        [wallArray]
        [roomArray]
        [entranceArray]
        [corridorArray]
]
```

This entire process will be reversed when a user loads a model within the model viewer. The objects within each array of the JSON file will be placed into the appropriate arrays in the code, as well as the walls being placed onto the scene.

5.2 Developing the model viewer

5.2.1 Import the models

To allow users to view models in the model viewer, they must first be imported. Section 5.1.10 described how the models are exported from the modelling suite, in the JSON format. The foundation of the model viewer is very similar to that of the modelling suite, however the floor plan and grid layers are not used by the viewer; the models are simply displayed on top of a white backdrop layer.

Firstly, the loading function parses the JSON file that has been selected by the user, and stores the response in an array. Each element of the array contains a group of certain objects, as depicted below (e.g. the wallArray contains all the wall objects):

```
modelname.json[
        [wallArray]
        [roomArray]
        [entranceArray]
        [corridorArray]]
```

The first major challenge is to calculate the grid transformation needed to position the model, as well as to determine the size of the white backdrop layer. The coordinates of the cells stored in the JSON file reflect where the model was positioned in the modelling suite. These coordinates will have to be modified to ensure the model fits perfectly in the centre of the model viewer. Figure 5.2.1 displays an example transformation (notice again that each grid coordinate is incremented by 2, not 1, as this is the value stored in the 'step' variable).

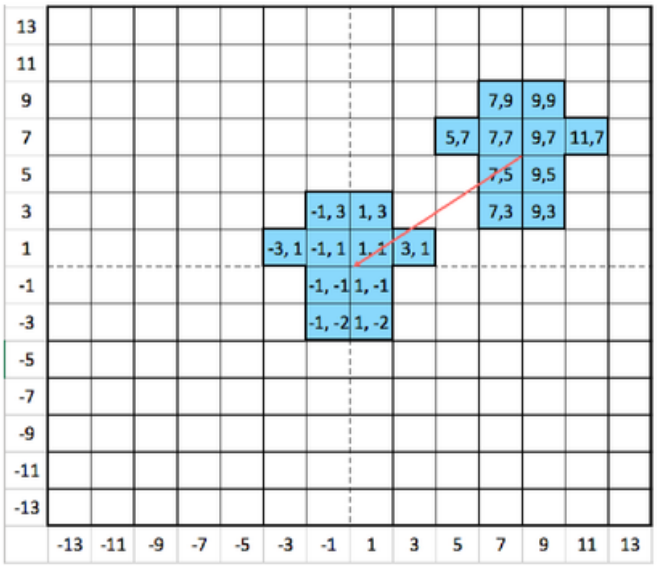


Figure 5.2.1 - An example grid transformation

The algorithm must be able to calculate the grid transformation for every model that is built with the modelling suite, by using the models size and cell coordinates as a guide. Frist of all, the size of the white backdrop layer is calculated. The layer's height and width must be slightly larger than that of the model's, and will be different for each model. To do this, the objects present in the 'roomArray' and the 'corridorArray' are used. The algorithm beings by looping through each object in both arrays and storing the z and x coordinates of every cell in two new arrays, named 'zCoords' and 'xCoords' respectively. For example, if there were 3 rooms in the 'roomArray', the x and z coordinates for each cell belonging to each room would be stored in the new arrays. Eventually, the two arrays will contain the positions of every room and corridor cell used to create the model in the modelling suite (with no duplicates). Using the example in Figure 5.2.1, the 'zCoords' and 'xCoords' arrays would as such (notice that neither array contains duplicate values):

xCoords = ['3', '5', '7', '9']

zCoords = ['5', '7', '9', '11']

The algorithm then determines the size/length of each array (how many items are in each), both of which is 4 in this scenario (variables are named 'maxX' and 'maxZ'). Each value is then multiplied by 10% leaving the two dimensions of the backdrop layer. In this scenario, the backdrop layer will be:

maxX = 4

maxZ = 4

layer.x = (maxX+(maxX/100)) = (4+(4/100)) = **4.4**

layer.z = (maxZ+(maxZ/100)) = (4+(4/100)) = **4.4**

Figure 5.2.1 clearly shows that the model will fit inside a 4x4 grid, therefore this calculation is correct. Next, the algorithm must determine the grid transformation. To do so, the highest values present in each array are stored in appropriate variables. The size of each array is also used again, at a later stage:

highX = 9

highZ = 11

These values are then used to calculate the translator value needed to convert the original coordinate values into the transformed values. For both the x and z values, the algorithm determines which value is greater, the size of the array, or the highest value present in the array. This allows the algorithm to determine whether the translator value will be a negative or positive value. The difference between the two values is then calculated, which is the final the translator value needed for the transformation, for that axis. For example:

diffX = ((highX - maxX) * -1) - 1 = 9-4 = **-6**

diffZ = ((highZ - maxZ) * -1) - 1 = 11-4 = **-8**

In the above example, the values are negated as the values of 'highX' and 'highZ' are greater than 'maxX' and 'maxZ'. Each value is also decremented by 1 to ensure the multipliers are kept even. This example states that each x coordinate must be decremented by 8 and each z coordinate must be decremented by 6, to translate the model correctly. Applying this calculation to the model present in Figure 5.2.1 proves the success of this algorithm. The values of 'diffX' and 'diffZ' are used in the next stage of loading the model. The code snippet for this algorithm can be seen below.

```
var highX = Math.max.apply(null, xCoords);
var highZ = Math.max.apply(null, zCoords);

var maxX = xCoords.length;
var maxZ = zCoords.length;

var diffX;
var diffZ;


if (highX > lowX)
   diffX = Math.abs(highX - maxX) + 1;
else
   diffX = (Math.abs(highX - maxX) * -1) - 1;

if (maxZ > highZ)
   diffZ = Math.abs(highZ - maxZ) + 1;
else
   diffZ = (Math.abs(highZ - maxZ) * -1) - 1;
```

Once the size of the backdrop layer has been determined and the translator values have been calculated, the application can begin to display the model. To do this, the application stores each object array present in the JSON file in an appropriately named array in the code (the names of the arrays used by the model viewer are the same as the names used by the modelling suite). At this stage, every object type present in the model (corridors, rooms, walls etc.) exists in its own array. The application then loops through each of these arrays and modifies the coordinates for each individual object, adding the translator value to each x and z value. For example, the application would start by looping through the 'wallArray'. The value -6 would be added to the x coordinate of each wall object, and -8 would be added to every z coordinate value. The process is then repeated for each array of objects.

Once the process is complete, the white backdrop layer will be visible in the very centre of the scene, with the entire model being displayed above. Figure 5.2.2 shows the example model being displayed by the model viewer.

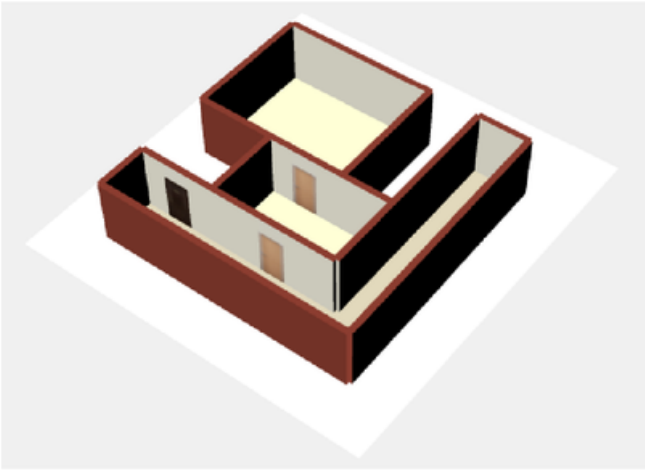


Figure 5.2.2 - The example model being displayed by the model viewer

5.2.2 Interacting with the models

The model viewer used the same controls and camera as the modelling suite, allowing the user to rotate, pan and zoom in/out of the model. As the models are primarily built to allow users to search for rooms within a building, and display routing information, a GUI has been designed and implemented, as shown in Figure 5.2.3. The GUI is split into two parts, a route planner and a room finder; the route planner allows the user to specify the name of an entrance and the name of a room within the model. Clicking the search button will then display the route between the specified room and entrance. The user can also specify the entrance name 'any', in which case the application will determine which entrance is closest to the specified room, and plot a route between the two. The room finder

is simply used to highlight a specific room inside a model (this function is also used by the route planner). Both functions are explained in greater detail in sections 5.2.4 and 5.2.3 respectively. The entire GUI is collapsible, ensuring it does not block the view of the model.

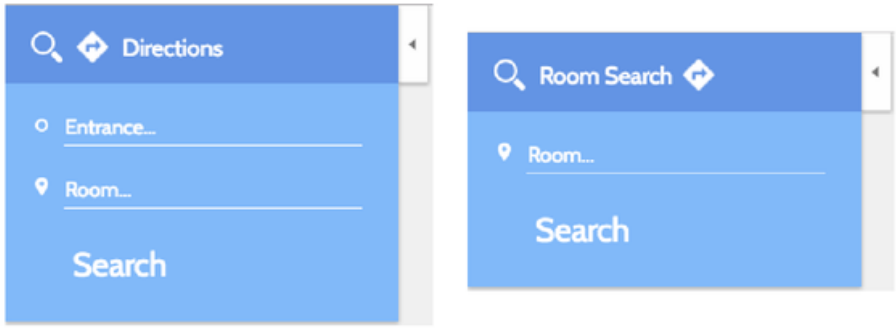


Figure 5.2.3 - Left: GUI used for route plotting Right: GUI used for room highlighting

5.2.3 Room finding

As mentioned in section 5.2.2, the user can use the model viewer to search for rooms within a model, using the room finder section of the GUI. To do this, the user enters the name of a specific room into the search box and click the search button. The application will then search the 'roomArray' (populated by the JSON file), for a room object matching the name of the specified room. If a room is found, the application loops through each coordinate belonging to that room. For each coordinate, the application creates a new mesh using the dimensions of a grid cell ('floorGeometry') and the 'routeMaterial' material (green – see Table 5.3). This mesh is then positioned slightly above the original coordinate, using the cells x and z coordinates, and a y coordinate of 0.1. The mesh is then added to the scene. This process continues until a mesh has been created for each coordinate belonging to the room.

```
for(var i = 0; i < room.positions.length; i++){
   var planeRoom = new THREE.Mesh(floorGeometry, routeMaterial);

   planeRoom.position.x = room.positions[i].x;
   planeRoom.position.z = room.positions[i].z;
   planeRoom.position.y = 0.1;

   scene.add(planeRoom);
}
```

Figure 5.2.4 shows room number 2 being highlighted in the example model. Notice that as the grid lines are not displayed, it does not appear as though grid cells/coordinates are being used.

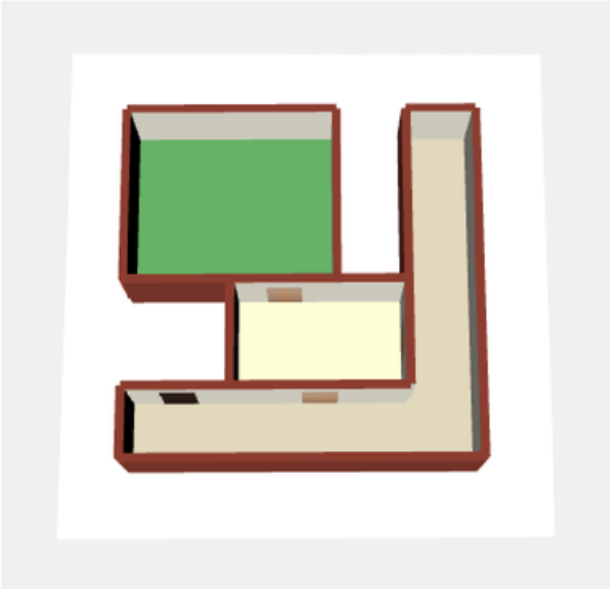


Figure 5.2.4 - The room highlighting function being displayed by the example model

5.2.4 Route planning - A* search algorithm

The sole purpose of the application is to find the shortest route between entrances to a building and the rooms inside that building. The model viewer uses the A* search algorithm to satisfy this objective; section 4.2.1 of this report specified why the A* algorithm was chosen. Firstly, the user must specify the name of an entrance and the name/number of a room inside the model, the application will then search the 'entranceArray' and the 'roomArray' for the values specified by the user. If an entrance is found, the application finds the coordinates of the entrance cell that resides inside the corridors of model (by evaluating the Boolean value). The coordinates are placed in a variable and returned to the main function for later use. If the user specifies 'any' entrance, the corridor cell coordinate for each entrance is added to an array and that array is returned to the main function. The application then searches the 'roomArray' for the coordinates of the doorway(s) belonging to the specified room. These coordinates are again returned to the main function.

At this stage, the application knows the coordinates of the entrance(s) to the building and the doorway(s) to the specified room. All the coordinates are guaranteed to reside *within* the corridors of the building. Figure 5.2.5 displays another example model, while Figure 5.2.6 displaying the 2D array/grid coordinates that represents the model. For this demonstration, the input values will be as follows:

Entrance = 'any'

Room = 1

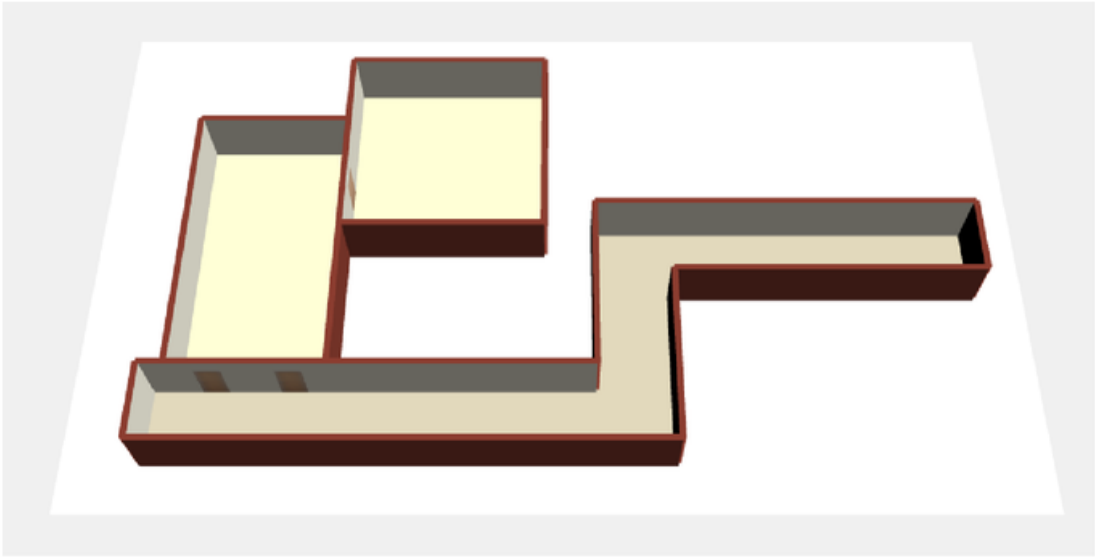


Figure 5.2.5 - The example model used to demonstrate the A algorithm*

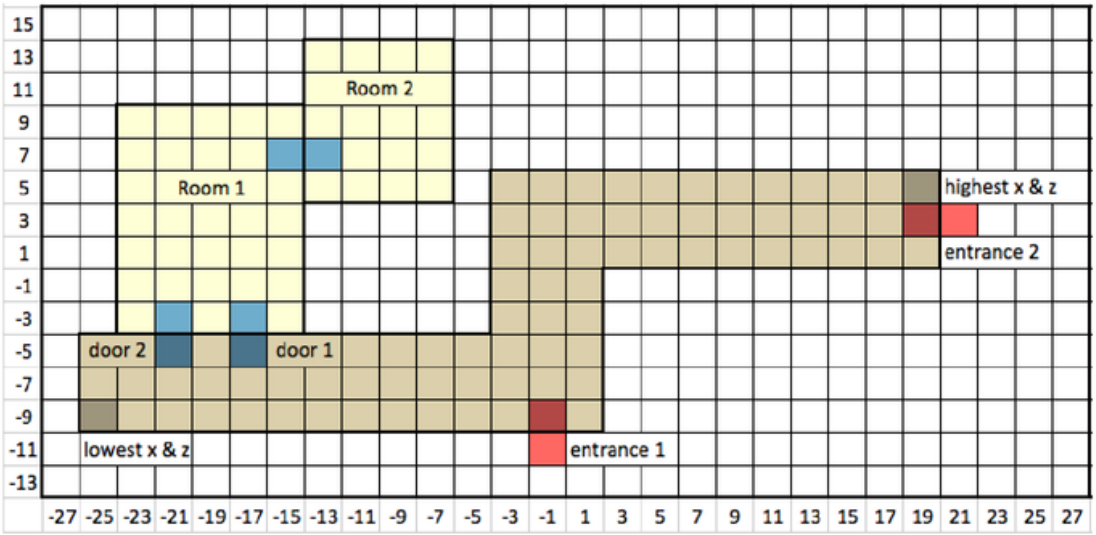


Figure 5.2.6 - The grid coordinates representing the example model

As the user has specified 'any' entrance, the application will create an array containing the 2 sets of entrance coordinates. Also, as room 1 has two doorways, the application will create another array containing the 2 sets of doorway coordinates (the coordinates belong to the cells attached to the corridor):

```
entrancePositions = ['(-1, -9)', '(19, 3)']
roomPositions = ['(-21, -5)', '(-17, -5)']
```

As previously mentioned, an external A* JavaScript library is used by the application to perform the shortest path calculation [31]. However, this library expects a start coordinate, an end coordinate and a grid as input parameters. The input grid must contain a set of coordinates that the algorithm is allowed to visit, and a set of cells that are unavailable to visit. The grid must also contain the start and end coordinates. The application must create this grid in code, using a 2D array, and pass it into the A* library. The application begins by looping through each of the corridor coordinates to determine the highest and lowest x and z coordinates. In this example they would be as follows:

lowX = -9

highX = 5

lowZ = -25

highZ = 19

Figure 5.2.6 shows how these values were determined; notice the 'lowest x & z' and 'highest x & z' labels. These values will be used by the algorithm later. The algorithm then creates 4 new arrays, used to separate and store the x and z coordinates of the entrances and rooms previously calculated ('entrancePositions' and 'roomPositions'). As a 2D grid is now being used, the 'z' coordinates are renamed as 'y' coordinates. These arrays are used later in the algorithm:

entranceX = ['-9', '3']

entranceY = ['-1', '19']

roomX = ['-5', '-5']

roomY = ['-21', '-17']

The algorithm must then create a 2D grid/array in code, containing the starting coordinates, ending coordinates, a set of coordinates that can be visited and a set that cannot. To do this, a for loop is nested inside another for loop, with the 'lowX', 'lowZ', 'highX' and 'highZ' values previously calculated being used as loop conditions. This ensures that the size of the 2D grid is the same height and width of the area surrounding the corridor coordinates, as depicted in Figure 5.2.7. In this scenario, the 2D grid created by the for loop will be 23x8.

```
for (var i = highX; i >= lowX; i -= step) {
    grid[countZ] = [];
    countX = 0;
    for (var j = highZ; j >= lowZ; j -= step) {
        grid[countZ][countX] = [];

        //main algorithm

        countX++;
    }
    countZ++;
}
```

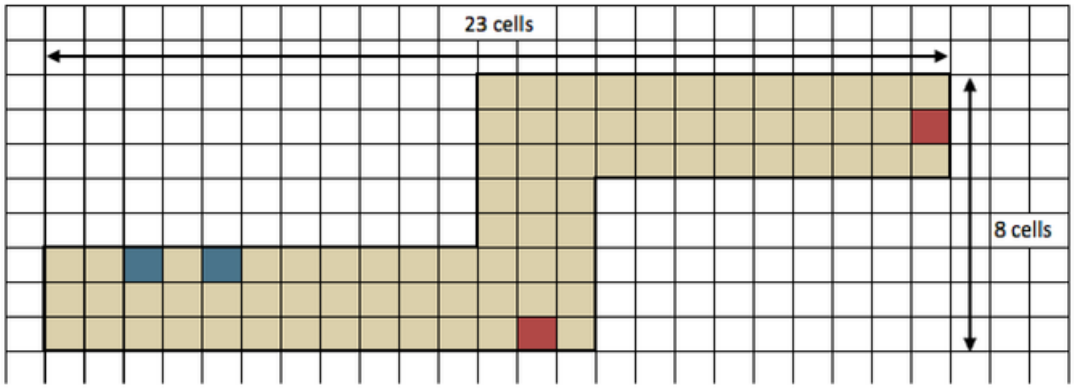


Figure 5.2.7 - How the size of the 2D grid is determined

The 'countX' and 'countZ' variables are used to track which newly created 2D grid coordinates corresponds with the original 3D grid coordinate. This is needed as the 3D grid coordinates can be negative or positive, and are incremented in steps of 2, whereas the 2D grid coordinates are always positive, begginning at 0, and increment in steps of 1.

The most important parts of the algorithm reside within the nested for loop. At the beginning of every cycle, the algorithm determines whether the current cell within the 2D grid corresponds to a corridor cell or any other cell within the 3D grid. If it corresponds as a corridor cell, the value of the 2D grid cell is set to 1, if not it is set to 0. The code snippet below is taken from within the nested for loop. The variables 'i' and 'j' are the loop conditions:

```
for (var k = 0; k < corridorCells.length; k++) {
    match = 0;
    if (corridorCells[k].x === i && corridorCells[k].z === j) {
        match = 1;
        break;
    }
}

grid[countZ][countX] = match;
```

Once the algorithm has determined whether a cell corresponds to a corridor, it must then determine whether the cell is also used by an entrance or doorway. Firstly, it checks whether the 'roomX' array contains the current X loop count value (i), as well as whether the 'roomZ' array contains the current Z loop count value (j). If both values are present it then determines whether the values both belong to the same coordinate, originally belonging to the 3D grid. If they do, the 'countZ' and 'countX' values are stored in two separate arrays; these values represent the position of a doorway within the 2D grid. This process is repeated for each of the doorways and entrances, until each 3D grid value has a corresponding 2D grid value:

```
if ((roomX.indexOf(i) > -1) && (roomY.indexOf(j) > -1) &&
((roomY[roomX.indexOf(i)] == j) || (roomX[roomY.indexOf(j)] == i))) {
    roomXConverted[roomXConverted.length] = countZ;
    roomYConverted[roomYConverted.length] = countX;
}

if ((entranceX.indexOf(i) > -1) && (entranceY.indexOf(j) > -1) && ((entranceY[entranceX.indexOf(i)] == j) ||
(entranceX[entranceY.indexOf(j)] == i))) {
    entranceXConverted[entranceXConverted.length] = countZ;
    entranceYConverted[entranceYConverted.length] = countX;
}
```

Once both for loops have been executed, a 2D grid/array will have been created, containing a set of coordinates that can be visited by the A* algorithm (the corridor cells), and a set if coordinates that cannot be visited (any other cell). Figure 5.2.8 clearly displays the calculated values of the 2D grid/array. Below are the calculated starting (entrances) and ending (doorway) coordinates (highlighted in Figure 5.2.8):

entranceXConverted = ['0', '22']

entranceYConverted = ['12', '6']

roomXConverted = ['2', '2']

roomYConverted = ['2', '4']

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
7	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
4	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0

Figure 5.2.8 - The values of the 2D grid/array

The algorithm now passes the grid, along with the starting and ending coordinates into the A* library. The application again uses a nested for loop, and for each cycle a new entrance/door pair is input into the A* algorithm, along with the entire 2D grid/array. The algorithm uses a binary heap to calculate the route between the entrance and the doorway, exclusively using the values in the grid/array marked as '1' as a potential pathway. The library then returns an array of coordinates (the path), which is stored in a variable labelled 'finalResult', along with the coordinates of the entrance and doorway, before continuing the loop. If the next cycle produces a path of fewer values (a shorter distance), then the original path is over written, along with the entrance and doorway coordinates. This process is repeated for each doorway/entrance pair.

Eventually, the algorithm will have calculated which doorway and entrance are closest, as well as determining a pathway (as a list of coordinates) between the two. For this example, the distances are as follows (note that the pathway cannot be diagonal):

entrance1 –> door1 = 8
entrance2 –> door1 = 19
entrance1 –> door2 = 10
entrance2 –> door2 = 21

The shortest route exists between entrance 1 and door 1, therefore this pathway will be used. The algorithm converts the coordinates present in the 'finalResult' variable back into their corresponding 3D grid coordinates, creates a mesh for each coordinate, and displays the mesh in the 3D model. The entire room is also highlighted. Figure 5.2.9 displays a bird's eye view of this route being displayed in the 3D model.

Figure 5.2.10 displays the route between entrance 2 and the room. Notice that the user never specifies which doorway to the room they wish to use; the application simply calculates which door is closest to the specified entrance. The entire process depicted in the section takes under 1 second to execute.

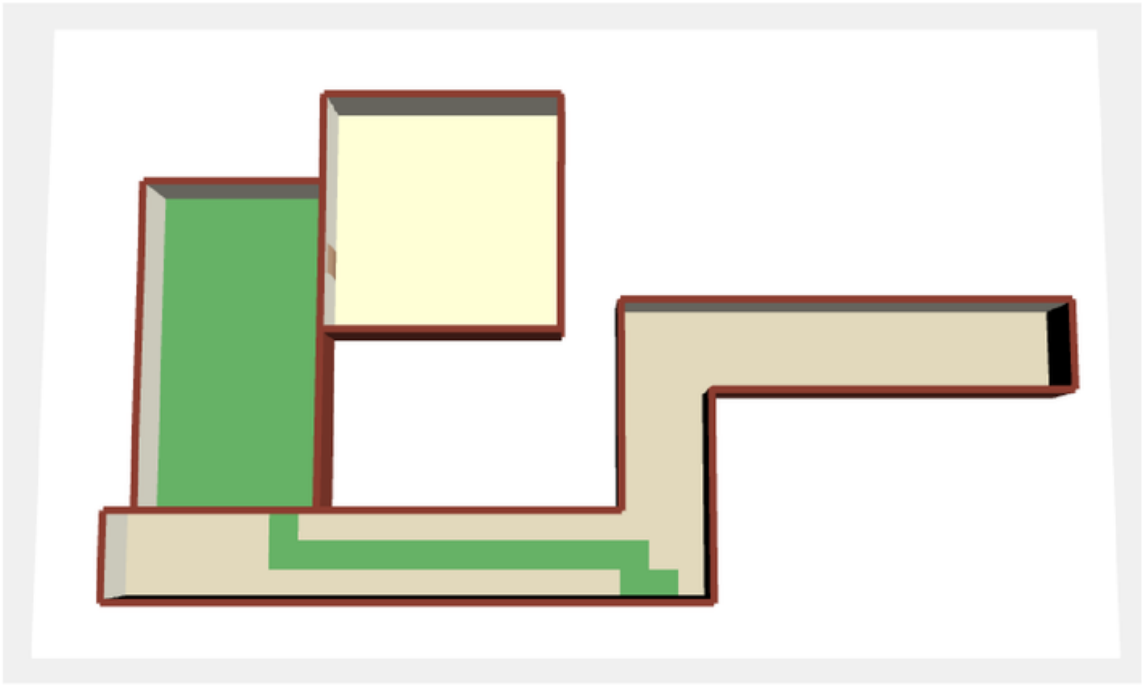


Figure 5.2.9 - Bird's eye view of the route between entrance 1 and room 1

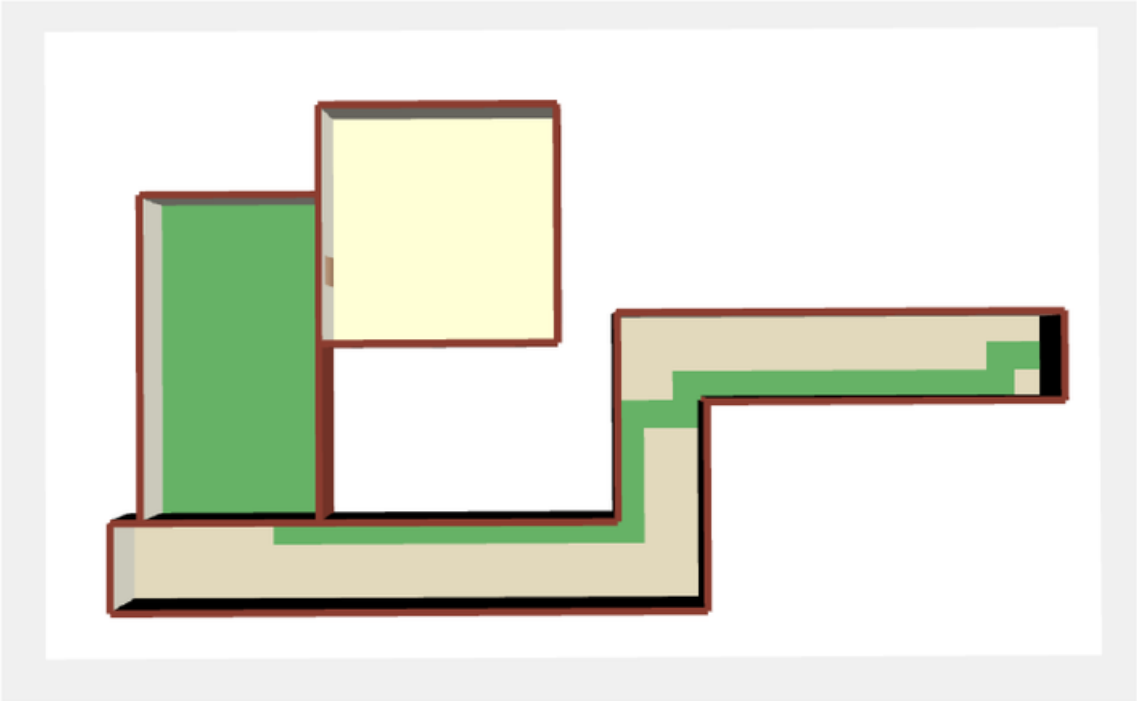


Figure 5.2.10 - Bird's eye view of the route between entrance 2 and room 1

If the user searches for room 2, the application will again show the shortest between entrance 1 and room 1, however room 2 will be highlighted in the model, as shown in Figure 5.2.11.

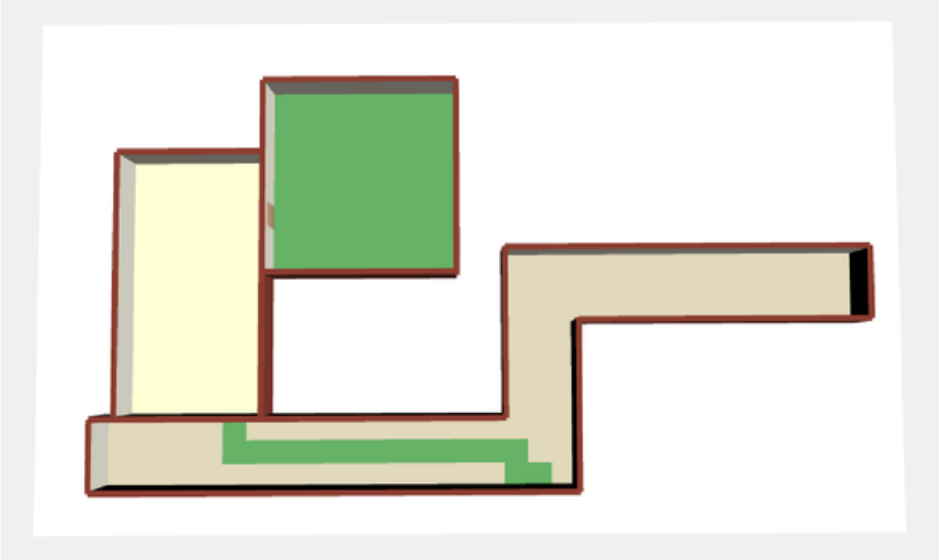


Figure 5.2.11 - Bird's eye view of the route between entrance 1 and room 2

5.3 Hosting & navigating the applications

As the applications are intended to be used online, an appropriate domain name needed to be purchased, which meant the application package needed to be named. As the application allows modellers and users to interact with 3D model buildings, using a floorplan to create the models, the 'Blueprint' was adopted. Both the modelling suite and the model viewer are hosted on the '3dblueprint.uk' domain. The modelling suite is located at the 'suite.3dblueprint.uk' subdomain, whilst the model viewer is located at the 'view.3dblueprint.uk' subdomain.

5.3.1 The modelling suite - suite.3dblueprint.uk

When the user loads the modelling suite, they are presented with a loading screen, as seen in Figure 5.3.1. They are then asked to select a floorplan from a dropdown menu, as seen in Figure 5.3.2. The menu contains a list of floorplan images located in the folder '.../floorplans/'. Once the user has selected a floorplan, the modelling suite is loaded, with the specified floorplan image loaded into the image foundation layer.

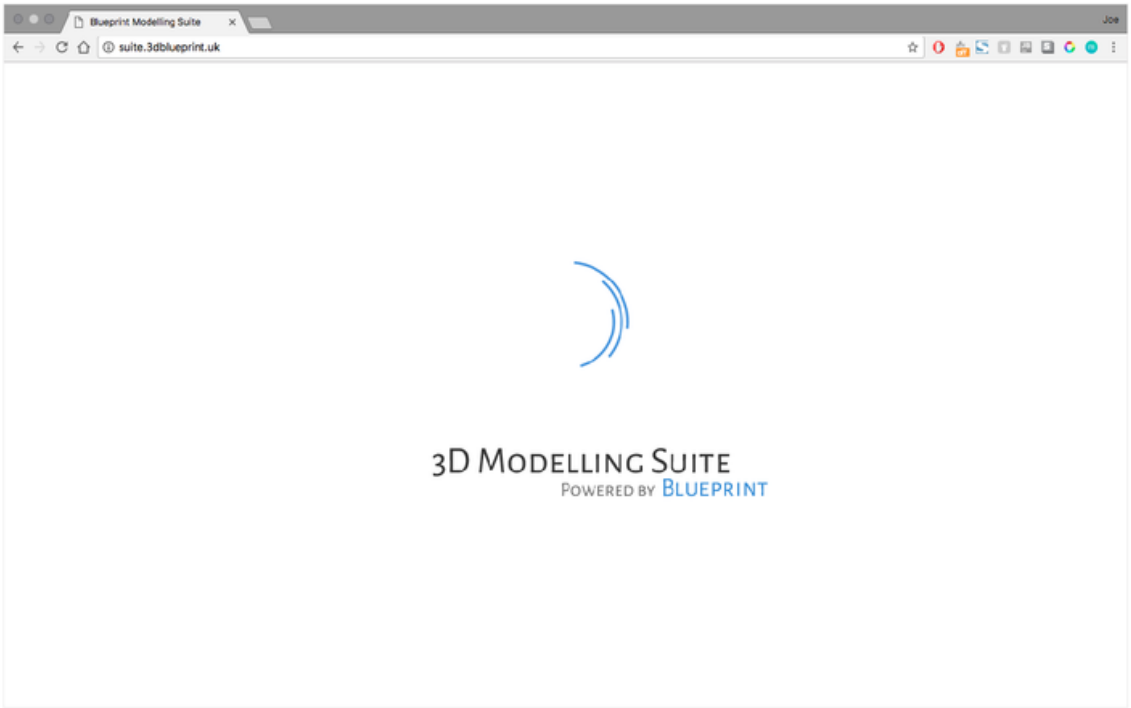


Figure 5.3.1 - The modelling suite loading screen

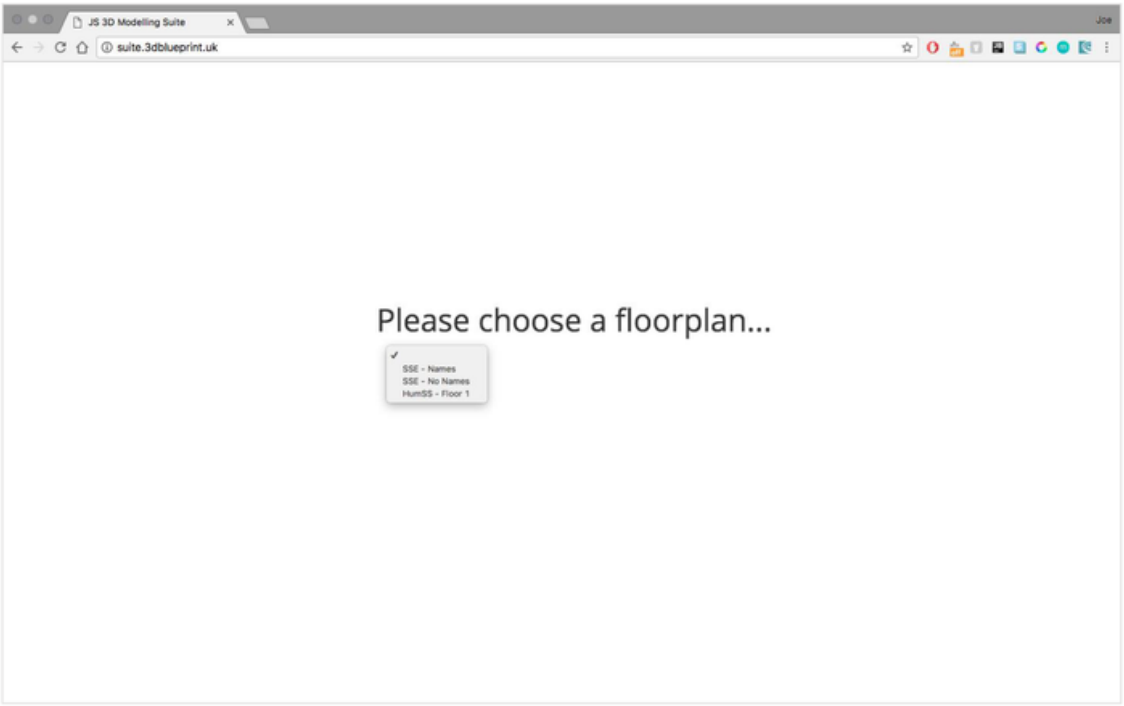


Figure 5.3.2 – The modelling suite drop down menu

5.3.2 The model viewer - view.3dblueprint.uk

When a user loads the model viewer, they are again presented with a loading screen, as seen in Figure 5.3.3. However, they must specify which model they wish to load by entering a parameter into the URL. The query string 'm=' is used to specify which model is to be loaded by the model viewer. For example, if the user wished to load a model named 'sse.json', they would enter the following URL:

view.3dblueprint.uk/?m=sse

The model viewer at this point would take the parameter present in the query string, in this case 'sse', and search the folder '.../models/' for the JSON file with the same name ('sse.json'). The JSON file is then loaded into the model viewer for the user to interact with.

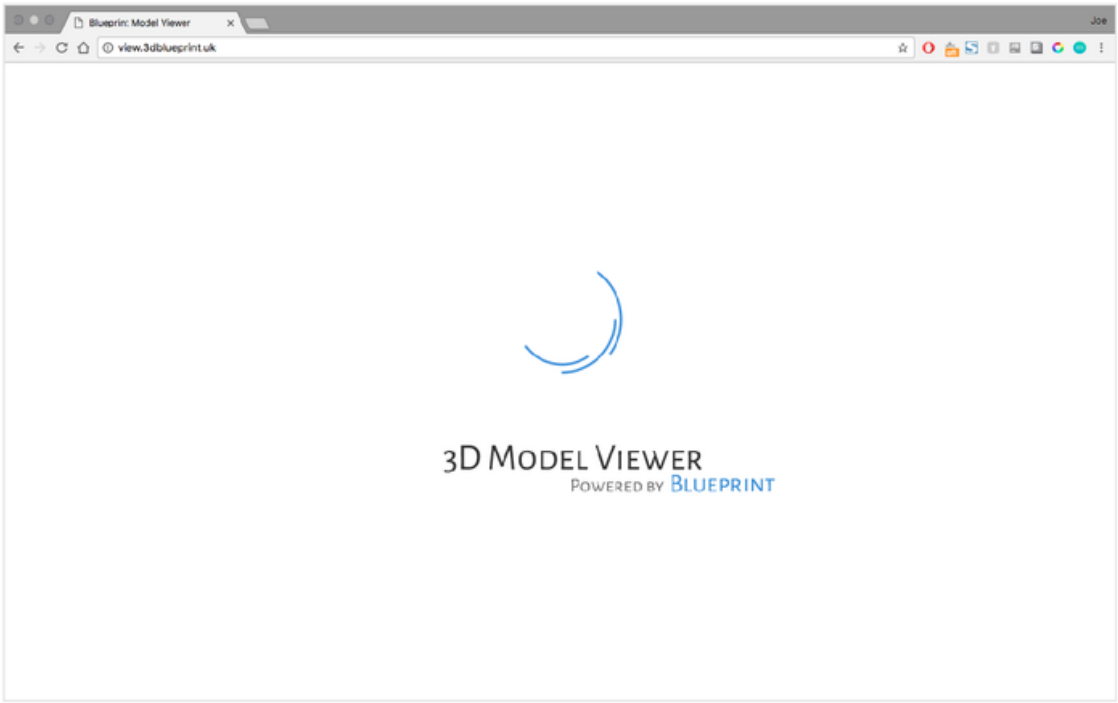


Figure 5.3.3 - The model viewer loading screen

6. Testing: Verification & Validation

As an agile project management methodology was adopted for this project, the applications produced were subject to frequent, rigorous testing. This section of the report outlines the different types of testing that took place at the end of the project. Each of the tests was performed using the macOS operating system, on the Google Chrome web browser, unless stated otherwise.

6.1 Verification & validation

6.1.1 Verification

As previously mentioned, an agile project management technique was adopted for this project, meaning that the solution being created was verified whenever a new development task was completed. This allowed the developer to ensure the solution was being developed *correctly*.

The initial requirements gathered in the PID (see appendix 1) were consulted several times throughout the development of each application. If a piece of proposed development work did not solve a problem outlined by the requirements it was often not implemented into the final application.

Many bugs were encountered during the development of both applications. To ensure that these bugs were fixed prior to the release of the software, they were logged in the project management tool used by the developer. Each bug would be recorded with a severity and priority value, allowing the developer to determine which bugs required the most attention.

6.1.2 Validation

To ensure that the *correct* applications were developed, several tests were conducted, relating to the requirements highlighted in the PID. Both applications were tested on their functionality, compatibility, usability and performance. The results for the tests were analysed, highlighting to the developer which areas of the application needed improving, to ensure the initial requirements were met.

6.2 Functionality testing

To visualise the progression of the modelling suite, models were created at 3 different developmental stages. To gauge the progression, the time taken to create each model was recorded, as well as how easy it was to create, the detail of the model and the performance of the application once complete. Limitations and other remarks were also recorded.

6.2.1 Creating the first model

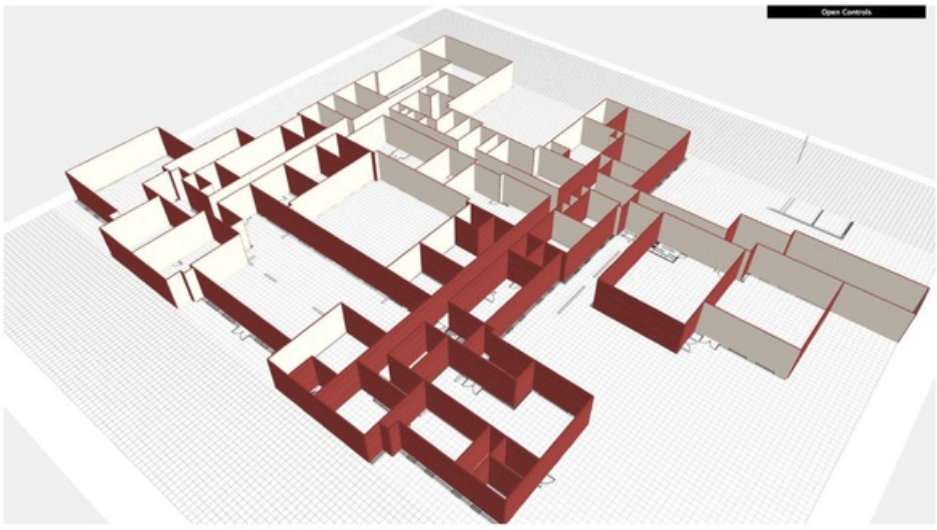


Figure 6.2.1 - The first model created with the modelling suite

Measure	Result	Comments
Time taken	25 minutes, only walls	The model took 25 minutes to build, with only the walls being modelled (no rooms/corridors/entrances)
Ease of build	3/10	It was complicated to build, with a high number of human errors throughout
Model detail	Poor	The model isn't visually pleasing, the walls are too tall and the materials displayed are incorrect
Application performance	Slow, very low FPS	Interacting with the model was slow; it could not be rotated without severe lag - adding rooms/entrances/ corridors would not be possible

Table 6.1 - Measures and results from the first model creation attempt

Overall, the first attempt of creating a model could be regarded as a success. While there were certain, clear limitations of the modelling suite, it allowed the modeller to create a full, interactive model based on an image of a floor plan. The limitations and criticisms of the modelling suite at this stage are as follows:

- Performance is not good enough to allow the modeller to build walls, corridors, rooms and entrances
- The model would benefit from the walls being either smaller or transparent, to allows the user to see though/behind them

- Clicking on GUI or around the model to move the camera is treated by the application as a mouse click and therefore a wall begins to be built
- Adding extra pieces of wall to fill the gaps in rooms is very difficult, possibly have an extra build mode that allows the modeller to add singular walls
- Would be nice to highlight certain grid lines on the x and z axes to visualise where the wall placement should end, i.e. where the final wall should be placed to finish the room creation

6.2.2 *Creating the second model*

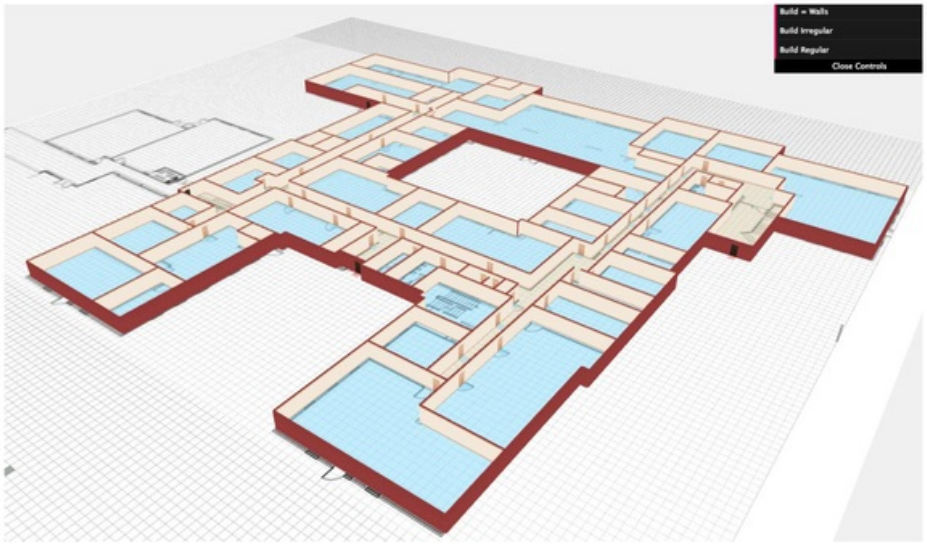


Figure 6.2.2 - The second model created with the modelling suite

Measure	Result	Comments
Time taken	14 minutes	The model took 14 minutes to build, with all aspects of the building (corridors, rooms, walls etc.) being modelled
Ease of build	6/10	The model was relatively easy to build; however, it was very easy to make a mistake
Model detail	Good	All walls are the correct colours, entrances and doors are visible within the model
Application performance	Slow, very low FPS	Interacting with the model is still slow; interacting with the model is very jittery

Table 6.2 - Measures and results from the second model creation attempt

The modelling suite has vastly improved since the first model was created. The second model was far more detailed and a great deal easier to build, however the performance is still subpar:

- The walls look great; however, some sections are built incorrectly. The problem occurs when doors and entrances and built in adjacent cells
- Rooms can be attached to other rooms
- The undo function is not working, therefore mistakes cannot be correct
- Shadows do not appear to be visible

Figure 6.2.3 - The second model before the automatic wall function was executed

Figure 6.2.4 - The second model after the automatic wall function was executed

6.2.3 Creating the third model (after performance improvements)

The third functionality test took place following a series of intricate performance improvements to the modelling suite, as detailed in section 6.3 below.

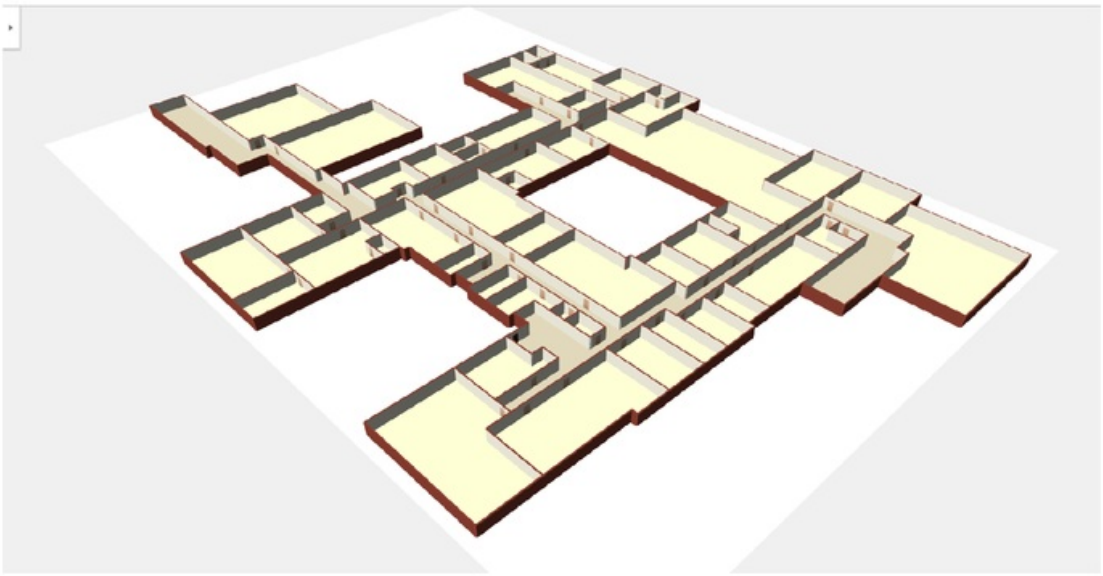


Figure 6.2.5 - The third model created with the modelling suite

Measure	Result	Comments
Time taken	13 minutes	The model took 13 minutes to build, with all aspects of the building (corridors, rooms, walls etc.) being modelled
Ease of build	8/10	The model was very easy to build; the fixed undo function allowed the modeller to retract any mistakes
Model detail	Very good	All walls are the correct colours, entrances and doors are visible, floors have colour and shadows are visible
Application performance	Very fast	Interacting with the model is much faster, it has a much higher FPS rate and can be used without lag

Table 6.3 - Measures and results from the third model creation attempt

The modelling suite in its finalised state was very easy to use and produced a fully functional, detailed model that posed no performance issues. However, some sections of wall are still being built incorrectly.

6.3 Performance testing

Throughout the development, both the modelling suite and the model viewer suffered from serious performance issues, as highlighted by the functionality testing in section 6.2. These performance issues needed to be fixed, for the solution to be deemed acceptable.

6.3.1 Identifying the performance issue

Before any performance improvements could be implemented, it was important to identify which part of the modelling process was causing the issues. The Three.js 'renderstats.js' library was used to visualise and track the number of 3D objects on the scene, and it was found that the performance was significantly reduced when the number of objects increased. This was due to each object having to be re-rendered every time the camera moved (rotating, zooming and panning. Two initial tests were carried out, in an attempt to pinpoint exactly which objects were the cause of the performance issues. For each test iteration, certain attributes were recorded; for each attribute, a lower number is preferred:

- Calls – The number of separate function calls that were made
- Vertices – The total number of object vertices present on the scene
- Faces – The total number of object faces present on the scene
- Geometries – The total number of shape geometries present in memory

The first test recorded the objects on the scene, before the walls had been built, as seen in Figure 6.3.1. The second test then recorded the number of objects on the same scene, after the walls had been built, as seen in Figure 6.3.2. Table 6.4 displays the attribute values recorded for both tests.

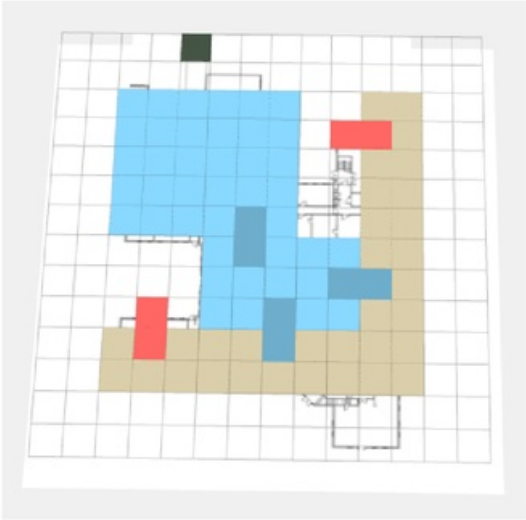


Figure 6.3.1 - Objects on the scene before wall placement

Figure 6.3.2 - Objects on the scene after wall placement

Attribute	Before walls	After walls
Calls	18	450
Vertices	162	2754
Faces	34	898
Geometries	280	286

Table 6.4 - Performance attributes before and after the walls had been built

Table 6.4 clearly shows that the walls of the model are generating the highest number of objects on the scene, thus significantly reducing the performance of the application. 3 stages of performance improvements were documented, to try and reduce the number of objects present on the scene. At each stage, the same model was created using the modelling suite. The model was then exported for testing in model viewer, as seen in Figure 6.3.3.

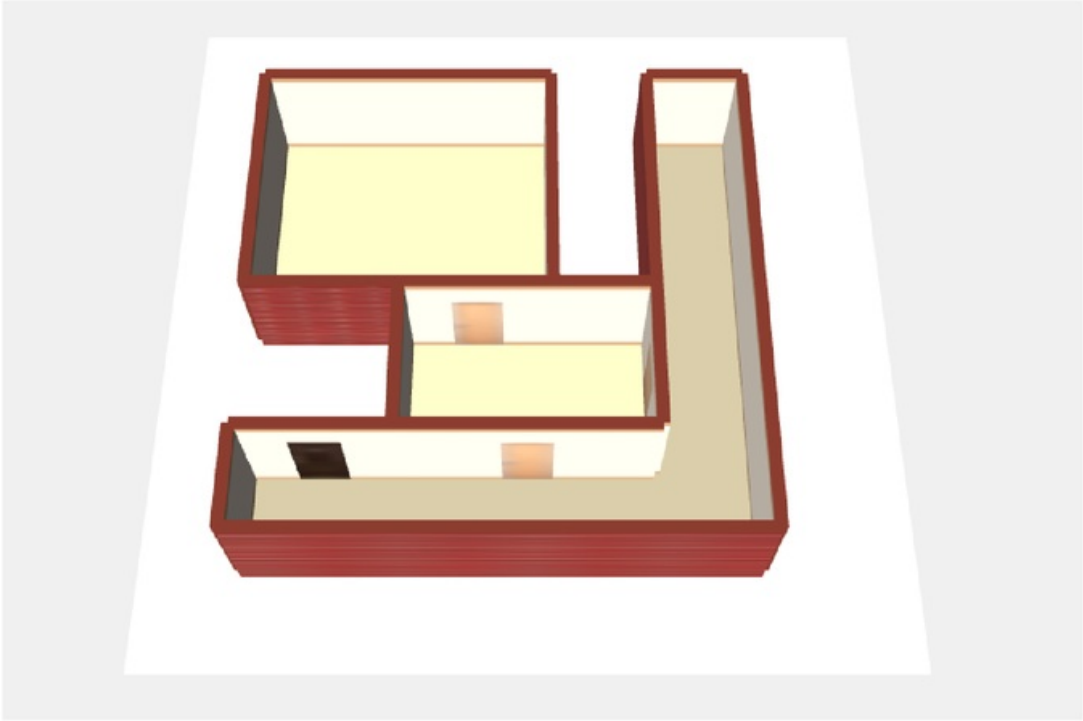


Figure 6.3.3 - The model created for each test iteration

6.3.2 Performance testing before improvements

Before any performance improvements were implemented, the base attribute values were recorded when the model was loaded by the model viewer. For each subsequent test, the values should improve upon the current values specified in Table 6.5.

Attribute	Previous value	Current Value
Calls	N/A	437
Vertices	N/A	2622
Faces	N/A	874
Geometries	N/A	27

Table 6.5 - Performance attributes before improvements

6.3.3 Fixing the performance issues – Stretching the walls

Firstly, it was important to determine exactly which part of the automatic wall algorithm was causing the performance issues. It was soon realised that each wall was composed of many small individual sections of wall, each spanning the width of one cell, as shown in Figure 6.3.4. Each section of wall was a cuboid, composed of 6 faces and 8 vertices, therefore, for 1 complete section of wall, spanning 10 cells, the wall contained 60 faces and 80 vertices. When an entire model was created using this method, thousands of vertices and faces were being rendered whenever the camera was moved, causing very poor performance.

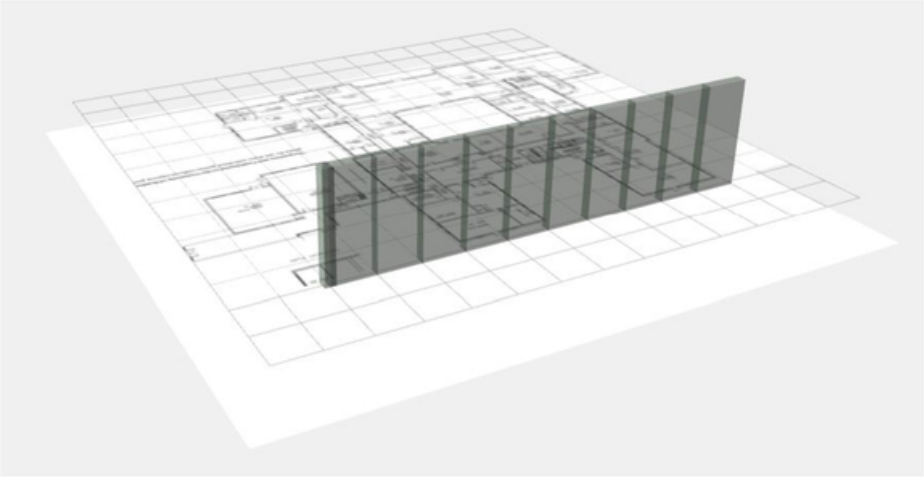


Figure 6.3.4 - 10 individual walls sections being created

The entire automatic walls algorithm was re-written to improve performance in this area. The old algorithm would analyse each individual cell along with its neighbouring cells, and determine which walls should be built around that cell. It would then proceed to erect each wall individually.

The new algorithm is composed of two parts. The first part traverses each cell along the z axis. For each cell the algorithm determines which, if any, walls need to be placed between the current cell and the cell above. It then stores this type of wall, along with the current cell coordinates, in an array, before moving on to analyse the next cell. If the next cell uses the same wall type as the previous cell, it's coordinates are added to the array. This process is repeated, comparing each cell with the cell above, moving along the z axis. The entire process is then repeated, only this time the algorithm traverses on the x axis; it compares each cell to the cell on its right and determines which wall should be erected.

Eventually, the algorithm will have compiled several different arrays, each containing a wall type and a list of coordinates that need that wall type. The highest and lowest coordinates are then extracted from each array and the wall type is used to build a wall between the two coordinates. Therefore, if 10 adjacent cells require the same type of wall, only 1 wall will be erected, spanning the entire length, as opposed to 10 individual sections of wall. This can be seen in Figure 6.3.5.

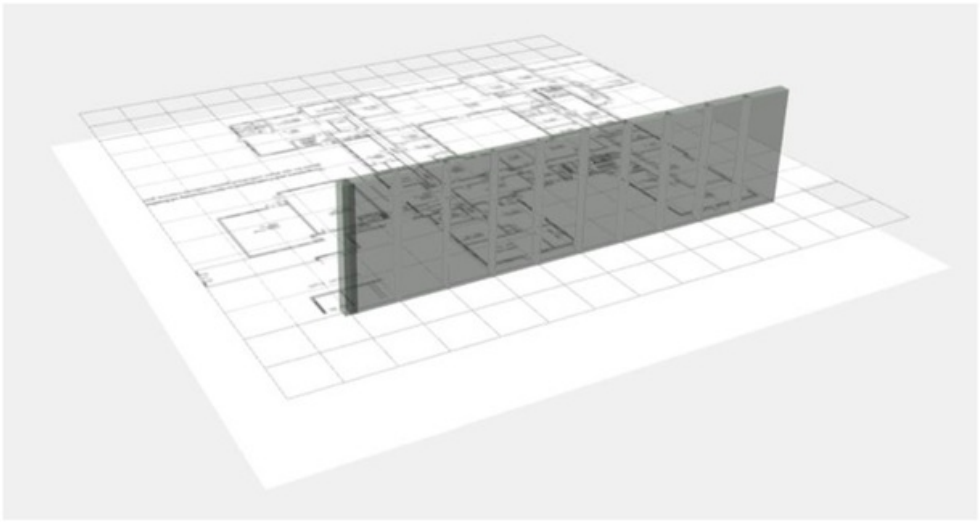


Figure 6.3.5 - 1 individual wall has been being created, spanning the entire length

Implementing the new algorithm caused in a huge increase in performance, as shown in Table 6.6. The number of vertices and faces created by the algorithm were much lower than the previous version, resulting in a much higher frame rate and significantly reduced lag, when moving the camera. However, it was noticed that the total number of geometries stored in memory was higher than it was previously.

Attribute	Previous value	Current Value
Calls	437	155
Vertices	2622	930
Faces	874	310
Geometries	27	30

Table 6.6 - Performance attributes after initial improvements

6.3.4 Fixing the performance issues – Merging geometries & materials

Expanding on the initial performance improvements, it was decided a few more changes should be made to the algorithm, to reduce the number of geometries stored in memory. As previously stated, the new algorithm created multiple arrays, containing a wall type and a set of coordinates. It then used the highest and lowest coordinates in each array to position the wall. It was found before each section of wall was placed onto the scene, the geometries present in each wall of the same type could be merged. Doing so resulted in a reduced total number of geometries present in memory, as shown in Table 6.7.

Attribute	Previous value	Current Value
Calls	155	155
Vertices	930	930
Faces	310	310
Geometries	30	14

Table 6.7 - Performance attributes after final improvements

A few other, minor performance fixes were also implemented at this stage, such as changing the type of materials used by each type of wall. The users of the model viewer could now load and interact with large, detailed models, without any performance issues. A detailed functionality test of the modelling suite at this stage can be seen in section 6.2.3 above.

Overall, the performance improvements implemented in the automatic walls algorithm resulted in a 64% performance increase when the models were exported and viewed in the model viewer.

6.4 Compatibility testing

To ensure the application is fully compatible with a range of devices and operating systems, several tests were conducted. As the modelling suite is only designed to be used on desktop computers and laptops, it will not be tested as rigorously as the model viewer. For each test, the device, operating system and browser was recorded, along with whether the application loaded successfully and could be used effectively. The most recent versions of each operating system and web browser were used for each test; as long as the browsers support WebGL, the application should pass the tests.

Test ID	Device	Operating system	Web browser	Pass/Fail
COMPMS1	Laptop	macOS	Chrome	Pass
COMPMS2	Laptop	macOS	Safari	Pass
COMPMS3	Laptop	macOS	Firefox	Pass
COMPMS4	Desktop	Windows 10	Chrome	Pass
COMPMS5	Desktop	Windows 10	IE 11	Pass
COMPMS6	Desktop	Windows 10	Microsoft Edge	Pass
COMPMS7	Desktop	Windows 10	Firefox	Pass

Table 6.8 – Modelling suite compatibility test results

Test ID	Device	Operating system	Web browser	Pass/Fail
COMPMV1	Laptop	macOS	Chrome	Pass
COMPMV2	Laptop	macOS	Safari	Pass
COMPMV3	Laptop	macOS	Firefox	Pass
COMPMV4	Desktop	Windows 10	Chrome	Pass
COMPMV5	Desktop	Windows 10	IE 11	Pass
COMPMV6	Desktop	Windows 10	Microsoft Edge	Pass
COMPMV7	Desktop	Windows 10	Firefox	Pass
COMPMV8	Tablet	iOS10	Chrome	Pass
COMPMV9	Tablet	iOS10	Safari	Fail
COMPMV10	Mobile	iOS10	Chrome	Pass
COMPMV11	Mobile	iOS10	Safari	Fail
COMPMV12	Mobile	iOS10	Firefox	Pass

Test ID	Device	Operating system	Web browser	Pass/Fail
COMPMV13	Mobile	Android 7.0 (OxygenOS 4.0.3)	Chrome	Pass
COMPMV14	Mobile	Android 7.0 (OxygenOS 4.0.3)	Firefox	Pass

Table 6.9 – Model viewer compatibility test results

Table 6.8 shows that the modelling suite passed every compatibility test, and works on every device that supports WebGL. Table 6.9 shows that the model viewer passes most of the compatibility tests, however it fails when tested on a device running iOS 10 and using the Safari web browser. This was investigated in greater detail; the conclusions and fix are recorded in section 7.1.

6.5 Usability testing

As usability testing aims to ensure the application is as user friendly as possible, the developer will not be involved with the testing process, other than to write the test cases. Two external users will be asked to complete the tests, and the results will be recorded by the developer.

6.5.1 Testing the modelling suite

Test ID	Description	Pass/Fail		Action taken
		User 1	User 2	
USRMS1	Can you load the modelling suite and select the correct floor plan?	Pass	Pass	None
USRMS2	Can you interact (zoom/pan/rotate) with the modelling suite scene?	Pass	Pass	None
USRMS3	Can you use the click-and-drag functionality to create corridors?	Pass	Pass	See section 7.1
USRMS4	Can you easily place entrances?	Fail	Fail	See section 7.1
USRMS5	Can you use the click-and-drag functionality to create rooms?	Pass	Pass	None

Test ID	Description	Pass/Fail		Action taken
		User 1	User 2	
USRMS6	Can you easily place doorways to rooms?	Fail	Fail	See section 7.1
USRMS7	Can you undo the room placement?	Pass	Pass	None
USRMS8	Can you build the walls of the model?	Pass	Pass	None
USRMS9	Can you export a model?	Pass	Pass	None
USRMS10	Is the GUI easy to understand and use?	Pass	Pass	None

Table 6.10 – Modelling suite usability test results

Table 6.10 shows that three usability tests for the modelling suite failed; USRMS3, USRMS4 and USRMS6. The reasons for these tests failing is discussed in section 7.1.

6.5.2 Testing the model viewer

Test ID	Description	Pass/Fail		Action taken
		User 1	User 2	
USRMV1	Can you load the correct model?	Pass	Pass	None
USRMV2	Can you interact (zoom/pan/rotate) with the modelling suite scene?	Pass	Pass	None
USRMV3	Can you use the GUI to search for a room inside a model?	Pass	Pass	See section 7.1
USRMV4	Can you use the GUI to plot a route between an entrance and a room?	Pass	Pass	See section 7.1

Table 6.11 – Modelling suite usability test results

Table 6.11 shows that all the usability tests for the model viewer succeeded, however USRMV3 and USRMV4 highlighted some important issues with the functionality. This is discussed in greater detail in section 7.1.

7. Discussion

To ensure that correct applications were developed, the results of the testing phase were analysed and reviewed. Both applications were also be tested against the initial requirements, specified at the beginning of the project.

7.1 Known issues

The validation testing completed in section 6 highlighted some key areas in both applications that needed to be improved. Table 7.1 displays the known issues within the applications along with how important it is for them to be fixed and whether any further action will be taken.

Application	Reference	Description	Priority	Action?
Viewer/Suite	N/A	Performance issues in the suite and viewer	High	Completed
Viewer	COMPMV9	Can't open models on tablets running iOS10 and Safari	High	To do
Viewer	COMPMV11	Can't open models on mobiles running iOS10 and Safari	High	To do
Suite	USRMS3	No clear instructions indicating how to use the modelling suite	Medium	None
Suite	USRMS4	No validation when placing entrances	Medium	None
Suite	USRMS36	No validation when placing doorways	Medium	None
Viewer	USRMV3	Hard to switch from room finder to route planner in GUI	Low	None
Viewer	USRMV4	Hard to switch from route planner to room finder in GUI	Low	None

Table 7.1 - Known issues within both applications

Due to having limited time, the developer will only fix issues marked as having a high priority. Section 6.3 descried the development steps taken to improve the performance of both the modelling suite and the model viewer. The other high priority issues lie within the model viewer, when loaded on a mobile device running iOS10 and using the Safari web browser. Safari's remote debugging tools were used to connect the mobile devices to the developer's laptop, allowing them to debug the application [29]. It was found that the function used to read the query string parameter was the culprit:

```
var urlParams = new URLSearchParams(window.location.search);
var model = (urlParams.get(m)) + '.json';
```

The 'URLSearchParams' function is not supported by devices running iOS10 and Safari [30]. This function was replaced with the following snippet of code. This code was tested on both devices; both tests passed successfully:

```
var model = getUrlParameter('m') + '.json';

function getUrlParameter(name) {
    name = name.replace(/[\[]/, '\\[').replace(/[\]]/, '\\]');
    var regex = new RegExp('[\\?&]' + name + '=([^&#]*)');
    var results = regex.exec(location.search);
    return results === null ? '' : decodeURIComponent(results[1].replace(/\+/g, ' '));
}
```

7.2 Initial requirements comparison

As no more development work will be undertaken on either application, they can be tested against the initial requirements gathered in section 2.3 of the PID. The requirements were split into two sections, the primary output (3D model viewer) and the secondary output (3D modelling suite).

7.2.1 The 3D model viewer

The first requirement for the 3D model viewer specified that it *must* be split into 3 separate applications; a web application, an iOS application and an Android application. When this requirement was specified, the capabilities of WebGL were unknown by the developer, therefore it appeared logical to expect 3 apps to be developed. When research was conducted into the capabilities of web browsers rendering 3D graphics (see section 3.1), it was soon realised that the application could in fact be web based, and loaded by a range of devices using their web browsers.

Secondly, the requirements specified that the application *should* be able to display routes over multiple floors within a model. As the modelling suite does not handle the ability to build multiple floors within a model, this requirement could not be satisfied.

Apart from the two requirements outlined above, the model viewer satisfied each of the 'must have' and 'should have' requirements specified in the PID. For these reasons, the solution developed, i.e. the model viewer application, can be regarded as a success.

7.2.2 The 3D modelling suite

Each of the requirements specified for the modelling suite have been satisfied, other than the ability for the models to contain multiple floors. Unfortunately, the developer lacked the time to implement this feature, however, the pseudocode has been written and will be implemented in the future.

Overall, both applications developed for this project satisfied the requirements outlined in the problem articulation, the technical specification and the solution approach. It was specified in the solution approach (see section 4.4) that the model viewer could not be deemed as acceptable if it is less efficient than asking a receptionist, or reading a physical sign. While the model viewer may never surpass the efficiency of asking for directions or using physical signs, it has almost definitely improved upon them, with regards to ease of use and accuracy. Therefore, both applications can be deemed as a success.

7.3 Application limitations

There are still certain limitations present in both applications, as outlined in Table 7.1. Each of the known issues marked with a medium or low priority will need to be amended in the future, as they impact the usability of the applications. However, as the modelling suite is not intended to be used by anybody other than the developer, the issues present within it may never need to be fixed. The GUI used by the model viewer on the other hand will need to be improved upon in the future. Section 10 addresses these, and other, future improvements.

8. Social, Legal, Health & Safety & Ethical Issues

The main social, legal, ethical and health and safety issues were outlined in the PID (see appendix 1). Even now, there are no obvious ethical or social issues related to this project. Perhaps the only social issue is that the rooms of certain members of staff may now be more accessible than before, however the information was always available, this project has just increased its visibility.

The legal issues discussed in the PID included gaining access to the floor plans for the selected buildings. To gain access to the floor plans, the Estates department at the University of Reading were contacted, and once the identity of the developer had been confirmed, access was granted. It was made clear that the floor plans were only to be used by the developer and were not to be shared with anybody else. Finally, since the Three.js framework is developed under the MIT licence [34], it's free to use, meaning this project has not impeded any software licencing laws.

9. Conclusion

To conclude this project, section 2.2 of the PID should be discussed. It was stated that the main objectives would be present in the design, development and testing of the applications. The first major development objective was to create a modelling suite, that allowed a modeller to create 3D model buildings, that could be viewed by end users in a web browser. The second objective stated that the users must be able to use these models to search for rooms within the buildings. Both objectives were satisfied by the creation of the modelling suite and the model viewer, which can be considered as the two key outcomes for the project. The third objective stressed the need for the model viewer to be packaged into two native mobile applications, using the PhoneGap framework. This objective was removed during the development process, due to the capabilities of the WebGL JavaScript API.

The rigorous testing of both applications ensured that they sufficed as a solution to the initial problem. However, in hindsight, further code testing frameworks, such as Mocha [35], should have been implemented throughout development, other than the usual smoke tests that were used by the developer. This however did not affect the usability, compatibility or performance of the final applications.

As an overall conclusion, the key outcomes of this project (the modelling suite and the model viewer) have been developed to the highest possible standard, within the given timeframe. They have both surpassed their initial requirements in terms of functionality, ease of use and aesthetics, and can be considered an incredibly profound solution to the initial problem.

10. Future Improvements

When the idea for this project was conceived, it was considered that the final solution would simply be an application that guided users between an entrance and a room within a building. However, as the development work progressed, the scope of this type of application has increased almost exponentially.

10.1 The modelling suite

Currently, the most time-consuming part of the process is the need to manually build each model, ever time. In theory, an application could be developed to automatically convert 2D CAD files, in the .dwg file format, into 3D models that can be viewed using the WebGL API. The modeller could then create different layers (such as an A* route planning layer) and apply them to each model, if needed.

10.2 The model viewer

Theoretically, there are endless opportunities that this type of application can take advantage of. There is a *vast* amount of information about buildings that has never been utilised and could be of great help to the public, for example:

- Hospitals:
  - Where certain departments are located
  - Which rooms have free beds
- Universities
  - Where a lecturer theatre/lecturer's office is located
  - Which computers are free to use in a lab
  - Which rooms are free for meetings
- Shopping Centres
  - Where certain shops/the help desk is located
- Airports
  - Flight times
  - Gate information
- Hotels
  - Room service information
  - Which elevators will take you to the correct floor

There is also an abundance of information available for almost every public building, for example.:

- Where the restrooms are located
- Where the nearest vending machine/water fountains/cafes are located
- Which entrances are suitable for the disabled
- Where the fire exits are located

All the above features could be implemented with ease, but the information they would provide with potentially thousands of users would be invaluable. As previously mentioned, the application will and *does* work on computers, smartphones and tablets. Users will not need to download or install any extra software as the application will work in their web browser and take seconds to load.

It's hoped that in the future, virtually any piece of information about a building can be 'plugged into' the models created with the modelling suite. The models could also offer services such as the ability to book a room for a meeting, sending the user's exact location to a friend, with directions of how to get there, or the ability to integrate with the user's device's calendar app, allowing them to see *exactly* where they are supposed to be. This information already exists, and is available for almost anyone to use, however nobody is utilising it. Offering this sort of information to the public, in an incredibly user friendly manner, is something that *will* happen eventually.

As mentioned before, this type of application could be used by several different establishments, including universities. There are over 150 universities in the UK alone, with an estimated number of 1.7m full time students [31], 532,300 of which were new in 2015 [32]. These new students would not know the locations of any important rooms on their new campus, either on the open day, or once they have enrolled. They would have to rely on 2D images of the building's floorplan to find a room, which are usually unclear and can cause confusion. This application could solve these problems, and this is just one of an infinite amount of problems that it can solve.

11. Reflection

Almost every part of this project has been a huge academic challenge for me. Firstly, I was unsure whether the idea I had proposed was even possible to develop, as it had never been attempted before. I had been writing basic JavaScript code for around 4 years, so I felt comfortable knowing that I would not have to learn an entirely new language, however working on this project has vastly improved my ability to use JavaScript correctly and effectively. I do however believe that I completely underestimated the work that needed to be done to develop a fully functioning modelling suite. In hindsight, I should have developed a plugin for an existing 3D modelling suite, such as Blender, that would have allowed me to create the 3D models automatically, as suggested in my Literature Review (see section 3.2).

Adopting an agile style of project management was a great way to ensure the project progressed smoothly. One project management technique however was not used; version control. Version control would have greatly benefited me throughout the development of both applications. I had used version control daily during my year in industry, so it was very narrow minded of me to not use it for this project.

Personally, I believe that this type of application could impact of the lives of potentially millions of people. In 10 years' time, people aren't going to be studying 2D floorplans in the receptions of large, public buildings, they're going to be using an application like Blueprint. The vast amount of information surrounding public buildings exists and is free to use, however nobody is utilising it. I believe that this type of application is the future and fortunately, this project has allowed me to take the first step towards that future. All in all, I'm incredibly happy and proud of the level of work I have achieved, and I'm incredibly excited about the future prospects of this application.

12. References

[1] The Complete University Guide, "The University of Reading," September 2016. [Online]. Available: https://www.thecompleteuniversityguide.co.uk/reading/. [Accessed October 2016].

[2] Bing, "Bing Maps welcomes Multimap to the family," September 2010. [Online]. Available: https://blogs.bing.com/uk/2010/09/29/bing-maps-welcomes-multimap-to-the-family. [Accessed October 2016].

[3] The AA, "AA Route Planner centenary," November 2012. [Online]. Available: http://www.theaa.com/newsroom/news-2012/aa-route-planner-100th-anniversary.html. [Accessed October 2016].

[4] e. Platform, Director, *ePSI workshop on Geo Data - INSPIRE 2014 (3/5)*. [YouTube]. 2014.

[5] B. McClendon, "A new frontier for Google Maps: mapping the indoors," November 2011. [Online]. Available: https://googleblog.blogspot.co.uk/2011/11/new-frontier-for-google-maps-mapping.html. [Accessed October 2016].

[6] MapsPeople, "MapsIndoors," 2016. [Online]. Available: https://www.mapspeople.com/mapsindoors. [Accessed October 2016].

[7] Mapswize, "Mapswize," 2016. [Online]. Available: https://www.mapwize.io/en/. [Accessed October 2016].

[8] eeGeo, "Indoor Maps," 2016. [Online]. Available: http://www.eegeo.com/features/indoor-maps/. [Accessed October 2016].

[9] Micello, "Micello Announces 15,000 Indoor Venue Maps Available Worldwide," 2013. [Online]. Available: https://www.micello.com/15k. [Accessed October 2016].

[10] CAPSLOCK101, "How long did it take for you guys to get good with blender?," January 2015. [Online]. Available: https://www.reddit.com/r/blender/comments/2rprwa/how_long_did_it_take_for_you_guys_to_get_good/. [Accessed October 2016].

[11] B. J. Diego Cantor, WebGL Beginner's Guide, Packt Publishing, 2012.

[12] M. Pesce, "iOS 8 release: WebGL now runs everywhere. Hurrah for 3D graphics!," September 2014. [Online]. Available: https://www.theregister.co.uk/2014/09/17/after_20_years_apple_finally_enters_the_third_dimension/. [Accessed October 2016].

[13] Creative Bloq, "Behind the scenes of 'Lights': the latest WebGL sensation!," November 2011. [Online]. Available: http://www.creativebloq.com/3d/behind-scenes-lights-latest-webgl-sensation-11116660. [Accessed October 2016].

[14] Ford, "Mustand Customizer," 2016. [Online]. Available: http://www.ford.com/cars/mustang/customizer/#!/customize. [Accessed October 2016].

[15] J. Dirksen, Learning Three.js, Packt Publishing, 2013.

[16] Xbox, "Xbox Design Lab," 2016. [Online]. Available: https://xboxdesignlab.xbox.com/en-US/customize. [Accessed October 2016].

[17] Noeticsunil, "Top 10 HTML5, JavaScript 3D Game Engines and Frameworks," May 2015. [Online]. Available: http://noeticforce.com/best-3d-javascript-game-engines-frameworks-webgl-html5. [Accessed October 2016].

[18] IndoorAtlas, "Our Platform," 2016. [Online]. Available: http://www.indooratlas.com/our-platform/. [Accessed October 2016].

[19] K. V. Kaifei Chen, "Design and Evaluation of an Indoor Positioning System Framework," 2016.

[20] A. Trice, "PhoneGap Explained Visually," May 2012. [Online]. Available: http://phonegap.com/blog/2012/05/02/phonegap-explained-visually/. [Accessed October 2016].

[21] A. Chakraborty, "PhoneGap - How Does It Work," December 2011. [Online]. Available: http://arnab.ch/blog/2011/12/phonegap-how-does-it-work/. [Accessed October 2016].

[22] All My Brain, "The Difference Between Dijkstra's Algorithm and A*," June 2008. [Online]. Available: http://allmybrain.com/2008/06/02/the-difference-between-dijkstras-algorithm-and-a/. [Accessed October 2016].

[23] W. Ertal, Introduction to Artificial Intelligence, vol. 1, UTiCS, pp. 98-99.

[24] 3Dlove, "HTML5 Game Devs," February 2015. [Online]. Available: http://www.html5gamedevs.com/topic/12789-choose-between-threejs-and-babylonjs/. [Accessed October 2016].

[25] M. Marschall, "Kanban vs Scrum vs Agile," July 2015. [Online]. Available: http://www.agileweboperations.com/scrum-vs-kanban. [Accessed October 2016].

[26] A. P, "Introduction to A*," 2010. [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. [Accessed October 2016]

[27] W. A. Company, Director, *Modeling and Texturing a Building in Blender.* [YouTube]. 2015.

[28] J. Petitcolas, "Importing a Modeled Mesh From Blender to Three.js," October 2015. [Online]. Available: https://www.jonathan-petitcolas.com/2015/07/27/importing-blender-modelized-mesh-in-threejs.html. [Accessed October 2016].

[29] Three.js, "Three.js Voxel Painter," 2016. [Online]. Available: https://threejs.org/examples/webgl_interactive_voxelpainter.html. [Accessed October 2016].

[30] dataarts, "GitHub - dat.gui," April 2017. [Online]. Available: https://github.com/dataarts/dat.gui. [Accessed October 2016].

[31] B. Grinstead, "A* Search Algorithm (Updated)," September 2010. [Online]. Available: https://briangrinstead.com/files/astar/. [Accessed December 2016].

[32] L. Atherton, "Remote debugging iOS Safari on OS X, Windows and Linux," 02 2015. [Online]. Available: https://blog.idrsolutions.com/2015/02/remote-debugging-ios-safari-on-os-x-windows-and-linux/. [Accessed February 2017].

[33] D. Walsh, "Get Query String Parameters with JavaScript," 08 2016. [Online]. Available: https://davidwalsh.name/query-string-javascript. [Accessed February 2017].

[34] Wikipedia, "Three.js," March 2017. [Online]. Available: https://en.wikipedia.org/wiki/Three.js. [Accessed March 2017].

[35] Mocha, "Mocha.js," March 2017. [Online]. Available: https://mochajs.org/. [Accessed March 2017].

[36] Universities UK, "Higher education in numbers," 2015. [Online]. Available: http://www.universitiesuk.ac.uk/facts-and-stats/Pages/higher-education-data.aspx. [Accessed March 2017].

[37] UCAS, "Record numbers of students accepted to UK universities and colleges this year," 12 2015. [Online]. Available: https://www.ucas.com/corporate/news-and-key-documents/news/record-numbers-students-accepted-uk-universities-and-colleges. [Accessed March 2017].

13. Appendices

Appendix 1 – Project Initiation Document

Department of Computer Science
University of Reading

Project Initiation Document

PID Sign-Off

Student No.	**22011265**
Student Name	**Joe Morgan**
Email	**vt011265@reading.ac.uk**
Degree programme (BSc CS/BSc IT)	**BSc CS**
Supervisor Name	**Mr John Roberts**
Supervisor Signature	
Date	**27th September 2016**

Appendix 2 – Logbook

School of Mathematical, Physical and Computational Sciences

Individual Project – CS3IP16

Blueprint – An Interactive Indoor Route Planning Application based on the University of Reading Campus - Logbook

Student:	Joseph Morgan
Student Number:	22011265
Supervisor:	John Roberts
Submission date:	2nd May 2017

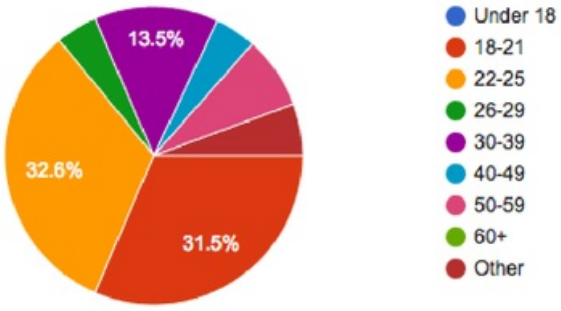
Appendix 3 – Survey results

A survey to find how navigation inside large buildings and establishments can be improved

89 responses

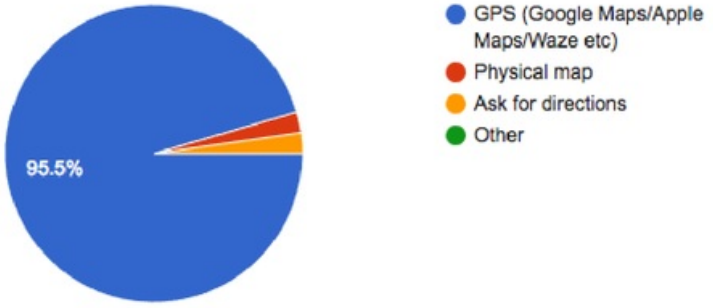
Publish analytics

Please select your age (89 responses)



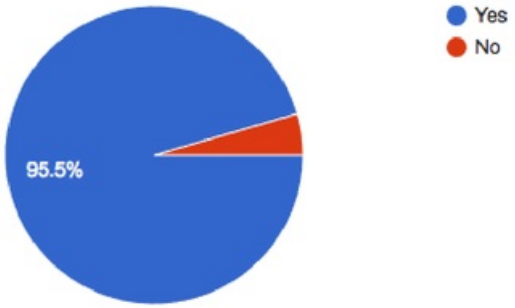
Question 1: If you were travelling (on foot/by car) to a place you had not been before, how would you most likely plan your route?

(89 responses)



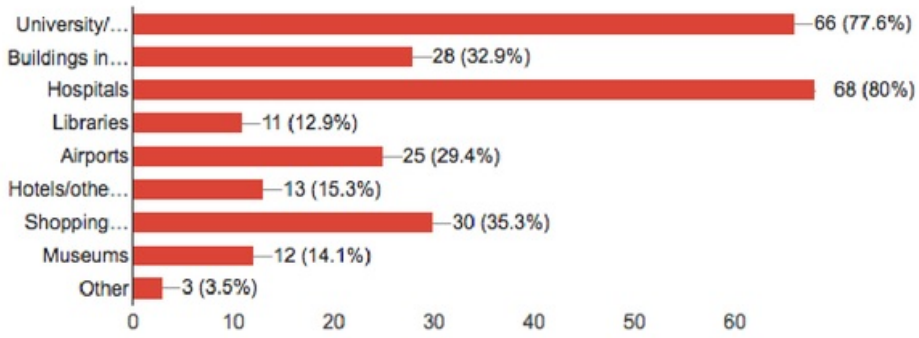
Question 2: Have you ever had trouble finding a specific room inside a building/establishment (e.g. university/hospital/shopping centre etc)

(89 responses)



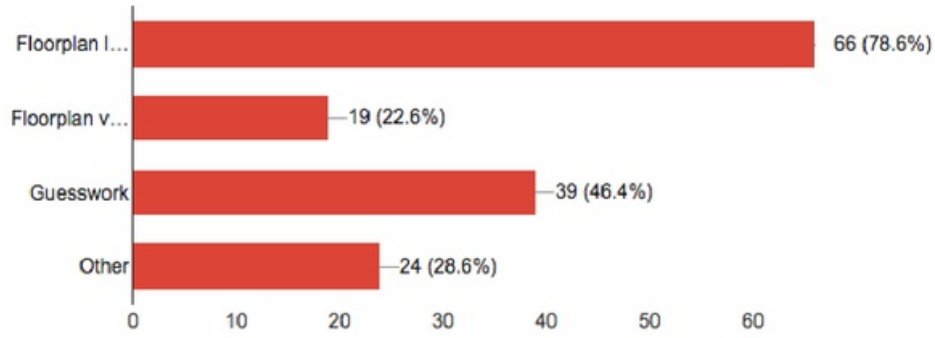
Question 3: If you answered yes to question 2, please select which type of buildings/establishments you have had trouble navigating. (Please add any other establishments in the 'other' section, using commas to separate)

(85 responses)



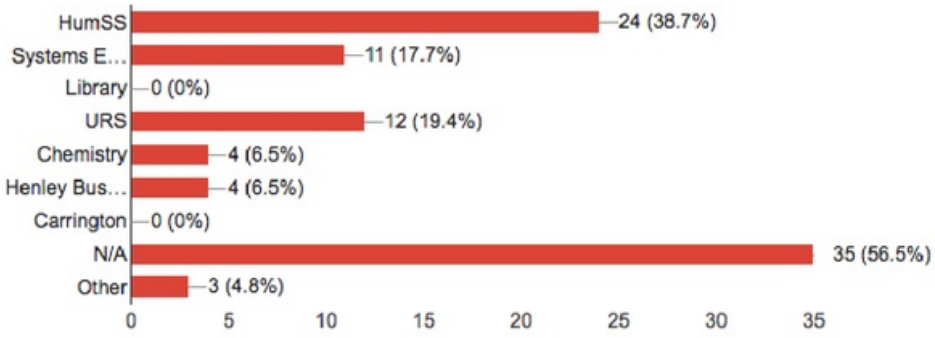
Question 4: If you answered yes to Question 2, how would you currently find a specific room inside a building/establishment that you are unfamiliar with?

(84 responses)



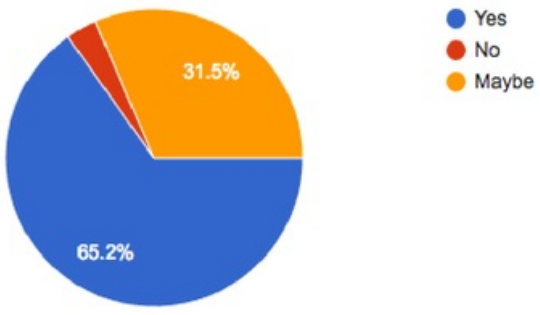
Question 5: If you attend (or have attended) the University of Reading, which campus buildings do you find most difficult to navigate? (Please add any other buildings in the 'other' section, using commas to separate)

(62 responses)



Question 6: If a room finding application was available for a specific establishment that you use (campus/hospital/airport/shopping center etc), would you use it?

(89 responses)



Question 7: If you answered yes/maybe to question 6, are there any specific features you'd expect the application to have? (e.g. finding the shortest path to a room from a starting point, works in a web browser etc)

(58 responses)

Needs the non-functional requirement of being quick to download and use. It would be a niche app if it was building-specific and therefore I may choose to uninstall after use.

Yes, a way to find the shortest route would be much more convenient and time saving

Estimated time of arrival

Specifically shows the route, time it takes from your starting point

-Shortest path to where you are going -Showing where you are in the building -All exits including fire exits -Points of interest e.g water fountain, vending machine -Where the toilets are -Different views e.g birds eye view -Search bar to enter room or offices -Where the stairs are -Disability points e.g lifts, disabled toilets

Shortest route and easiest for access

Map turn by turn guidance

Just a map of the building

Similar to maps, able to follow the path via the app

For me, it has to be very easy to use, older people shy away from technology and need help, it's like learning a new language, even the "basics" cause poblems.

Live directions or save directions for future use so you can use without 3G

Be able to show you directions from a selected entrance point and which flight of stairs or lift to get into.

A visual display of the route I need to take as opposed to written directions. Well optimised for smartphones.

I would prefer a browser based solution unless it was a building I visited frequently. Shortest path or simplest route would be useful, shortest may not be simplest to navigate. Option to visit other attractions en route e.g. for a museum. Estimated journey time.

Search for pacific room or shop and take me there.

Shortest path for sure

mobile friendly app as well as web app. Navigates like GPS and shows your current position.

Works in web browser, easy to use, can show me where I am and how to get to where I need to be

I would want the 'room finding app' to be really easy to use and I wouldn't want to have to go through a lot of effort to download and set it up. If I used an app like this for places I go to regularly (uni, shopping center) then I wouldn't want the app to take up a lot of storage space, as I'd keep it on my phone to avoid constantly having to re-download it. If it was an app for a hospital or a new airport (places that I don't go to as regularly) then I think I would spare the time to download the app on each occasion that I go. I would incorporate the downloading of the app as a stage in my planning for that trip, in order to make my visit that much easier.

Finding quickest route and disabled access for haemos

To have a booking service to which informs you if the room is in use and/or link to appointment system which tells you that the appointment/meeting room currently in progress is busy

Quickest route

Works on mobile phones and always primarily shows the shortest path

is there the potential to book the room for work/group tasks?
Path finder
If you're in a building and the gps picks up where you are on mobile then from lat and long it will automatically bring up the current building plan
Easy to read or set at a glance
Shortest path, including the most efficient way to enter the building
Show elevator/escalator locations
If applicable, if said room was open/closed? E.g, a uni room being used for exams would be 'closed', and shops in a centre would have listed business hours? Maybe even a share function if two parties are meeting at a said spot. Or even, a share location function like Apple which could show how far away people are to other users/friends?
Should be speedy to use (not just in performance, but also in UI use).
Quickest way, shortest way, and Via points
finding the shortest route from current position
SatNav-esque features
Shows shortest route, can be loaded and used offline like Google Maps
No
coordinate with calendar on phone (which has the room of your lecture on) so it automatically launches and knows where you need to go when you are on the way to the lecture. Maybe the closest water fountain/place to get a drink to the room.
Work on iPhones
ETA
I would want a "path finder" feature which actually directs you to the room via something like a red line on the screen. Imagine pokemon go, but instead of pokemon there is a real time line on your screen which directs you to the room you need! Wouldn't that be great?
Basic floor plan would do.
In app form
Finding the easiest route
Maybe a simple list of what normally happens in the room at what times.
Room titling, shortest path calculation, commenting on/suggesting new ideas on maps.
Shortest route, using/not using lifts
Sort of like google maps, where it gives you different options depending on your start
Works offline; shows the quickest route.
Starting point from your location, highlighted route to room on a map
If it was like used in a hotel, you could add some sort of 'room service request' button or something like that integrated into the hotels app or You could provide an information tab relevant to the type of location. i.e. flight times and when to be at specific terminals for an airport or shops with any sales currently going on in shopping center.
Web Browser (infrequent use therefore app might be too much effort for single use), Search for room name/code, Browse layers
Time taken to get there
how long it will take you to get to that room, the timetable for that room potentially
Yes
Make sure the map can be rotated. The floor map for my university is upside down for anyone coming from the main bus stop.
Ability to find and selectively choose your destination on a map/floorplan. So as an alternative to typing in a room number, the ability to select it from a map (as this shows surrounding rooms and facilities). Ability to save a specific route for repeat use (until you no longer require the app to reach your destination).
To locate your start position and give you directions similar to Sat nav to your location

Voice Guided Feature

Any other comments about the application/idea (where the application could be used/potential issues)

(28 responses)

It would be helpful for the application to be 'mobile friendly' so that it can be used on the go. A potential issue may be can I use the application without an internet connection?

-If you wanted to take the application further, show temperatures of each room within the building to give students a heads up if they are entering a sauna however that would require a interface with the buildings system and that is if its technology is that far advanced. -Again if you wished to take it further you could have an extra where it shows if the person you wish to see are in their office, but again that would require their assistance.

Some hotels are a nightmare, finding rooms etc. Could it be used abroad when not understanding notices ?

It would be nice if an app could show you the route and track where you are walking as for instance Hospitals seem to be very good at making signs disappear. However as some buildings such as hospitals can be notorious for not having any GPS signal this may prove difficult.

Could give you an estimated time to your destination. Would be useful in very large buildings such as airports.

Definitely Hospitals

Large office building for guests. Train stations for finding correct platform.

Certainly for the less able it would make sense to include routes that are friendly to the less able. It's hard enough without the added challenge of a disability.

Hospitals, Theme parks

Mostly all explained in my answer to Q.7 - I wouldn't want the app to be an effort to download. Also, any adverts/submission forms would put me off using the app. The app would have to be efficient enough to compete with the ease of simply asking a receptionist. I think an app like this would really work in busy places like hospitals, public libraries, uni campuses etc where there isn't always someone nearby to ask, or where you often need to get somewhere fast. Whereas many corporate buildings/workplaces have security guards/receptionists manning the front desk at all times.

Hospitals with a different route planner for disabled persons

Businesses located in large buildings which have multiple meeting rooms with a link to a screen at the front of the room with the room number and details about the meetings scheduled in that particular room

Google Maps integration, using the Google StreetView would be cool.

Probably would require more than just a floor plan for me to download a native app (if I can see a map elsewhere). A webbased version however I would use.

No

should be promoted by the university-more trustworthy maybe a feature embedded in a uni app so you don't have to download as many apps.

Pathing avoiding restricted areas

Incorporate emergency exits into the plan (at least 2)

Maybe when u get to a room it could tell you your nearest fire-escape and assembly point. Optional. A wing-nut is a wing-nut, not a butterfly-nut. Love from Dara
Locations could use a QR code on physical floorplans that can be scanned to download a map to the app.
From car parks to shopping centres/airports
only real potential issue I can think of is the lack of mobile internet connection that you find in large buildings, meaning that you can't download the map of the building. - Ceiron xoxox
Handful potentially interesting (but I'm sure pain in the ass to implement) ideas: On staircases click on them an up/down button appears then when you choose up or down it changes floor and re-centres the map on the same stair case + highlights them Highlight/colour-code helpdesks & receptions (In some work places this could mean different desks for different companies in different sections/floors) For offices, have overlays to highlight specific companies/team sections or to highlight all the meeting rooms/desk space/exec offices/break out areas/eating spaces/entertainment stuff separately A step above and beyond for internal use might be to overlay unbooked meeting rooms with a link to the room booking system, I'm sure this is applicable to university and other areas too On the search by "Free room" note, allow for filters with room functionalities, internet access, video conference capabilities, catering?, number of seats etc If it's for internal company use, having peoples/teams seating plans would be handy to search for specific people instead of just meeting rooms or offices Again if for internal, easy to use functionality to add/remove users in seating plans and/or room references would be cool
Use bright colours
Great survey joe
Typical university campuses with as many as 50 different buildings dont usually have digital copies of floor plans for every floor of every building. A significant effort would have to go into data collection and recreating the floor plan in a compatible software. While such an app would be very useful, it would require a lot of effort. Perhaps more effort than just increasing physical navigation aids present in buildings (signs, maps, arrows).
Could be used by emergency services for locating people in buildings
Hospitals, Shopping Complex, Specific Government Offices / Ministries

Joseph Morgan | vt011265 - Individual Project Report

GRADEMARK REPORT

FINAL GRADE

/100

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57

PAGE 58

PAGE 59

PAGE 60

PAGE 61

PAGE 62

PAGE 63

PAGE 64

PAGE 65

PAGE 66

PAGE 67

PAGE 68

PAGE 69

PAGE 70

PAGE 71

PAGE 72

PAGE 73

PAGE 74

PAGE 75

PAGE 76

PAGE 77

PAGE 78

PAGE 79

PAGE 80

PAGE 81

PAGE 82

PAGE 83

PAGE 84

PAGE 85

PAGE 86

PAGE 87

PAGE 88

PAGE 89

PAGE 90

PAGE 91

PAGE 92

PAGE 93

PAGE 94