Final Project Report
3D Engine and Turn-Based Strategy Game with AI

Author: Josh Hornsby (C1627813)
Supervisor: Dr Frank Langbein
Moderator: Dr Matthew Morgan

One Semester Individual Project (CM3203)
40 Credits

Abstract

The project details the design and development of a 3d engine, game and artificial intelligence for a turn-based strategy game. It will assess the feasibility of writing a game engine in a low-level language, design and analyse a rule-based artificial intelligence, and detail the implementation of the game, engine and AI. It is not always obvious if or why a developer would implement their game engine. This report details what work is required to make an engine, or could provide the lower level knowledge required to do so. The report will also analyze a rule-based artificial intelligence implementation, and evaluate how effective it is for use in modern strategy games.

Acknowledgements

I would like to thank my supervisor Dr Frank Langbein for his mentoring throughout this project. His support, feedback and motivation in weekly meetings helped to keep the project on track. I would also like to thank my friends and family for supporting me and giving valuable feedback on the game.

Contents

1 Introduction **8**

2 Background **9**

3 Specification and Design **10**

3.1 Data-Oriented Design 10

3.2 System Architecture 10

3.3 Libraries 13

3.3.1 BGFX 13

3.3.2 GLFW 14

3.3.3 GLM 14

3.3.4 STB 14

3.3.5 Libmorton 15

3.4 STL Replacements 15

3.5 Engine 17

3.5.1 Asset System 17

3.5.2 Images 17

3.5.3 Fonts 18

3.5.4 3D Models 20

3.5.5 Shaders 22

3.5.6 Graphical User Interface 24

3.5.7 Mouse Raycast 25

3.6 Gameplay 27

3.6.1 Rules 27

3.6.2 Map 27

3.6.3 Map Generation . . . 27
3.6.4 Mouse Picking . . . 31
3.6.5 Units . . . 33
3.6.6 Unit Actions . . . 34
3.6.7 Action-bar . . . 35
3.6.8 Cover and Shot Chance . . . 38
3.7 Artificial Intelligence . . . 41
3.7.1 Minimax . . . 41
3.7.2 Alpha-Beta Pruning . . . 43
3.7.3 Monte Carlo Tree Search . . . 43
3.7.4 Evaluation Function . . . 44
3.8 Optimizing the Evaluation Function . . . 45
3.8.1 Temporary Game States . . . 47

4 Results and Evaluation **48**
4.1 Engine . . . 48
4.2 Artificial Intelligence . . . 48
4.2.1 Minimax . . . 49
4.2.2 Monte Carlo Tree Search . . . 49
4.2.3 Evaluation Function Tuning . . . 49
4.3 User Feedback . . . 51

5 Future Work **54**
5.1 Engine . . . 54
5.2 Gameplay . . . 54
5.3 Artificial Intelligence . . . 55

6 Conclusions **56**

7 Reflection on Learning **57**

List of Figures

1 Structure of the modules in the engine code 12

2 Structure of the modules in the gameplay code 13

3 The two faces before using vertex indexing 21

4 The two faces after using vertex indexing 21

5 A cube rendered in a single color 23

6 A cube rendered in with ambient and diffuse lighting 23

7 The shader code to calculate the color of a pixel after diffuse lighting 24

8 Draw functions for colored rects, textured rects and fonts 24

9 A diagram visualizing how a 2d screen coordinate of the mouse is transformed into a ray that is then intersected with a 2d plane 26

10 The shader code to calculate the color of a pixel after diffuse lighting 26

11 A map generated with just a road winding from the top right of the map to the left 28

12 A map generated with segments displayed in random colors 28

13 A map generated with simplified segments displayed in random colors 29

14 A map generated with buildings in segments that fit the criteria, and the simplified segments shown in random colors 30

15 Example 1 of a fully generated map 30

16 Example 2 of a fully generated map 31

17 Example 3 of a fully generated map 31

18 Psudocode implementation of an algorithm to get the location of intersection of a ray and plane 32

19 Two friendly units, the green unit showing that this is the current unit the user has selected 33

20 The action-bar for the currently selected unit 35

21 The action-bar message, shown just above the action bar 35

22 The user interface after entering move mode 36

23 The user interface after entering shoot mode 37

24 The combat log, shown in the bottom left of the screen 38

25 An illistration of partial line of sight calculation 39

26 An illistration of no line of sight calculation 40

27 A diagram showing a Minimax tree of a game with 2 players (P1 in green and P2 in red) who can each take 2 actions at any point of the game 42

28 A diagram showing how a z-order curve maps a 2d 4x4 box into a spatial index 47

1 Introduction

This project aims to create a 3d game engine, build a turn-based strategy game with the engine, and create a competent rule-based AI for the game. All of the code is implemented in a low-level language using only fundamental libraries. The engine includes a few features for graphical fidelity, but this is mainly out of scope, and the project focuses on other engine features as well as the gameplay and AI. The game has friendly and enemy units, with the enemy units controlled by the AI. Friendly units controlled by a human player using the games user interface and units can be moved around the square map and shoot the enemy units to kill them. In order to win the game, all units on one team must be defeated.

The intended audience of the project is anyone in a small or solo game development team wanting to assess the amount of work that goes into creating a game engine and a game together, and the advantages of this approach. The project source-code provides a good starting point for a fully functioning engine, and also for reference on how to implement standard features that new game programmers may find challenging.

The outcome of the project should be a user-friendly game that is enjoyable, easy to play without much prior knowledge and provides decent replayability. The engine should support the game so that it can include all the features required, and should allow the game to run well on modern hardware running the Windows operating system. The AI should be a challenge for a human player to play against, move in a way that the user can predict but is still hard to win against, and not take too long for the AI to evaluate and perform actions when the human player finishes their turn.

The report starts by in the background section with how the project started, what research informed it, its inspirations and an overview of the problem space. Then it moves on to give an overall structure of the system, and how the final product was designed and implemented in the specification and design section of the report. The project is then thoroughly tested and evaluated, seeing the usability and performance of the engine, the effectiveness of the gameplay, and the success of the AI as a competent opponent. The report is summed up with a discussion of future work that could improve the game, conclusions of the evaluation and project as a whole, and finally a reflection on the learning experience throughout the project.

2 Background

AI in computer games has been around since computer games themselves. In older games like pong, the AI was as simple as predicting where the ball would move to and move the paddle to that location. However, modern games have highly advanced AI, and even AI for traditional games like chess and go are exceptionally well researched. In modern strategy games such as XCom 2 [9] use behaviour trees to achieve their AI. However, this leads to an AI that is hard to train, tweak and to develop. This works well for large teams working on a massive game, but for a small team working on smaller projects, this may be less feasible to implement.

Likely stakeholders are people in or trying to enter the game industry, especially those in the indie game developer space. These developers will be looking for proof that writing an engine from scratch is fesible, and that it is worth the effort that it requires. Other stakeholders may include those interested in AI development, particularly in games

A great article by Louis Lafair [11] was a big inspiration for the AI approach for this game, especially for the AI. The article explains how Louis used Minimax and board evaluation functions to create an AI for his own board-game Pathwayz. The project was highly inspired by this piece, but since his game Pathwayz is very different to the strategy game within this project, it would be an interesting investigation into how effective a similar method that worked successfully in Pathwayz and chess could be applied to strategy video games. Another existing piece of theory within this area is another article [5] which talks about how the game Total War: Rome II [2] used Monte Carlo tree search. This was particularly interesting and inspiring as Total War is a similar game to the one this project is developing.

Because of time constraints, the number of features in the game (especially the number of actions that can be performed by each unit) were to be limited. Not only would this save development time of the game to focus on the engine and AI, but it also ensures that the Minimax AI would not have too many possible actions to evaluate, in order to try to get the most out of the algorithm within the short time available.

3 Specification and Design

The following section details the specification and design of the engine, gameplay and AI systems down to the pseudocode level. The section provides enough information to replicate the entire system architecture, without showing and commenting on all lines of code (approximately 4700 lines).

3.1 Data-Oriented Design

Instead of the prevalent Object-Oriented Programming, the project code adheres to the rules of Data-Oriented Design (explained thoroughly in a lecture by Mike Acton [1]), which is a methodology growing in popularity in the game industry. Data-Oriented Design aims to focus only on what the paradigm considers the purpose of programs - to transform data from one form to another. It stresses the importance of performance (games need to run at 60fps on various hardware), compile times (turn-around time is essential to implement features quickly and fix bugs), maintainability and limiting abstractions (abstractions make the program more complicated, not less). For maintainability, the methodology stresses to add as little abstraction as possible while still maintaining an efficient working environment. By adding abstractions, more is added to the problem which makes the solution more complicated, instead of making it more straightforward as Object-Oriented Programming implies. For compile times, the paradigm encourages avoiding the use of C++ templates, the standard template library, multiple inheritance and operator overloading. Not only do these features drastically increase compile times, but they also make the program harder to debug and the program more complicated.

Not only is this a learning experience, but could also be an evaluation of how this methodology works when making a game in a one person team, and comparing its results with Object-Oriented Programming which it aims to replace.

3.2 System Architecture

The code is split up into .cpp and .h files, each segment of the engine and gameplay code has a "module", and each module has a corresponding .cpp and .h file. A comprehensive diagram of all modules split up into the engine modules, and the game module, in figures 1 and 2. The arrows between modules show how the modules interact, a module with an arrow pointing to another means that the module uses the other module to perform its tasks.

The code of the engine and gameplay are individually separated to maintain low coupling, but modules inside each part of the engine and gameplay form systems (e.g. the asset system) which ensures high cohesion. In the engine, one of the systems is the asset system, which

uses the asset manager to store and retrieve assets, and individual asset modules to handle loading the asset (shader, image, mesh or font) from disk using the file module to be stored in memory in a format the engine can use. Another system in the engine is the graphics engine. The graphics system uses assets loaded from the asset system and draws them either in world space using the Graphics module or in screen space using the GUI module. The graphics system also handles the window (resolution, fullscreen or windowed, and more) to which the graphics are drawn. It also handles control of the camera. Finally, some extra utilities for math, physics and mouse and keyboard input are in their own "utility" system.

In the gameplay code, there are three other systems. The first system is the Turn system. This system uses the Turn module to coordinate human and AI turns. This includes passing over the turn to the AI system when the human player has finished their turn, and let the human player control its turn using the user interface provided by the Action-bar module to run particular actions in the Action module. Another system in gameplay code is the AI system, which uses the AI module to control the AI's turn, using the Minimax module for searching for the best action, and the Minimax module uses the Board Evaluation function to get the current evaluation of the map. The last gameplay system is the map system. This system uses the Map module to hold the state of the map (where entities and cover are), as well as altering it. When the Map module is loaded, it uses the Map Gen module to generate the map and the Entity module to manage the locations of entities on the map.

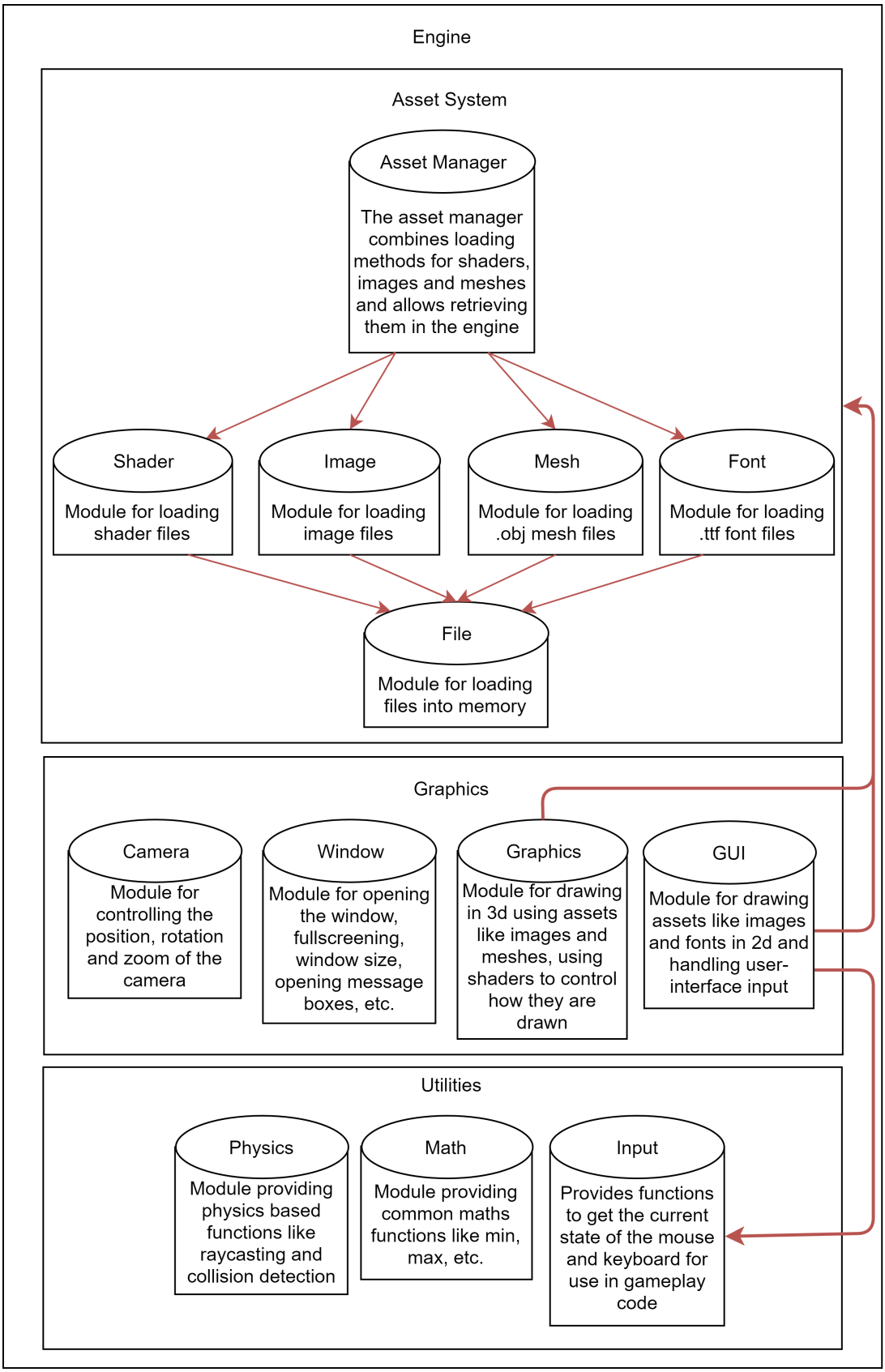

Figure 1: Structure of the modules in the engine code

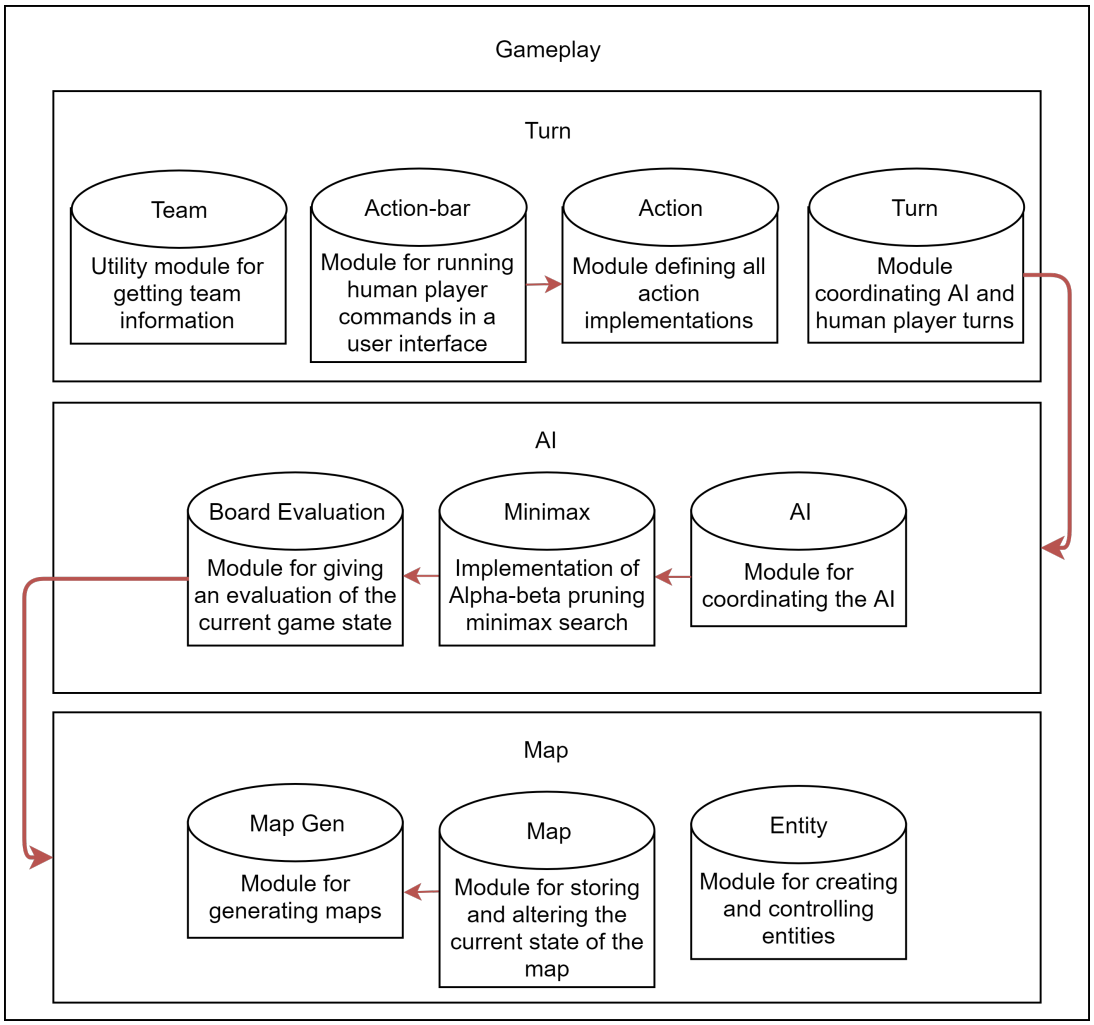

Figure 2: Structure of the modules in the gameplay code

The engine and gameplay code could easily be split up into two separate projects, and the game could include a dll with all the engine code. This would be preferable in future if another game were to be made with the same engine, the code is set up correctly for this (none of the engine modules has any gameplay specific code). However, instead, the engine and gameplay code is in one project, mostly for the simplicity of development.

3.3 Libraries

3.3.1 BGFX

BGFX [10] is a cross-platform rendering library that supports multiple rendering backend APIs such as Direct3D, Metal, OpenGL and multiple platforms such as Windows, Linux,

iOS, Android and more. This library was chosen due to its cross-platform support and support for multiple rendering APIs, but still being low level and not providing any game engine, which was part of the scope of the project to develop myself. BGFX also provides a shader tool which compiles GLSL-like shader code into compiled shaders (explained further in section 3.5.5) that can be used with all of the supported backend graphics APIs.

Overall, BGFX is excellent and even has C bindings which were used because they suited the project code style more than the C++ API. The project has excellent documentation. However, the examples are convoluted and tend to span across many files which make them harder to understand. If the examples were one file implementations of specific applications, the speed of following and understanding how they are implemented in the API would be quicker.

3.3.2 GLFW

GLFW [7] is a cross-platform library for creating windows and handling input on Windows, macOS and many Unix-like systems. When combined with BGFX, this library allows the engine to run on multiple desktop platforms. The window is created using this library in the "window" module, and then the window is passed to BGFX for it to render onto the window. Although the engine uses the input handling code, all of the functionality is hidden inside the input module, so the gameplay code interacts with this engine module rather than the GLFW library directly.

3.3.3 GLM

GLM [8] is a math utility library, providing functions for vector and matrix math required in 3D games. Most of the use of the library in the engine is used for controlling the camera as most modern graphics libraries use model, view and projection matrices.

3.3.4 STB

The STB libraries [4] are single file public domain libraries. Specifically, stb_image and stb_truetype libraries were used for loading image files and for loading TrueType fonts. The way these libraries are used for image and font loading are explained in sections 3.5.2 and 3.5.3 respectively. None of this library is used in gameplay code, as the engine hides this functionality, the gameplay code only interacts with the asset system (explained in section 3.5.1).

3.3.5 Libmorton

Libmorton [3] is a header-only library to encode and decode coordinates into Morton codes quickly. It was used when optimizing the evaluation function by trying to avoid cache misses using a geospatial index like Morton codes (both the optimization and Morton codes are further explained in section 3.8).

3.4 STL Replacements

The standard template library (also referred to as STL) is a C++ library providing implementations of common data structures. However, due to the aim of the project of using data-oriented design for the code (explained further in section 3.1) and more specifically the avoidance of C++ templates to avoid poor compile-times; the engine code as a custom implementation of a hash table, dynamic string and dynamic array. Although partly due to data-oriented design, there also was a desire to understand the implementations of data structures used throughout the project, as well as in general understanding more about how to efficiently manage the memory of the program entirely by the engine.

Hash Table This is a regular hash table implementation in C/C++, implementing methods to add a new values with a specific key, getting an element with a key, removing an element at a key, as well as a few default hash functions: splitmix64 [14] (with a different magic number [12]) for unsigned 32-bit integers and djb2 [15] for strings. The implementation is fundamental and uses the open addressing method for collision resolution, where if there is a collision on adding an element in a bucket, the element is put in the next empty bucket. This is not the best collision resolution method, but for my purposes it performs well as long as the hash-table has enough buckets relative to the number of elements that are likely to be inserted and that the hash function spreads the keys across the table effectively, and it removes the requirement of each bucket containing a list of elements.

Dynamic String Dynamic string is an implementation of a dynamically expanding string. The dynamic string supports appending various types such as other strings, single characters and integers, and clearing and trimming the string. The reason for using it is because strings can be modified and appended to each other quickly without the need for any memory allocations (as long as the internal buffer is long enough to hold the modified string). If the internal buffer is not large enough for a modification to be applied, the capacity is expanded to the larger of the required capacity and $n * 2 + 2$ where n is the old length of the internal buffer. The reason that the larger of these two values are used is so that if a lot of smaller things are appended to the string, the number of allocations is reduced. This is because $n * 2 + 2$ allocates more memory than is required but stops further small appends needing to

do more allocations. However, still, if a long string is appended only the required memory is allocated.

For example, if 10 separate characters are appended one-by-one to a 32 character string, only one allocation is made on the first character appended to make the internal buffer of size $32*2+2 = 66$ bytes long, for the rest of the appended characters the rest of the extended buffer is used before another allocation is required. If the internal buffer is only extended by 1 byte each time a character was added, it results in 10 separate allocations. However, if a 1 billion character string is appended to the same 32 character string, only 1 billion + 32 bytes will be allocated, instead of the 2 billion bytes if the $n * 2 + 2$ calculation was used to extend the buffer (which is a difference of 1 billion bytes, or an entire gigabyte of saved memory).

Dynamic Array The dynamic array is an implementation of an automatic expanding array, used as an alternative to a linked-list or a statically sized array. When adding a new element to the array, if the internal buffer is not big enough to fit the new element, it is extended to size $n + n/2$ where n is the current size of the array by allocating more memory to the internal buffer. When removing an element from the array, any elements after the removed elements are copied forwards in the array. This is to keep the elements in continuous memory, which is the main advantage of an array over a linked-list, since accessing contiguous memory instead of fragmented memory is faster due to caching.

3.5 Engine

3.5.1 Asset System

The asset system supports loading and usage of assets throughout the engine.

The following types of asset are supported:

- Mesh - Wavefront .obj files
- Images - most popular image formats supported such as: JPG, PNG, TGA, BMP, PSD, GIF
- Shaders - GLSL style compiled shaders using the BGFX tool shaderc
- Fonts - TrueType fonts (.ttf)

The engine requires explicitly registering assets. For example, when loading an image, it is as simple as using the image loading method in the engine (image_load) to load the image and then passing the returned image struct to the asset manager with the asset_manager_register method. To register an asset with the register method, the struct must extend the asset struct; this is a rare example of the use of C++ features in the engine. The asset struct has an asset_id string which is a unique id for each asset and an asset_type (asset_type is an enum, with values like ASSET_TYPE_IMAGE) which helps ensure that when getting an asset from the asset manager, the asset requested is of the expected type.

Assets are stored in a hashtable, with the key being the asset id, this means that the lookup time for assets is constant no matter how many assets are registered. Assets can be retrieved using the asset_manager_get_asset method and then casting the resulting asset based on the asset_type field. They can also be retrieved based on their types explicitly, by passing an asset_type to the asset_manager_get_asset method, or by calling asset_manager_get_(asset type) where asset type is the name of the asset type, for example, to get an image from the asset manager call: asset_manager_get_image with the asset_id as the argument. This method then returns the asset as an image, ready to use.

3.5.2 Images

Images are used in the engine for GUI elements, but can also be rendered in world space. Images are loaded using the stb_image library (explained in section 3.3.4), and therefore can be loaded in most popular formats: JPG, PNG, TGA, BMP, PSD and GIF. The library has a function to load an image file from a specified path, along with a few pieces of information like what pixel format to return (for example, return four channel RGBA pixel array). The

function then returns the width and height of the image, along with an array of pixels in the specified format. This pixel array is then passed to BGFX to upload the picture to the GPU and return a texture handle. This texture handle can then be used when drawing a 2d quad or even a 3d model.

All of this functionality is wrapped in a single function call in the image module called image_load which takes only one argument of the name of the image file. This then does the above processing and returns the user an image struct, the data model for which is found in table 1. The image struct can then be passed to a draw function such as gui_draw_image (described in section 3.5.6) which will then use the image handle in the image struct to draw a quad with the image as a texture on top in screen space.

Name	Data Type	Description
asset_id	string	The ID of this asset in the asset system
asset_type	enum	The type of this asset, will be ASSET_TYPE_IMAGE
width	unsigned 32-bit integer	The width of this image in pixels
height	unsigned 32-bit integer	The height of this image in pixels
handle	texture handle	The graphics API's representation of this image as a texture on the GPU

Table 1: The data model for the image struct

3.5.3 Fonts

Fonts are rendered as part of the GUI system, although world space rendering is not supported in the engine currently, it could easily be supported. TrueType fonts are one of the most popular font file formats and are loaded into the engine with help from the stb_truetype library (explained in section 3.3.4). First, the ttf file is loaded into memory using the engine's file_load method in the file module which opens a file and reads its contents into an allocated memory buffer. This memory buffer is then passed to stb_truetype by the stbtt_BakeFontBitmap method which processes the ttf file data and renders characters to a texture atlas. The method returns an array of pixels for the atlas as well as data about the characters (such as their location in the atlas, kerning, width, height). The atlas data is then loaded into an image struct (defined in table 1) just like other images in section 3.5.2. The image and character data is then combined into a font structure (the data model for which is in table 2). This font struct can then be passed to gui_draw_text function (described in section 3.5.6) which then uses the img and char_data in the font struct to draw each character in the text passed to the draw text method. The font is then drawn with each character onto an individual quad, setting the UV coordinates of each vertex to map to the texture

coordinates of the current character (calculated with the char_data of the font).

Name	Data Type	Description
asset_id	string	The ID of this asset in the asset system
asset_type	enum	The type of this asset, will be ASSET_TYPE_FONT
img	image (see table 1)	The font texture atlas
char_data	bakedchar	Character data (position in texture atlas, kerning, etc)

Table 2: The data model for the font struct

3.5.4 3D Models

For loading 3D models into the engine, the Wavefront .obj file format was chosen. This format is a simple, well supported and human-readable format to store 3D models. The main reason for using this file format was its simplicity for loading into the engine quickly, as the format is well documented and has lots of previous example code for loading these files. The file format is also supported by blender, the tool used in this project to create and export 3D models.

In a Wavefront .obj file, a geometric vertex is denoted with a "v", for example: "v x,y,z" where x, y and z are the coordinates of the vertex in 3D space. "vt" denotes a texture coordinate in the following format: "vt u,v" where u and v are the x and y coordinates of the texture. Another part of the file format is normals, denoted as "vn" and used as followed: "vn x,y,z" where x, y and z are the direction of the normal.

The vertexes, texture coordinates and normals are then combined into "faces" and referenced by their "index" (they are stored in ascending index order starting from 1). Faces are made up of 3 parts, each containing: a vertex, texture coordinate and normal corresponding for that specific vertex on the face. These three parts make up a single "face" or triangle in a 3D model. Faces are denoted in the file with the prefix "f", and formatted as so: "f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3" where v1 through v3 are the three vertices, vt1 through vt3 are the texture coordinates for v1, v2 and v3 and finally vn1 through vn3 which are the corresponding normals for those vertices.

The loading of the file is trivial, read each line storing the vertices, texture coordinates and normals in the order they are read from the file, storing them in an indexed data structure, and then building an array that contains each vertex, texture coordinate and normal in a struct referred to as a "pos_normal_vertex" (outlined in table 3 . This array is then used to create what is referred to as a mesh in the engine.

To reduce the amount of GPU memory used to store a single mesh, any duplicated vertices are removed from the array, and an array of indices is passed to the graphics API along with the vertex data to tell the GPU what data to use for the current vertex. An illustration of this technique is shown in figures 3 and 4. Figure 3 shows two faces with six vertices, with the vertices for face one shown in purple, and for face two shown in red. However, two sets of the vertices in the middle are shared and therefore have the same vertex positions. Using vertex indexing, these can be combined (as long as their texture coordinates are the same) into one vertex to save memory. The indexed vertices are shown in figure 4 where the green vertices are where two or more vertices with the same position have been combined into one. When the vertices are combined and referenced by index, the normals of the two old vertices are combined by adding them together and normalizing them.

Name	Data Type	Description
x	32-bit float	X position of the vertex
y	32-bit float	Y position of the vertex
z	32-bit float	Z position of the vertex
u	32-bit float	The x texture coordinate for this vertex
v	32-bit float	The y texture coordinate for this vertex
normal	unsigned 32-bit integer	The x, y and z of the normal vector, packed into a single 32-bit integer (1st 8 bits for x, 2nd 8 bits for y, last 8 bits for z, final 8 bits are unused)

Table 3: The data model for the pos_normal_vertex struct

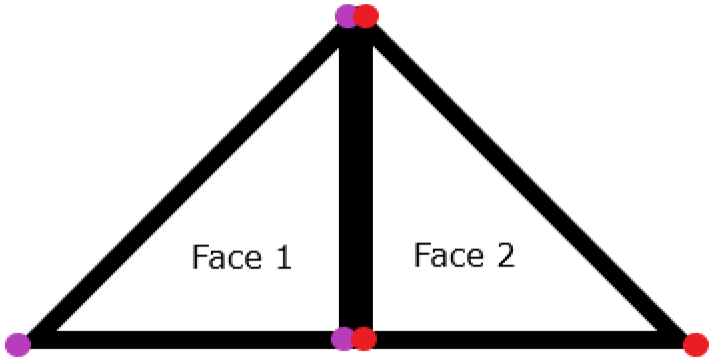

Figure 3: The two faces before using vertex indexing

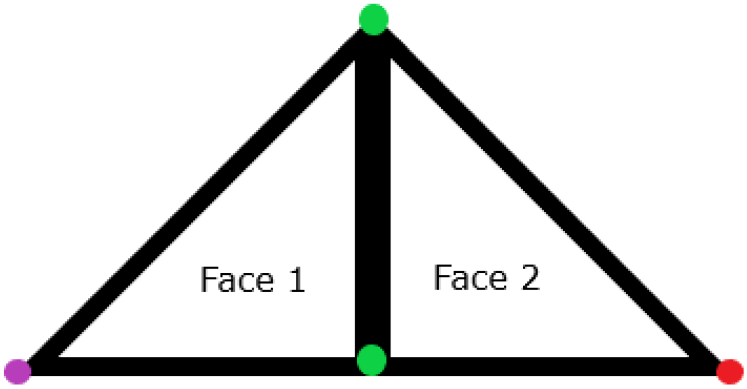

Figure 4: The two faces after using vertex indexing

3.5.5 Shaders

A shader is used to control how the engine renders different models and images. Shaders are written in a GLSL-like language and then compiled into a graphics API specific binary file (a binary must be compiled for each required graphics API, e.g. one for OpenGL, one for DirectX, Metal and more) using a tool called shaderc which is provided by BGFX. The engine provides some default shaders, listed below, but the game can implement its shaders if required.

Engine Shaders

- GUI shader - a shader only used for user interface elements, with support for textures (images) but not lighting
- Font shader - similar to the GUI shader in that it does not support lighting and does support textures, but the colour of the text is not based on the texture but a specified tint colour, the texture only controls the alpha (otherwise all fonts would be black as this is what colour the font texture atlas is)
- Colored shader - a GUI shader but without support for textures, the colour of the mesh is defined only by a specified tint colour
- Diffuse shader - a shader with lighting support, explained further in the below section

Diffuse Shader The shader used for the 3d geometry in the engine is named the diffuse shader. This shader includes ambient and diffuse lighting, where ambient lighting is from light that indirectly lights the object through bouncing off other objects, while diffuse lighting is from light that directly hits the surface of the object without bouncing. You can see a cube rendered with no lighting in figure 5, and a cube rendered with ambient and diffuse lighting in figure 6. Without the lighting shader, it is impossible to make out the separate faces of the cube, as they are all the same colour, so it looks like a 2d image. This is why it was required to create the diffuse shader, graphical fidelity was not an aim of this project at all, but some amount of lighting is required to be able to make out 3d shapes.

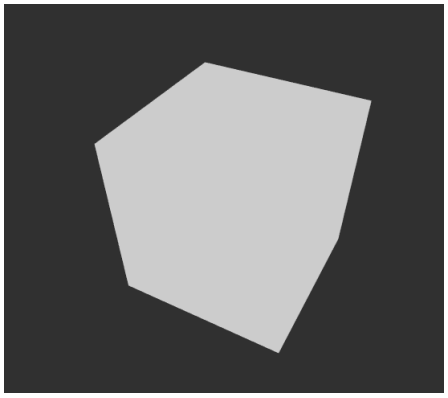

Figure 5: A cube rendered in a single color

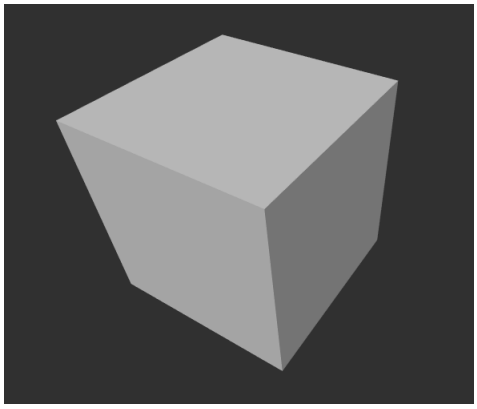

Figure 6: A cube rendered in with ambient and diffuse lighting

The calculations for both ambient and diffuse lighting is done in the fragment shader. The calculation for the color of a pixel for ambient lighting is very simple, $ambient = ambient\_strength * light\_color$. Where *ambient* is the output colour of the pixel as a colour vector of length 3 (where each component of the vector is the R, G and B level of the pixel), *ambient_strength* is the ambient light strength as a number between 0.0 and 1.0 (where 0.0 is no ambient light, and 1.0 is full brightness), and *light_color* is another colour vector allowing for different light colours.

The calculation for the colour of the pixel for diffuse lighting is more complicated than for ambient lighting. The pseudocode for this calculation is shown in figure 7. First, the current vertex normal (passed in by the vertex shader) is normalized, and then the dot product of the normal and the direction of the light is calculated and then clamped so that if the result is below 0, it is set to 0. The reason for this is that if the dot product is negative, this means the normal is facing away from the light and therefore is in complete darkness. This dot product signifies to what degree the current vertex is facing towards the light, the higher the dot product is, the higher the brightness of the pixel, because the pixel is facing towards the light. The lower the dot product, the less bright that pixel is, as that pixel is facing further

away from the light. This can be seen in figure 6, the top of the cube is lighter than the right side of the cube, because the light is on top of the cube and the top of the cube is facing it directly while the right side of the cube is facing at a 90° angle from the light.

Finally, the colour of the pixel is set to the results of the ambient and diffuse lighting stages added together and then multiplied by the desired object colour ($(ambient+diffuse)*tint\_color$). This results of these two lighting effects are shown in figure 6.

```
vec3 normal_dir = normalize(v_normal);
float dot = max(dot(normal_dir, light_dir), 0.0);
vec3 diffuse = dot * light_color;
```

Figure 7: The shader code to calculate the color of a pixel after diffuse lighting

3.5.6 Graphical User Interface

The engine supports multiple types of user interface components including buttons, coloured rectangles, textured rectangles and fonts. Coloured rectangles, textured rectangles and fonts are all drawn with method calls. The pseudocode definitions for these methods can be seen in figure 8. Images and fonts can be loaded through the asset system (explained in section 3.5.1) and then passed to these draw methods. Buttons, however, work differently, which is explained further below.

```
function gui_draw_colored_rect(vector4 color, uint32 x, uint32 y,
        uint32 width, uint32 height);

function gui_draw_image(image img, uint32 x, uint32 y,
        uint32 width, uint32 height);

function gui_draw_text(font fnt, string text, vector4 color,
        uint32 x, uint32 y, float scale);
```

Figure 8: Draw functions for colored rects, textured rects and fonts

Buttons are stored in a dynarray (explained in section 3.4) and entirely handled by the GUI system (drawing of the button, input handling). The button struct to hold button data is shown in table 4, allowing customization of the button location (x and y position, width and height), look (background image, hover background, icon image) and function by setting a callback that is invoked when the button is clicked. This makes GUI interfaces easier to create by handling normal functions like changing appearance on hover and running a function when clicked. As well as ease of use, it also allows the engine to prevent world interactions (like selecting units) when intending to interact with GUI elements. When

handling mouse functions like clicking and scrolling, a function can be called to check if the user interface has already handled this event (called gui_handled_click, which returns true if the GUI has already handled this click), in which case the click should be ignored.

Name	Data Type	Description
x	unsigned 32-bit integer	X position
y	unsigned 32-bit integer	Y position
width	unsigned 32-bit integer	Width
height	unsigned 32-bit integer	Height
visible	boolean	Is the button to be drawn or not
hovering	boolean	Is the user hovering over the button or not
icon_img	image	The image to show over the button
bg_image	image	Button background image
hover_bg_img	image	Button hover background image (shown when hovering the mouse over the button)
click_callback	function	A function called when the user clicks the button
hover_callback	function	A function called when the user hover or un-hovers the button

Table 4: The data model for the button struct

3.5.7 Mouse Raycast

For a player to interact with the 3d world using their mouse, the engine supports a method to project the mouse position on the 2d screen into the 3d world called a raycast. The functionality is used in the game for selecting units in the world using the mouse (explained further in section 3.6.4). A diagram demonstrating this method is shown in figure 9, and a pseudocode implementation of the calculation in figure 10. The vector math works by taking the 2d screen space coordinates of the mouse (normalized between -1.0 to 1.0) and transforms it into world space coordinates using the view and projection matrices.

For the camera, the view matrix is multiplied with a world space vector to convert it into camera space, and then the resulting vector is then multiplied again by the projection matrix to convert the vector from camera space to screen space. Because of this, to transform the screen space position of the mouse into world space, we can do the opposite operation as the camera. This is what the pseudocode in figure 10 is performing. First creating a vector in screen space (and points it out from the front of the camera, which is why the z-axis is set to 1), then multiplying the resulting ray_clip by the inverse of the projection_matrix which defined by the camera, and then multiplying it again by the inverse of the view_matrix. The

ray is then normalized to remove the magnitude of the vector. The resulting ray is now in world space and can be checked for collisions with various objects in the world to see if the mouse is over the object.

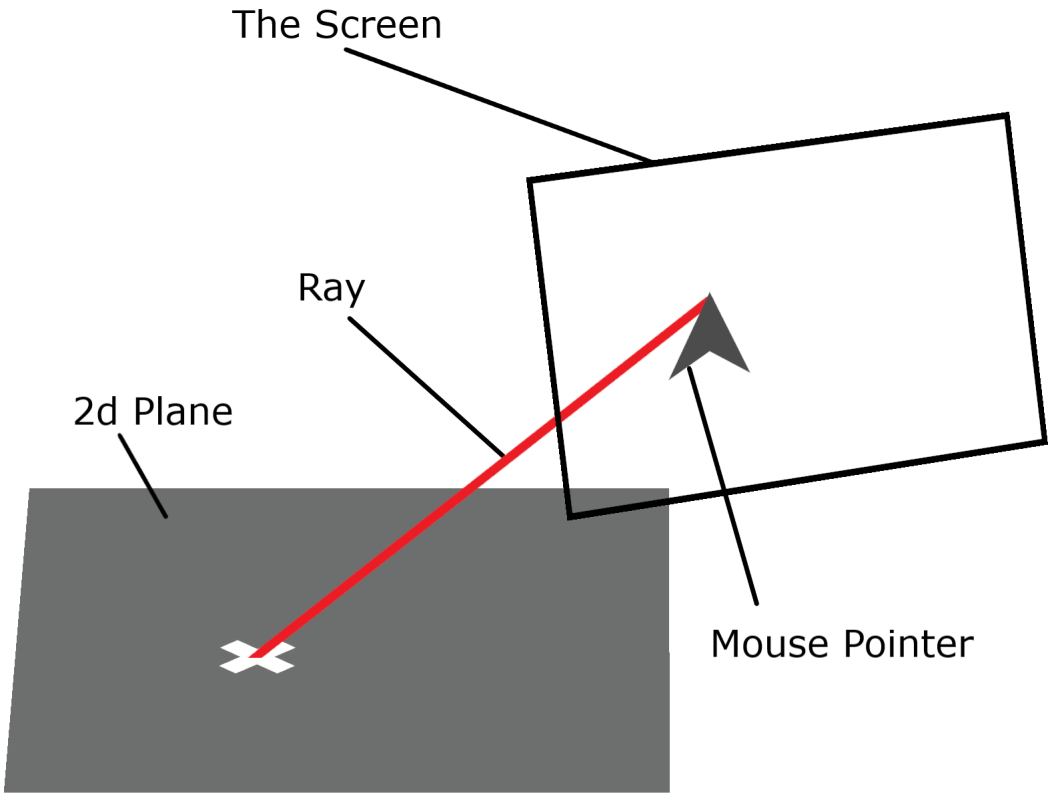

Figure 9: A diagram visualizing how a 2d screen coordinate of the mouse is transformed into a ray that is then intersected with a 2d plane

```
vector ray_clip = vector(mouse_x, mouse_y, 1, 1)

vector ray_eye = inverse(projection_matrix) * ray_clip
ray_eye.z = -1
ray_eye.w = 0

vector ray = normalize(inverse(view_matrix) * ray_eye)
```

Figure 10: The shader code to calculate the color of a pixel after diffuse lighting

3.6 Gameplay

3.6.1 Rules

The game is a turn-based strategy game, meaning each player has a set of units (explained in section 3.6.5) and takes turns to take actions with the units. Once one team is finished taking unit actions, the turn is passed over to the other team for them to take their actions, this continues until one team has no units remaining. The team with no units remaining loses.

Each player can gain a tactical advantage by proper use of cover mechanics (explained in section 3.6.8), proper selection of actions (when to move or shoot, explained further in section 3.6.6) and general positioning tactics.

3.6.2 Map

The map is made up of squares, referred to as "blocks", the map can be of any size but for most of my testing the map used was of sizes 64x64 and 128x96 as these seemed large enough to have an enjoyable game while not taking too long to scale the map with a unit. To make the game more replayable, and to assist in training the AI to learn how to play the game for different map styles, a new map is randomly generated for each new game (explained further in section 3.6.3).

The map is stored in memory as an array of "blocks". A block is made up of a block type (one of BLOCK_TYPE_NOTHING which is a blank block, BLOCK_TYPE_COVER which is cover and BLOCK_TYPE_ENTITY which is an entity). As well as the type, there is also some data which is either height of the cover (explained in section 3.6.8) or a pointer to the entity that is at that block (explained in section 3.6.5). They are stored in z-order to avoid cache misses when accessing blocks that are close together (this is explained further in section 3.8).

3.6.3 Map Generation

The map generator works in a hierarchy, generating the largest structures first, and then filling the structures with smaller elements. The first element of the map to be generated is the road. The road spans from the top right of the map and ends on the left side of the map. The way the road is generated is by moving in a direction for a random amount of time, and then changing direction, and moving in that direction for a random amount of time, before randomly changing direction again. This continues until the road hits the edge of the map, and it is forced to end so that the road is facing the left corner of the map and not the bottom

simply because it improved the maps visual look. There is also an editable constraint that specifies the minimum length of a road segment. Otherwise short road segments of length one could be generated and look unnatural. The final map generation is shown in figure 11.

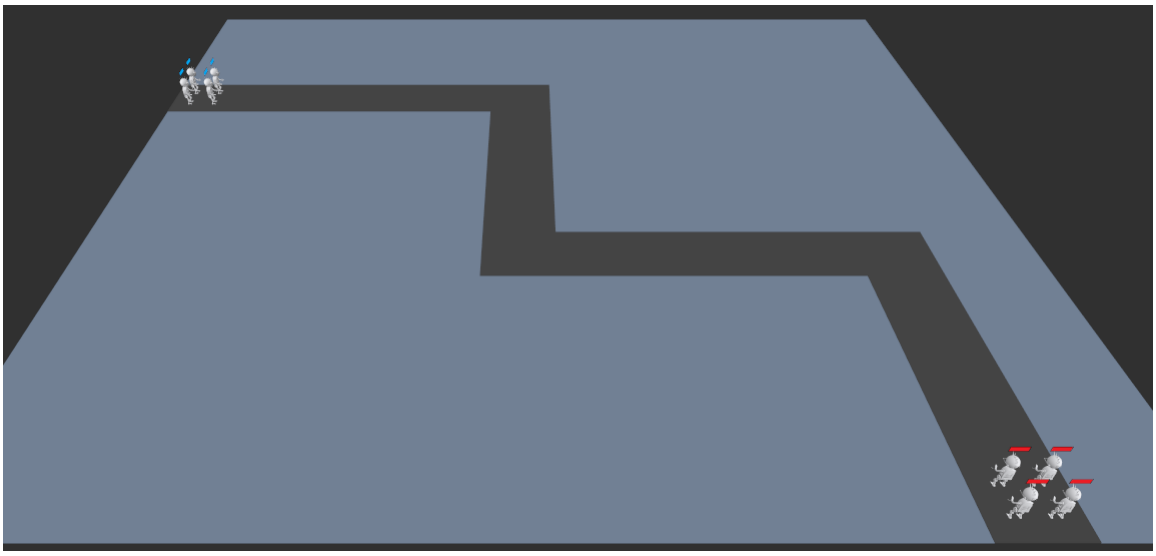

Figure 11: A map generated with just a road winding from the top right of the map to the left

After the road has been generated, the remaining map area is split up based on the road segments, so that it is split into distinct rectangles that do not overlap the road. A visualization of this on a map is seen in figure 12

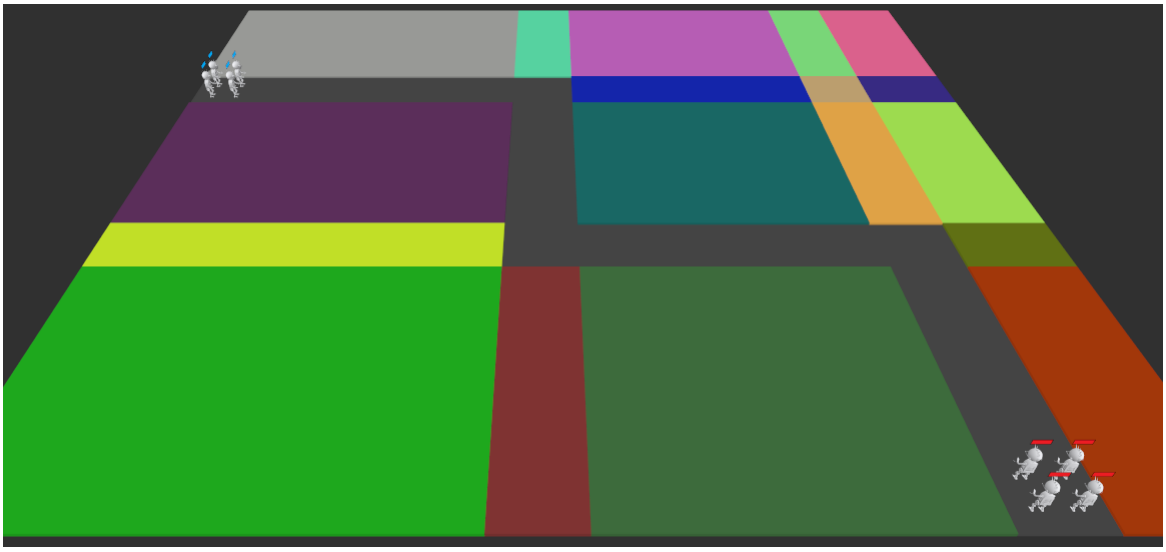

Figure 12: A map generated with segments displayed in random colors

However, because these segments are small and seemingly random, they are simplified

into larger and more meaningful rectangles, but that is still split by the road. The algorithm for simplifying the segments is simple but achieves good enough results for the map generator. First, the largest segment is chosen, and then it is compared with surrounding segments to see if it can be merged to expand itself. If it is expanded until it cannot be expanded any further. Then, the next largest segment is chosen and checked if it can be merged, it is checked that after this merge if the larger segments before this can now be merged, and they are if it is possible. This process continues until all segments are unable to be merged. An image of a generated map after this process is shown in figure 13.

Figure 13: A map generated with simplified segments displayed in random colors

These segments are then used to generate structures within them. Buildings generate in map segments larger than 20x20, and that is no longer (both of these parameters can be changed to achieve different resulting maps). The reason for the 20x20 restriction is to stop buildings of too small size generating, and the width to length ratio restriction is to stop abnormally "thin" buildings (wider than they are long or vice-versa) which end up looking unnatural.

Buildings start as rectangles of random sizes placed at random positions in map segments that meet these restrictions. The internal walls are inserted inside the building, and then as more internal walls are inserted, small rooms within the buildings form. Finally, to give units access to the buildings for cover, walls are removed from the external building wall to give building access and some internal walls to give assess to the interior rooms. An image of the final generated buildings is shown in figure 14.

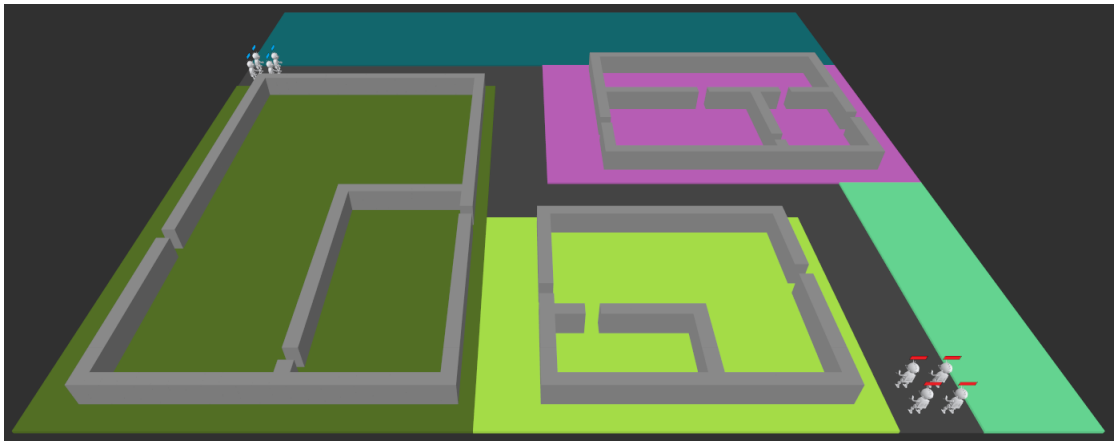

Figure 14: A map generated with buildings in segments that fit the criteria, and the simplified segments shown in random colors

As final touches, to create maps with lots of cover and chance for good tactical encounters, "cars" are generated along the road to provide more cover and to stop a direct line of sight down road segments and to add more interest to the map. The enemy units are then placed at the start of the road, and the friendly units placed at the end of the road. A few fully generated maps are shown in figures 15, 16 and 17. Although the map size and various building restrictions placed on the generator, the generator is still able to generate sufficiently interesting and differing maps each time, and aids in testing the AI on different maps.

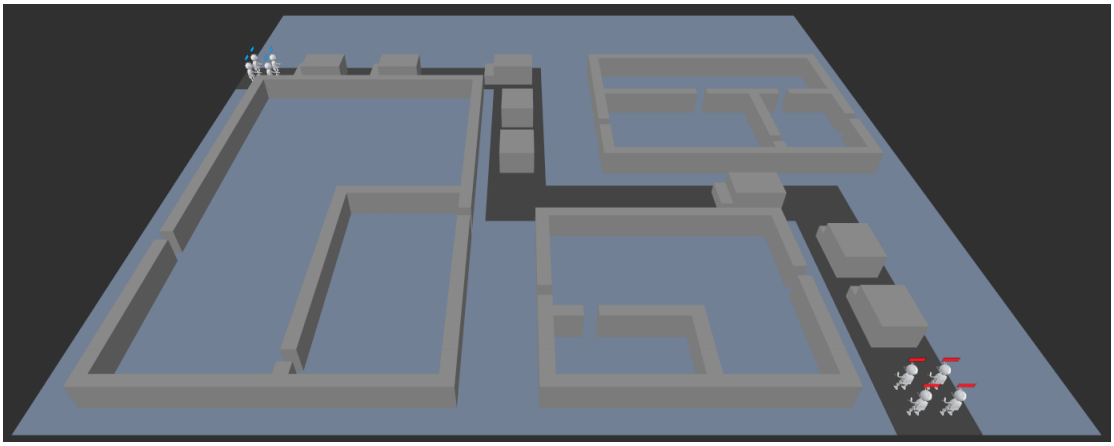

Figure 15: Example 1 of a fully generated map

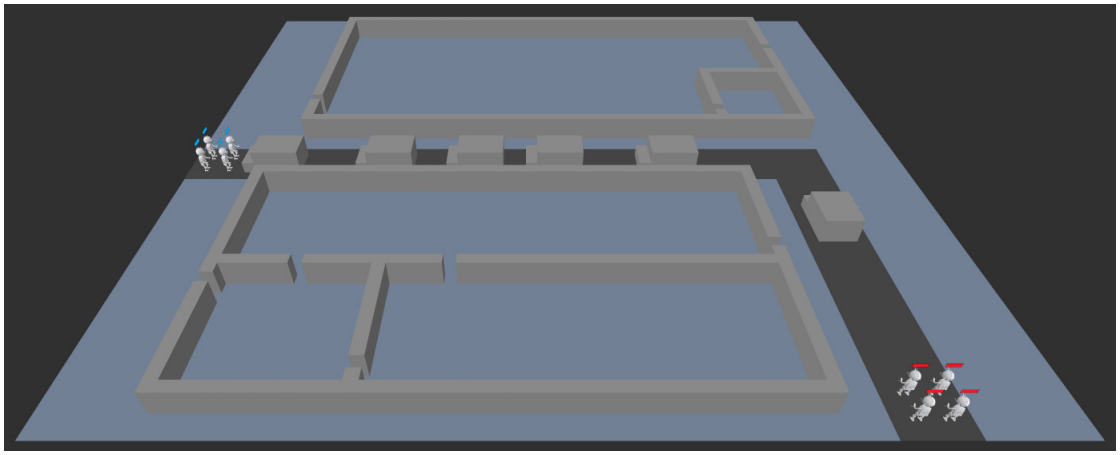

Figure 16: Example 2 of a fully generated map

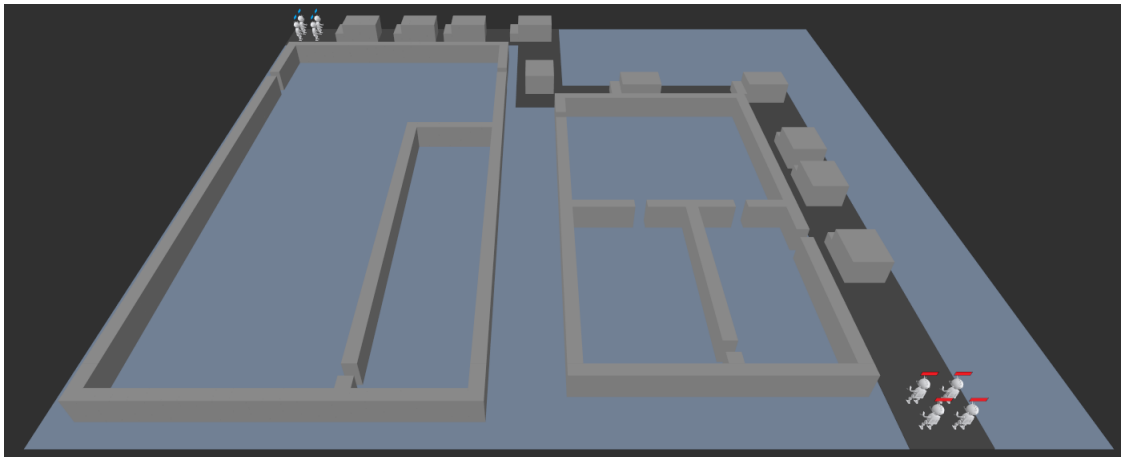

Figure 17: Example 3 of a fully generated map

3.6.4 Mouse Picking

To select units and select a location on the map to move to with the mouse, the mouse is turned into a ray (using a feature in the engine called a raycast, explained in section 3.5.7) and then this ray is intersected with the 2d plane of the map terrain. By intersecting the ray with the 2d plane, the point at which the intersection takes place can be found, and therefore a 2d position at which the mouse is pointing at the map. The reason for doing this instead of other collision methods is because it is a lot faster and simpler than say a mesh collision check. Although a 2d plane collision does not support terrain of different heights, this is currently not used in the game, but it does mean that if a variable height map is added to the game in future, a completely different collision algorithm would be required.

A pseudocode algorithm for getting the point of intersection of a ray on a 2d plane. is shown in figure 18. This implementation is based on an excellent worked example from

MIT's OpenCourseware [13] which explains the mathematical theory required to write the code.

```
// the angle between the plane and the ray
dot = dot_product(plane_orientation, ray_direction)

if(abs(dot) > 0) // stop divide by 0
{
    // distance along the ray at which it intersects with the plane
    t = dot_product(plane_origin - ray_origin,
                plane_orientation) / dot

    if (t > 0) // if there is an intersection
    {
        // get the point t distance along ray_direction
        plane_intersection = ray_direction * t

        // offset by the ray origin
        plane_intersection = plane_intersection + ray_origin

        // there was an intersection at this point
        return plane_intersection
    }
}

// there is no intersection
return none;
```

Figure 18: Psudocode implementation of an algorithm to get the location of intersection of a ray and plane

3.6.5 Units

Units (referred to as "entities" in the gameplay code) are the soldiers that each team has to defeat the other, each unit's data is stored in the "entity" struct, the data model for the entity struct can be seen in table 5. An image of how the units look in the game can be seen in figure 19, the model for which is a royalty-free 3d model [6]. Each unit starts with has ten health and 2 action points. At the start of each turn, the last team's units have their action points set to 0, and the current team's units have their action points reset to max_ap (2, in this case), For each action point, the unit can take one action (shoot, move, do nothing), meaning per turn each unit can take two actions. Each unit belongs to a team, identified by the team field in the struct which is set either to TEAM_FRIENDLY if it is on the players' team, or TEAM_ENEMY if on the AI team. Friendly units have a blue health-bar rendered above their heads, while enemies health-bars are rendered red. When a units health falls to 0 or less, the "dead" field is set to true in the entity struct, and the entity will be removed from the list of entities at the start of the next frame. This means that any code that needs to act on the death of the entity has the end of the frame to use the data before it is freed.

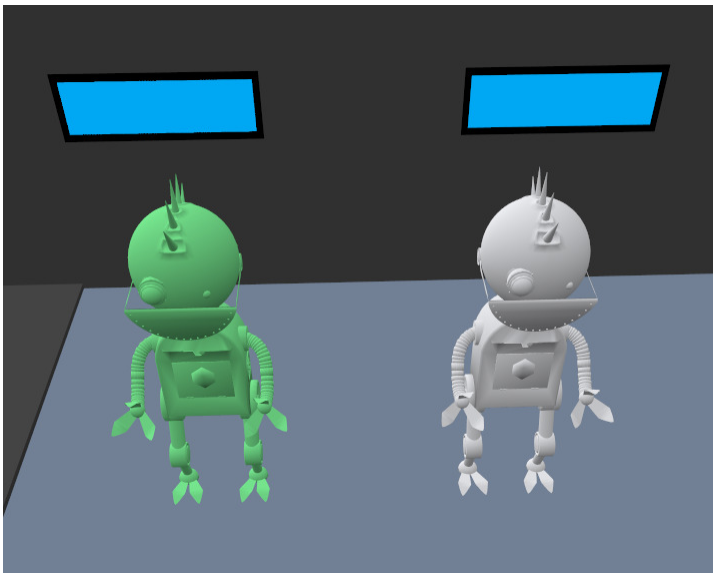

Figure 19: Two friendly units, the green unit showing that this is the current unit the user has selected

Name	Data Type	Description
id	unsigned 32-bit integer	A unique number identifying the unit
pos	vector3	The x, y and z positions of this unit
mesh	mesh	The mesh that will be drawn at the unit location
health	signed 32-bit integer	The current health of the unit
max_health	signed 32-bit integer	The maximum health of the unit
ap	signed 32-bit integer	The current action points the unit has
max_ap	signed 32-bit integer	The maximum action points the unit has (ap will be set to this value at the start of each turn)
team	enum (TEAM_FRIENDLY or TEAM_ENEMY)	The team the unit belongs to (currently friendly or enemy)

Table 5: The data model for the entity (unit) struct

3.6.6 Unit Actions

Units have several actions available to them; these actions are explained below. The user selects which action it wants to perform from the action bar, which is explained in section 3.6.7. Each of these actions uses exactly 1 action point, regardless of how much is achieved (even if you miss, or only move one block, for example).

Move Action allows the unit to move within a ten block radius since two action can be taken per turn, one unit can move a maximum of 16 blocks per turn. The movable blocks are determined by a breadth-first search of nearby blocks until the maximum distance of 10 is reached.

Shoot Action allows units to have a chance to shoot units within infinite blocks, depending on if they have a line of sight and how much cover the target is in and how far away they are (LOS and shot chance calculation is explained in section 3.6.8), which does 6 damage. Since units can take 2 actions per turn, units can do up to 12 damage with this action per turn.

Do Nothing Action sets unit action points to 0, would be used when a unit does not want to do anything this turn (if it is in a good position but has no enemies to shoot, for example).

3.6.7 Action-bar

The action bar is the primary way that the user interacts with the game. When selecting a unit with the mouse, a bar shows up at the bottom of the screen. This can be seen in figure 20 with the currently selected unit shown in green. The bar has a button for each action the unit can take: move, shoot and do nothing (described further in section 3.6.6). Above the action bar is space reserved for a message that can pop up on the screen for a set amount of time, called the action-bar message, an example of an action-bar message is shown in figure 21 and used to inform the user of something.

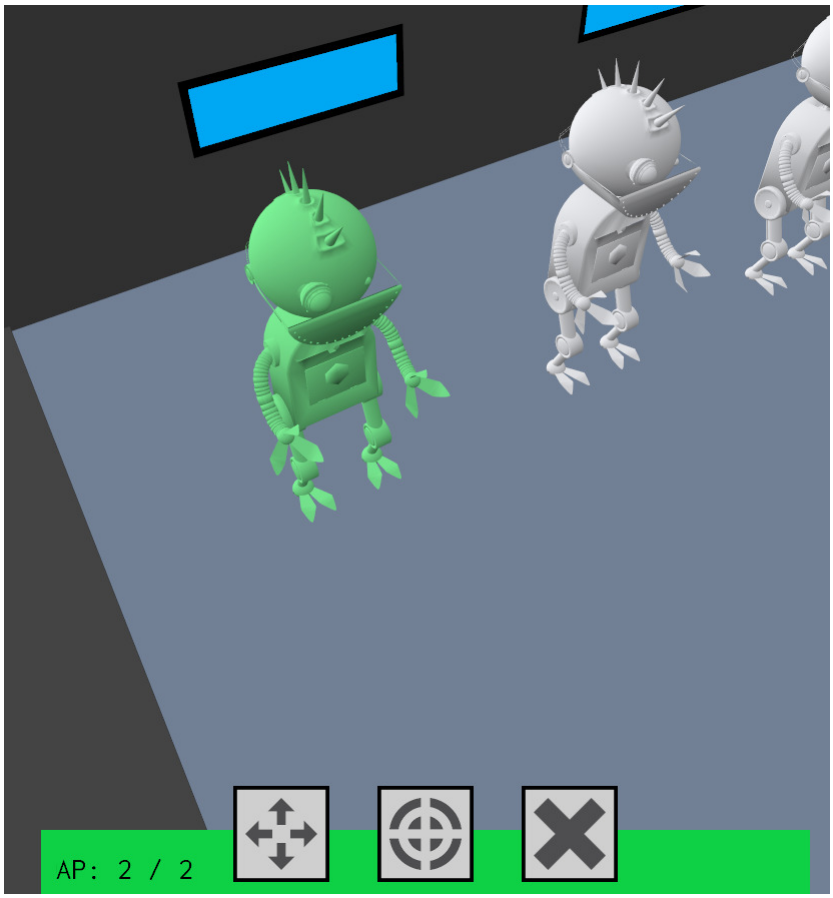

Figure 20: The action-bar for the currently selected unit

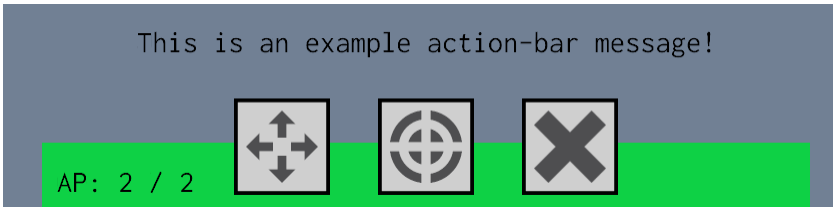

Figure 21: The action-bar message, shown just above the action bar

When clicking the first button on the action bar (with an image of 4 arrows pointing in 4 orthogonal directions), move mode is activated. In move mode, a box at the top of the screen with the text "Move Mode" tells the user the current mode, and the possible moveable locations are shown to the user in a blue tint. This can be seen in figure 22. The user can select one of the tinted locations to move there. If the user clicks on a location that is out of range or unreachable from the current position (units cannot move through walls), then an action-bar message shows stating "Invalid move position".

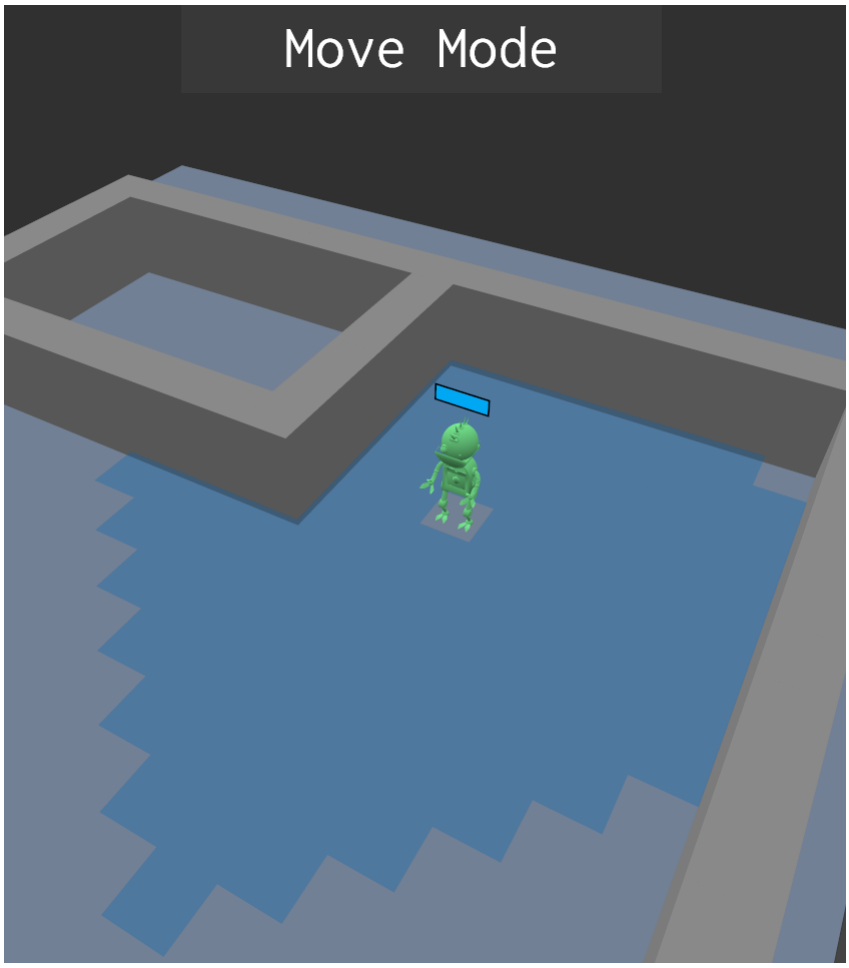

Figure 22: The user interface after entering move mode

The second button on the action bar (with a shooting target as an image) activates shoot mode, which is shown in figure 23. If there are no targets around, when clicking the shoot button, an action-bar message pops up telling the user that there are "No targets to shoot". If there are targets nearby, a button for each unit in line of sight is displayed above the action bar, containing the percentage chance that the shot has to hit. When hovering over the button, the target enemy is shown in red, and when clicking the button, the shot is taken. The percentage chance to hit is based on if the enemy is in cover or not, the calculation for which is explained in section 3.6.8.

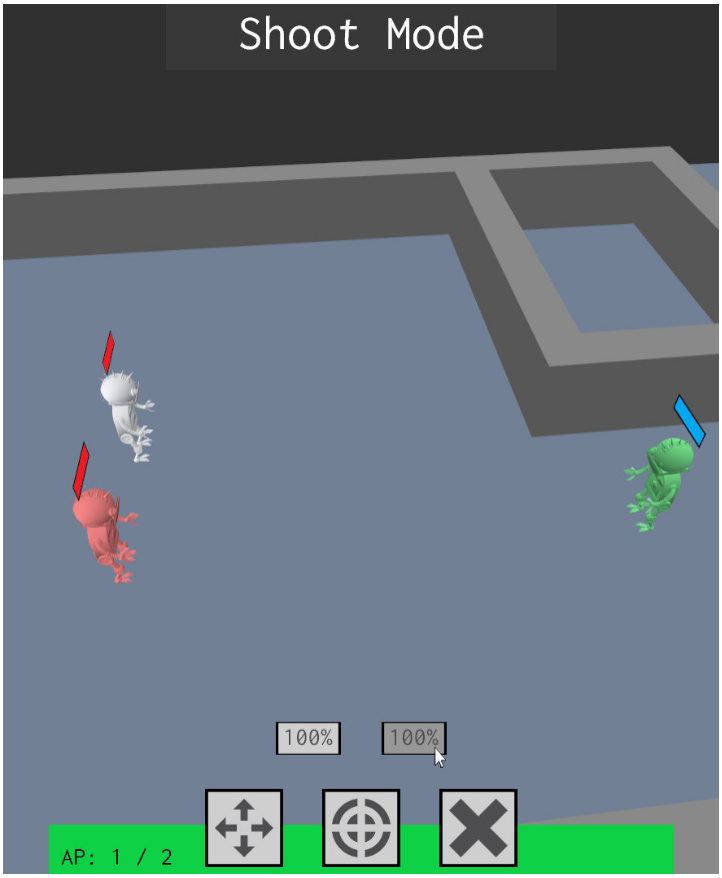

Figure 23: The user interface after entering shoot mode

If the user clicks the "do nothing" action button (shown as a button with a red cross), the currently selected unit has the action points set to 0, so it cannot take any more actions this turn. The reason for wanting to use this is if the unit is in the desired position and other action points are remaining since the other teams turn cannot start without all units having used up their action points.

The combat log is shown in the bottom left of the screen and can be seen in figure 24. The combat log has a list of all damage done and units killed throughout the game, the most recent event is at the bottom of the combat log, and as more events happen, the earlier events move up and eventually out of view.

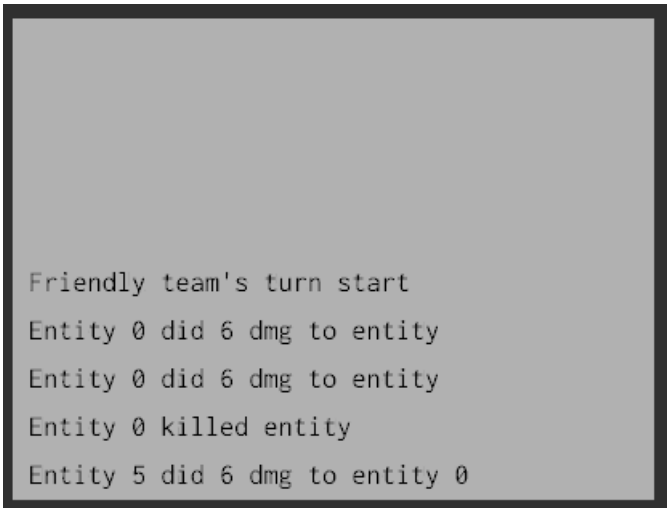

Figure 24: The combat log, shown in the bottom left of the screen

3.6.8 Cover and Shot Chance

Cover is the strategic element of the map that a player can use to gain an advantage over other players. It is shown on the map as a 3d cube. Cover can either entirely block line of sight, therefore stopping another player taking shots at the unit, or partially blocking LOS which reduces the chance that a shot taken at the unit is likely to hit. A unit is considered to be fully in cover relative to another unit if the unit cannot see any part of the other unit from their position because it is all behind cover. Similarly, a unit is partially behind cover if part of the unit is blocked by cover, but is still visible.

The algorithm for testing if there is LOS between two units is where multiple rays are calculated between the shooting and targeting unit at different positions on the shooting and target unit. If at least one of these rays hits the target without colliding with cover, like in figure 25, then the unit has at least partial LOS to the target. If all of the rays hit cover before hitting the larget, the unit has no LOS to the target, like in figure 26.

If there is some line of sight between the units, the chance that the unit can hit the target is calculated based on how much of the other unit they can see. The amount of vision is calculated by first finding an adjacent (within one block, including diagonals) piece of cover to the target which is closest to the shooter. If there is no cover adjacent to the target, this means they are not in partial cover, and the shooter has full LOS. If there is a piece of cover, then the chance to hit is $min(1, 2 * dot(cover\_to\_target, cover\_to\_shooter))$ where cover_to_target is the normalized vector of the closest cover position to the target position, and cover_to_shooter is the normalized vector of the closest cover position to the shooter position. If the dot product is negative, this means the cover is behind the target, and the shooter, therefore, has full LOS. This calculation gives the extent to which the shooter is peaking around the cover to see the target.

As well as this, the probability that a shot is hit is reduced as the distance between the two units increases. The calculation for for the chance to hit a shot is

$$los * min(1, \sqrt{map\_width^2 + map\_length^2}/6)/distance(shooter\_pos, target\_pos)$$

where *los* is the line of sight amount calculation explained above and the distance function is Euclidean distance. This formula caps the chance to hit at 100%, and if the two units are close enough and have full LOS (not partial) then the chance will be 100%. If the two units are at the furthest diagonals of the map, then the chance to hit will be 0% (Although it is highly unlikely to ever be in this situation). If the units are map_width distance apart, the chance to hit will be 25% if they have full LOS.

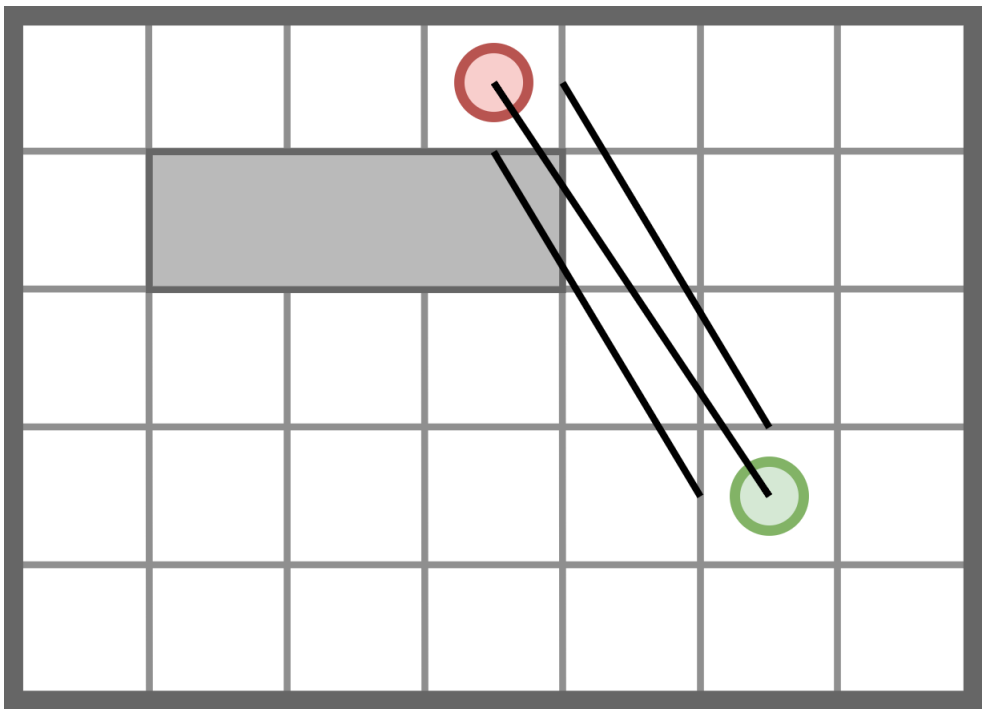

Figure 25: An illistration of partial line of sight calculation

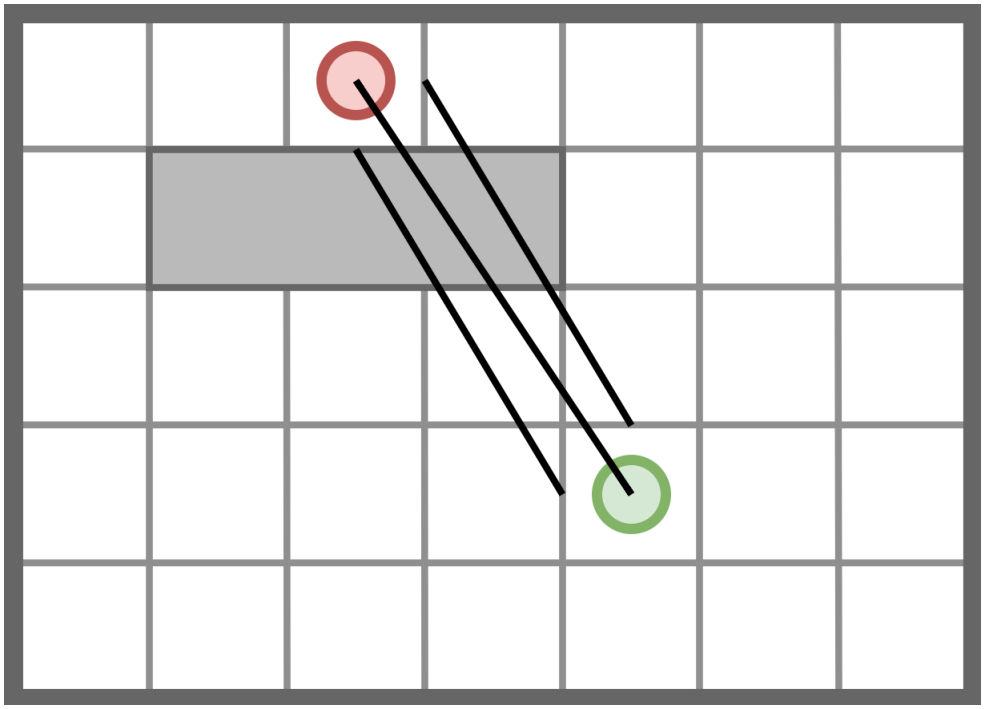

Figure 26: An illistration of no line of sight calculation

3.7 Artificial Intelligence

The game AI uses a decision rule system called Minimax (section 3.7.1), this is similar to the approach that early chess AI players used, as the game shares similar base principles of chess such as pieces (units), a grid-based movement system (the map) and that it is a zero-sum game. In order to get good results using minimax, a number of other algorithms must be used to make a performant AI (does not take too long to decide, since the game is run in real-time) and to get an AI that can challenge human players.disputingrithms are alpha-beta pruning (section 3.7.2) and Monte Carlo tree search (section 3.7.3).

The reason for using Minimax instead of another AI method such as deep learning or a simple rule-based system was because it would not be viable within the time constraints to make the engine, the game as well as developing and training an AI using a method such as deep reinforcement learning. While on the other hand, using a simple rule-based system, in the authors' opinion, would not have produced a very capable AI that has enough strategic understanding to be challenging.

3.7.1 Minimax

The base implementation of the AI is a simple Minimax algorithm. The Minimax algorithm works for two-player games where the rules of the game mean that it is a zero-sum game. A zero-sum game is when there is a gain for one player is a loss for the other, and vice-versa. This is why Minimax should be appropriate for this game, since the game is a two player game, and is a zero-sum game. For example, when one players unit shoots another players units, the other players unit health decreases which ends with player 1 in a better position and player 2 in a worse position.

The most simple Minimax algorithm implementation works by searching a tree made up of all possible directions the game could go based on the actions each player takes. If Minimax were only to be used at 1 level of deepness, it would find the best action for the current player by making each move and using an evaluation function on the current state. For example, Minimax takes each possible move action available to the unit and after acting, use the evaluation function on the current game-state (further explained in section 3.7.4) and compare the actions' evaluation to see which action is likely to be the best move. The evaluation function gives a numeric value based on the current state of the game so that it can be compared to other game-states, and the one with the higher evaluation is considered more "advantageous" state of being in for the acting player.

Although only searching one layer of the minimax tree is fast, it does not give the best results because it does not evaluate how good a move is going to be later in the game, only how good the move is right now. Minimax can resolve this by searching further in the tree. This means that first a friendly unit action is performed, and then the AI picks

the best possible enemy move (i.e. the enemy action that has the lowest evaluation for the friendly player), and then the best possible friendly move (with the highest evaluation for the friendly player), and so on and so forth. This is where the Minimax algorithm gets its name, as the algorithm picks the action with the maximum evaluation value for the friendly player and then the action with the minimum evaluation value for the enemy. This occurs for each possible action, as deep as specified or until the end state of the game is reached. By searching deeper into the tree, the algorithm is more likely to pick the actions that will lead to the AI player winning the game, as it is looking forward and checking that the action is not only advantageous now, but also leads to a game state that is advantageous in the context of the whole game. A diagram of the base Minimax algorithm tree is shown in figure 27.

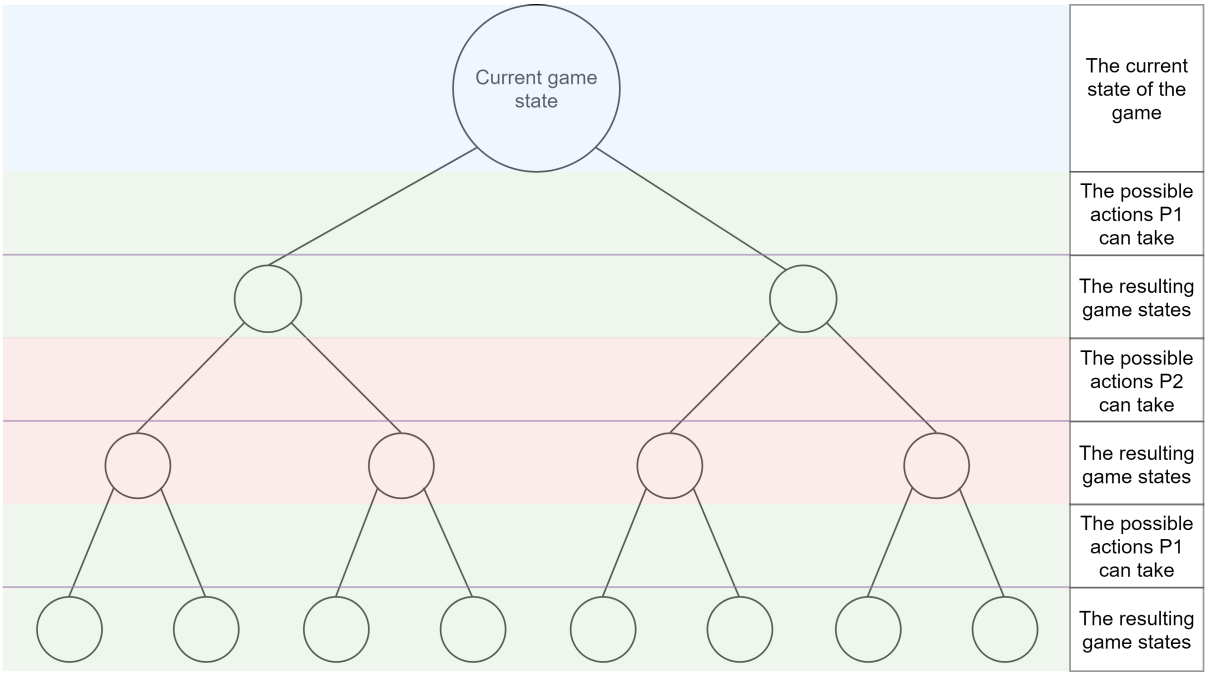

Figure 27: A diagram showing a Minimax tree of a game with 2 players (P1 in green and P2 in red) who can each take 2 actions at any point of the game

When searching deeper into the tree, the search-time increases exponentially. Because of this, various methods can be used to reduce the number of nodes that need to be evaluated at each level. Two popular methods are Alpha-Beta Pruning (explained in section 3.7.2) and Monte Carlo Tree Search (explained in section 3.7.3). Evaluating only the most promising nodes at each level means that Minimax can explore deeper into the tree, and therefore is more likely to provide a more accurate prediction of the best current move for the unit.

3.7.2 Alpha-Beta Pruning

To improve the performance of Minimax, an algorithm called alpha-beta pruning was used. This algorithm works by reducing the number of nodes that Minimax has to search, which in turn reduces the runtime of the algorithm for the same sized tree. The alpha-beta pruning algorithm stops evaluating further actions in the current branch when an action has been found that proves the action to be less advantageous than a previous action. This means that if the actions with the highest evaluation function value are evaluated first in the tree, more nodes do not need to be examined. However, it can be more expensive and a difficult problem in itself to order the actions by how likely they are to be promising, and because of the time limits of the project, this is not implemented. Even without the ordering of actions, alpha-beta pruning still gives a significant performance boost to Minimax and is worth implementing because of this speed-up.

The algorithm works by using values alpha and beta, where alpha is the current minimum value the AI player can get, and beta is the maximum score the opposition can get. Alpha is initialized to minus infinity (the worst score the AI player could get), and beta is initialized to positive infinity (the worst score the opposing player could get). If beta becomes less than or equal to alpha, no more children of the node need to be considered because this means that either the maximizing player does not pick this node because the AI player already has a better action it could pick, or the opposing player already has a worse action (for the AI player) to pick.

3.7.3 Monte Carlo Tree Search

Monte Carlo tree search is another optimization that can be used with Minimax to get deeper tree exploration. The algorithm uses heuristics along with randomness to reduce the number of nodes that need to be evaluated. The number of shot targets that need to be evaluated at any node is only equal to the number of units that are alive, which is a fairly low number. However, when evaluating move targets, all blocks within ten blocks of the entity are considered. If there are no blocked move positions, this can be up to 220 possible move positions per entity ($11 * 10 * 2$) which drastically slows down minimax as this is a large branching factor. Presuming all possible move positions are available to the entity, at the first level of the search tree there would be 220 nodes, and then at the second $220^2 = 48400$, the 3rd $220^3 = 10648000$, and so on.

The move positions can be randomly sampled to reduce the branching factor and therefore the speed of the search. If 50% of the possible move positions are sampled, the average number of move positions evaluated is reduced to about half (110 blocks), and therefore the number of nodes at the 3rd level of the search tree would be reduced by a factor of 8. This is massively advantageous and could make Minimax search further into the tree in the same amount of time, or reduce the runtime required for the AI to search to a specified level.

However, this may end up stopping the AI from picking the best possible move positions. To reduce the chance that a good move position is not evaluated, a heuristic can be applied to increase the chance that a move position is evaluated in certain situations. Since moving to a cover position is very advantageous and preferred to not being in cover in almost all cases, the heuristic can increase the chance that a position that is in cover is evaluated while still decreasing the number of total nodes to be evaluated. With the heuristic, move positions within one block of cover have a 70% chance of being evaluated and positions not adjacent to cover have a 10% chance of being evaluated.

3.7.4 Evaluation Function

The evaluation function is used to give a numeric value of how advantageous a current game state is for a player. The larger the numeric value of the evaluation, the more advantageous the game-state is predicted to be for the player. This evaluation function is used in Minimax (explained in section 3.7.1) in order to compare game-states after taking different actions, and therefore pick the best action based on the action that is most likely to achieve the largest possible evaluation function value for the player (and therefore, most likely to be an advantageous position). The evaluation function is split into different elements, as explained below. Each element is hand weighted, so the effect it has on the final evaluation of the state can be changed and tuned to give the most accurate prediction. This tuning procedure is explained further in section 4.2.3. Below are the different hand weighted elements the evaluation function uses to predict how advantageous a game state is:

Shot Chance The chance that we can shoot our enemy and the chance that they can shoot us.

Health How much heath our units and enemy units have, as well as how many dead friendly units we have and how many dead enemy units they have. This attempts to encourage the AI to stay alive, as well as encourage the units to kill enemy units.

Distance to Enemy How far is each unit from the closest enemy. There is a cut-off of 10 blocks, so the AI does not get rewarded for moving closer than this to stop the AI walking right in front of the enemy which is a behaviour discovered early on. This tries to make the AI move toward their enemy, to engage in a fight. The weight of this determines how defensive the AI player acts. Without this weight, the AI is likely to find a place where all units are as covered as possible and wait there.

Cover How much in cover are our units from each direction. If there is an enemy in the cover direction, extra weight is added to prefer cover from the direction of the enemy.

3.8 Optimizing the Evaluation Function

The evaluation function is run many times during an AI turn. It is run every time a test action is taken to see how optimal the move is. Improving the performance of this function not only decreases the time that the user has to wait between the end of their turn and the start of the AI turn, but also increases the number of moves that can be evaluated by the AI. By evaluating more moves, the AI can perform better during gameplay.

VTune (Intel VTune Amplifier) is an application from Intel to profile the performance of x86 applications. This tool was used on the game while running the AI turn in order to identify what parts of the code are taking up the most CPU time during an AI turn, and therefore optimize these pieces of code in order to improve run-time. VTune also supports memory profiling, which identifies parts of the code that are memory bound, which is where the CPU is not being fully utilized because it is waiting for data to be retrieved from memory (for example, in a cache miss).

Line of Sight Cache After profiling a few times using VTune, it identified that a large proportion of the CPU time was taken up running LOS calculations. In order to see if the same LOS positions were being evaluated multiple times, the AI was rerun but would log each position it checked for LOS on, and show a list of the positions that were evaluated multiple times. It turned out that the starting and ending position of each entity was checked for LOS thousands of times. Since the result of this calculation is always the same, the result can be cached and retrieved from the cache instead of recalculating thousands of times. A hash table was used with the key as the x and y positions of the starting and ending locations for the LOS lookup. The value at each key was the result of the LOS calculation. The cache is built before the start of the AI turn by looking up the LOS between each units starting position, the result is then put in the hash table for lookup. Without the LOS cache, ten simulated turns of AI actions looking three turns ahead took around 130 seconds and using the cache this was reduced to about 66 seconds (about half the time).

Z-Order Curve Still, after using the cache optimization, much of the CPU time was spent on LOS calculations. Memory profiling on the code using the VTune tool showed that much CPU time was spent waiting for the current cover status of a block (is there cover there or not) to be retrieved from memory. This was because of a cache miss. A cache miss is where the CPU cache, which is much faster than main memory, is searched and the data is not in the cache, so the data has to be retrieved from main memory. This is why accessing continuous memory is much faster than fragmented memory, because chunks of the main memory are copied into the cache and can be accessed much faster there when reading the next piece of data instead of going back to main memory.

The problem with cover data is that it is stored in row major order, where each row is the cover along the x-axis. This means that (0,0) is stored at index 0, (1,0) is stored at

index 1 and so on. This continuous memory uses the cache effectively and performs well. However, if searching for cover along the other axis, the cache is not used effectively, and so performance is much worse. For example, position (0,0) is at index 0, position (0,1) would be at index 63 if the map is 64 blocks along the x-axis. These two memory locations are not close to each other, and so when a read is cached for (0,0), it is likely that another main memory read will be required for (0,1) which is very slow.

To improve performance when searching for cover along any axis other than the x-axis, the ordering of the data must be changed. In databases, a common way of improving cache performance on location data is by using a z-order curve, also known as Morton codes. A z-order curve is a function that maps multi-dimensional data (in this case, 2d coordinates) into a one-dimensional index. However, the advantage of using z-order curves is that the index preserves locality, meaning that nearby data is more likely to be stored nearby in memory than the regular $index = x + y * width$ method. An illustration of the z-order curve is shown in figure 28. Since the efficient calculation of Morton codes is relatively complicated, a library called libmorton was used to encode and decode the indices.

After using a z-order curve for the index of cover blocks, the cache utilization was improved during LOS calculations, and therefore the performance of lookups along another axis other than x was improved. Although this does decrease the performance of searching for LOS along the x-axis, it increases the performance of all other searches and makes the AI turns of a more consistent length.

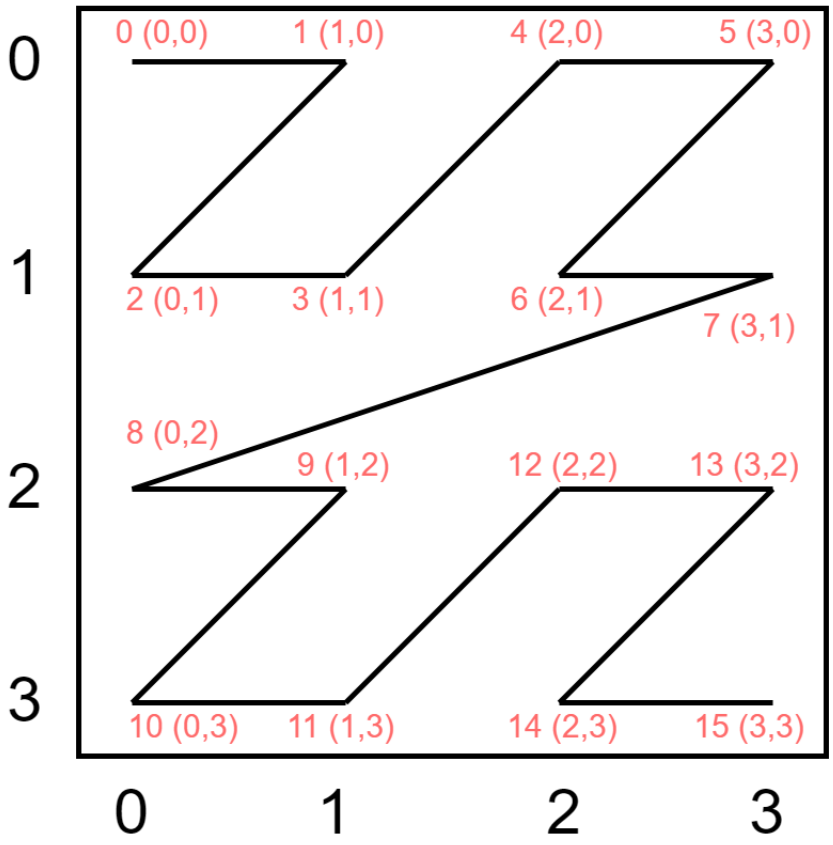

Figure 28: A diagram showing how a z-order curve maps a 2d 4x4 box into a spatial index

3.8.1 Temporary Game States

To implement Minimax it is required that several actions can be performed and then this state evaluated. Since the Minimax search is performed on a tree of actions, this functionality can be implemented by using the stack. Before Minimax searches further into the tree, it first creates some data called "undo_data". This undo data holds all the information needed to undo an action, and then the action is performed, and the algorithm searches further into the tree. When the result of a search is completed until the specified depth, the algorithm walks back on its actions, poping undo information off the stack and performing an undo using this information. For example, if the action was shooting an entity for five damage, and then the next action shooting an entity for another four damage, there are two pieces of undo data that when used healing the entity first by four health points and then again by four health points. This means that when the search concludes, and all the actions have been performed, evaluated, and undone, the map returns to the state at the start.

4 Results and Evaluation

4.1 Engine

For testing the performance and memory usage of the engine, a tool called MSI Afterburner was used on various hardware using a Windows 64-bit release mode build of the game. All tests are run for 5 minutes and involve playing the game using the same steps on the same generated map to get the most consistent results possible. The results of these tests are shown in table 6. The game performs over 60 fps on all of the desktop computers and manages to stay above 30 at all times on the laptop with integrated graphics. This shows that the engine is capable of performing well on lots of modern systems, even older productivity laptops not intended for gaming. As for the memory usage, the game uses around 150mb in 32-bit builds, and about 400mb in 64-bit builds. The 150mb 32-bit version is reasonably impressive, and the 64-bit builds 400mb is acceptable. The reason for the increase in memory usage of the 64-bit build is likely due to the doubling of the size of pointers from 4 bytes to 8 bytes.

Name	Processor	Graphics Card	Avg FPS	Min FPS	Max FPS
2018 PC	i7-8086k, 5.00GHz	GTX 1080ti	1847	1724	1897
2014 PC	i5-4690k, 3.50GHz	GTX 770	649	587	723
2018 PC	FX-6300, 3.5GHz	GTX 1050	253	211	292
2014 Macbook	i5-4278U, 2.60GHz	Intel Iris 5100	51	36	59

Table 6: Performance of the game running on various hardware, from high end modern desktop, to a 2014 laptop with integrated graphics

4.2 Artificial Intelligence

To evaluate the effectiveness of the AI, different versions of the AI are run against each other for 1000 games, and the percentage of won games for each AI is calculated. Each game sample is run on a different randomly generated map to test the AI on different map layouts.

First of all, to ensure that there is no advantage between the placement of friendly and enemy units, 1000 game simulations were run with both enemy and friendly units using 1 level of Minimax search depth and the default hand-weighted evaluation function. After these 1000 simulations, the results in table 7 under the "baseline" test show that the enemy won 50.19% of the time, which shows there is a very slight bias towards the units starting on the enemy side of the map, but the difference is negligible. Therefore, further testing can continue without any changes.

4.2.1 Minimax

In order to evaluate the effectiveness of using Minimax to search multiple turns of actions against only using the evaluation function to evaluate current actions on this turn, the AI was run against itself using the default hand-weighted evaluation function but the enemy AI was run to search 3 turns ahead, while the friendly AI only searches the current turn. After running 1000 game simulations, the result is shown in table 7 under the "0 turns ahead vs three turns ahead" test and the three turns ahead AI wins 76.62% of the time vs the AI that does not look ahead. This shows that the Minimax AI implementation is working correctly, and adequately providing more information to make more favourable action decisions over the same AI that is not searching ahead.

4.2.2 Monte Carlo Tree Search

An identical AI was run against itself, apart from the friendly AI used Monte Carlo tree search, and the enemy did not. The expected outcome for this experiment would be that the enemy AI would win, as it is be sampling more possible move positions. However, the result (seen in table 7) was that the friendly (Monte Carlo tree search version) of the AI won 76.83% of the time vs the version without Monte Carlo tree search.

This is an unexpected result, as it would be expected that using Monte Carlo tree search would improve performance, but not that it would give a strategic advantage. An explanation for this could be that by only sampling 10% of out of cover blocks, the AI is less likely to pick one of these positions. This could be advantageous as it is always advantageous to be in cover in the game. To test this theory, the test was rerun with one player using Monte Carlo tree search and one player not, but the algorithm was changed to sample 70% of out-of-cover blocks and only 10% of in-cover blocks. The result was the player not using Monte Carlo tree search won 58.54% of the time, meaning that the theory was likely correct and that the original Monte Carlo tree search AI was winning because it was more likely to pick in-cover blocks. This could mean that the board evaluation function is not correctly rewarding units for being in cover.

4.2.3 Evaluation Function Tuning

In order to produce the best evaluation function possible, the AI can be run against itself using different evaluation functions for each team. The weights determine how much features increase or decrease the evaluation of a state. In order to complete an effective experiment, only one of the experimental variables (evaluation function weight in this case) will change each time.

The first experiment was changing the dead_enemy_weight. This weight should affect

how aggressive the AI acts, as the higher the weight compared to other weights, the more likely the AI is to take risks in order to kill enemies. The result is shown in table 7, with the friendly AI (dead_enemy_weight 3) winning 95.3% of games vs the dead_enemy_weight 0 AI. Having this weight as three is much better than having it at 0.

Another weight that could change the aggressiveness of the AI would be the dead_friendly_weight, which tells the AI how good/bad it is to lose a friendly unit. First two AIs with dead_friendly_weight set to -3 for the friendly and -1 for the enemy yielded a minimal advantage to the friendly AI, shown in table 7, with only a 56.24% win rate over the enemy. Again, changing the weight to -5 and comparing that to the weight as -1 yielded small but lesser advantage of 54.45% win rate. Overall, this weight seems to have little effect on the AI, but setting it to -3 seems to give a slight advantage. For testing the hit weights (friendly_hit_enemy_weight and enemy_hit_friendly_weight), setting the friendly hit enemy weight to 1, and enemy hit friendly weight to -3 yields the best wine. These results are shown in table 7.

The cover part of the evaluation function was also tested and the results logged in table 7. The wines show that having the cover evaluation is slightly advantageous, with a 56% win rate over the AI without the cover evaluation.

In conclusion, it is clear that the most significant factor in how well the AI performs is optimizing the dead_enemy_weight and the friendly_hit_enemy_weight. An AI with these weights at 0 loses over 90% of the time vs an AI with the correct weight. Another significant factor that affects how much an AI can win over another is how many turns it looks ahead. An AI that only evaluates the current turn loses over 70% of the time vs an AI that looks three turns ahead. This is an excellent result for the AI. It shows that the evaluation function and Minimax search features are having drastic effects on the AIs ability to win games.

	Friendly Winrate	Enemy Winrate
Baseline	49.81%	50.19%
0 turns ahead vs 3 turns ahead	23.38%	76.62%
With vs without Monte Carlo tree search	76.83%	23.17%
Monte Carlo tree search 10% in-cover	41.46%	58.54%
dead_enemy_weight 3 vs 0	95.30%	4.70%
dead_friendly_weight -3 vs -1	56.24%	43.76%
dead_friendly_weight -5 vs -1	54.45%	45.55%
friendly_hit_enemy_weight 0 vs 1	0.01%	99.90%
friendly_hit_enemy_weight 1 vs 3	50.36%	49.64%
enemy_hit_friendly_weight 0 vs -1	6.39%	93.61%
enemy_hit_friendly_weight -1 vs -3	46.75%	53.25%
with cover evaluation vs without cover evaluation	56.24%	43.76%

Table 7: The winrates of the AI playing against itself with different attributes changed, after 1000 game samples

4.3 User Feedback

In-order to evaluate the gameplay, usability of UI and AI vs humans a survey was created. The survey asked the following questions:

- Level of experience in these games
- How many games played and won
- to what extent was the game enjoyable
- to what extent was the game user interface easy to use
- to what extent was the AI hard to play against
- If there was one thing you would change (not add) about the game, what would it be?
- If there was one thing you could add to the game what would it be

All of the users said that they were intermediate players of strategy games, and all of the users played 5 games. Over all 20 games played by the 4 users, only 1 game was won by the AI. one user said the AI was "easy" to play against, and the rest said it was of a "normal" difficulty. All of the users said that they had fun while playing the game, and that the user interface was easy to use.

The users all gave different answers for things they would like to change about the game, their responses are listed and discussed in tables 8 and 9.

User	One change	Author Comments
1	In my opinion, the AI plays a bit sporadically. Sometimes they do really good moves, and other times not. If they stuck to one play style, defensive or offensive, I personally think it would be a bit better.	I would agree with this statement, the AI doesn't really have a play style or any plan that it carries out
2	Graphical fidelity	Obviously the graphical fidelity is not up to modern game standards, although this was not the aim of the project
3	Make the health bars swivel with the camera so they can be read easily from all angles.	This is definitely a good idea, health bars are currently quite hard to read.
4	Make it clear when the enemy team takes its move	This is something that would've been a great addition to the UI, currently it's not always clear when the friendly turn is over

Table 8: User responses for the question "If there was one thing you would change (not add) about the game, what would it be?"

User	One addition	Author Comments
1	Bigger maps with more cover options and generated structures to play around and have more options with would be a great improvement.	This is definitely the case, the map can be bare sometimes, and more cover can help to create better strategic opportunities
2	Variety of actions	More actions would improve the game, however
3	Mouse controls for camera movement	More camera controls would be a great quality of life addition to the game, as well as better rotation
4	A scoring system as well as various map types could add more replay-ability	A scoring system would definitely make the game seem more like a game, and give the player a better long term goal than "kill all the enemies"

Table 9: User responses for the question "If there was one thing you could add to the game what would it be?"

Although this basic testing is not conclusive (it's a fairly small sample size), it shows that when the AI is against even intermediate players, it is unable to win games, and in some cases unable to provide a challenge at all. However, the testing also shows that the overall gameplay was enjoyable and that the user interface was well designed.

5 Future Work

There are a lot of features that could be worked on in order to improve the project and explore further the ideas presented throughout the report. This section lists a few of what the author considers to be the next steps that someone could take to further improve the project.

5.1 Engine

One of the most significant features missing from the engine (and therefore from the game) is animation support. Animation support could massively improve the visual fidelity of the game, and further polish the gameplay. However, 3d animations are considered to be one of the hardest features to implement into an engine, and the graphics of the game were outside of the scope of the game, as the project focus was AI and vital engine features.

Another engine feature that could be added and would drastically increase the ability of the engine to perform well is draw call batching. A draw call is when an object is sent to the GPU for rendering. Each draw call has a big performance penalty, especially if there are many thousands of draw calls happening per frame. In order to reduce the number of draw calls, a lot of engines support draw call batching. Draw call batching reduces multiple draw calls for objects with the same mesh into one draw call, as these objects share enough data to be sent to the GPU as one, and transformed into their individual positions as required. The reason this was not implemented was because the game performed well without this feature, but for future development it would be vital.

5.2 Gameplay

A significant gameplay improvement that could have been added to the game would be performing the AIs turn on another thread. This would mean that when the friendly turn ends, the game is still showing the enemy's progress as it makes its turn, instead of completely freezing the game until all unit actions have been performed which is what happens in the current version of the game. The work for this would include creating a separate representation of the map that is used by the AI code to evaluate different moves and adding the support to create different threads in the engine. This would take a fair amount of development time, but could drastically improve the gameplay experience.

Although a grenade action was implemented into the game, it negatively impacted the ability of the AI and made the game harder to balance overall. The grenade feature allowed units to throw grenades within a particular range and did damage to entities within 3 blocks of the target, as well as destroying all cover in this range. This would have been a powerful

ability and changed the play-style of the game entirely (for example, you would not want multiple units standing close together or using the same cover), and for this reason, it became hard to balance. However, with more time, it could have added more flexibility to the playstyle of the game and provided more strategic challenges without making the game unbalanced.

5.3 Artificial Intelligence

In terms of improving the potential difficulty of playing versus the AI, plenty of further optimizations and improvements could be made to the AI. One of the most compelling improvements that could drastically increase the performance of the AI is implementing narrowing beam Minimax. This optimization allows Minimax to search deeper into the tree by searching fewer and fewer actions in each layer of the tree. This approach could be improved even further by ordering the actions by how likely they are to be the best. The performance of alpha-beta pruning is also improved when ordering actions by their likelihood of being the best action, as the sooner the best or worst action is found, the sooner the alpha or beta cut-off can stop searching further actions.

With more time, an interesting feature to develop for the AI would be an adjustable ai difficulty level. By adjusting the Minimax search depth, as well as tweaking the board evaluation, it could be possible to control the difficulty of the game without changing the rules of the game like other games do (for example, other games might give enemy units more health). This could be an interesting advantage of the AI approach used in the game, it's ability to be tweaked to get different results without changing any of the algorithms used.

6 Conclusions

In conclusion, the project has succeeded in creating a 3d engine, game and AI. The 3d engine performs well on all modern hardware it was tested on and the game was found to be easy to pickup and play by the users that tried it. The project also aimed to provide an example of how a single developer could fesibly produce an engine specifically for a game in a short space of time, and make it effectively in a low level language without Object-Oriented Programming.

The project also aimed to create a competent AI, or to at least test the fesiblity of using Minimax and other methods developed for chess AI in modern video games. Although the AI was fairly competent versus new players, it was not able to provide an effective challenge to more experienced players. With more time I think it is likely that the AI could become challenging for these players, however not without many optimizations to Minimax, and further additions to the game would likely make the AI redundant. For this reason, I think this report shows that other AI methods are more fesible for this type of game such as behaviour trees, and if a training AI is desired more complicated methods such as deep learning would be more effective to produce an AI that challenges advanced players.

7 Reflection on Learning

The largest learning experience was in learning how to properly program in a low level language like C. Especially without using object-oriented programming, learning how to structure code without these paradigms while still creating highly readable, performant and maintainable code was as great of a learning experience as it was a challenge. Another thing I learnt while using C was how to better use pointers, structs, unions and memory to manage and create the required data for use inside the engine. A lot of modern langauges like Java attempt to manage memory for you using garbage collection, but it was a great learning experience to use memory at a low level. As I learnt more throughout the project, I found myself refactoring various parts of the engine to fit the new and improved style that I developed just by writing other engine features and realizing cleaner ways of structuring the code.

Another skill I deveoped throughout the project was using a debugger to resolve code and memory bugs. Using the debugger built into Visual Studio, I could watch how variables and memory changed as the code was run, this helped in quickly finding and resolving bugs. A particularly useful feature of the debugger I used was the ability to break execution whenever a value at a memory location changed, this was very useful for seeing what code was changing the memory to a faulty value.

Optimization was another requirement of the project, and learning how to use VTune to get the most performance out of the game was a great experience. I learnt how to use the software, and possible ways to improve performance like exploring SIMD (although I later found that the compiler was already using SIMD effectively in release builds, so I did not use SIMD in the final code) and improving memory access times by exploiting the cache and avoiding cache misses.

Having regular weekly meetings with my supervisor Dr Frank Langbein was also a great experience, and helped me improve my communication skills and my confidence. As well as keeping me on track, the meetings taught me to reason more effectively about my work and decide on my own what I should be focusing on next.

I have very little AI experience, so it was great to research and implement a full AI player using methods like minimax. I particularly learnt a lot from the optimizations to the AI implementations, like monte carlo and alpha-beta pruning in order to get the best possible performance from the AI player.

Also, a less obvious learning experience was learning how to motivate myself and to plan out days of development to optimize my efficiency. Motivation can be the hardest part of any big project, and I was surprised how I was able to keep motivated throughout the months of development. Mostly I achived this by using Trello to plan out my days and create checklists of things to be completed by the end of the week. This allows me to see how much progress

I am making vs how much progress I am predicting I can make, and can help to adjust the amount of work that I was doing each day to keep up.

Overall, the project has been very successful in my opinion. Not only in it's resulting game, but also in the experience it has provided me and the lessons it has taught me. Being able to work on a project of my choice for months with support of a senior lecturer throughout was an unparalleled experience and I expect to use all it has taught me throughout my career.

Glossary

alpha-beta pruning A search algorithm to attempt to reduce the number of nodes evaluated by Minimax. 41, 43, 55, 57

cache miss Where a piece of data is requested from a cache but is not found in the cache. 45

camera space The space defined relative to the camera, also known as "eye" space. This camera-relative coordinate system is the intermediate step before a world space coordinate is projected onto the screen. 25

CPU Central processing unit, the main computer processor. 45, 59, 60

dot product In reference to geometry, the dot product is a calculation for finding the extent that one vector goes in the direction of another. 23

fragment shader A shader, otherwise known as a "pixel shader", that is operated on each pixel, and controls the color and depth of each pixel on a piece of geometry. 23

GLSL OpenGL Shading Language, a high level programming language similar to C/C++ for writing code for several parts of a graphics card. 14, 17, 22

GPU Graphics processing unit, a co-processor used for graphics acceleration. 18, 20, 54, 60

GUI Graphical user interface, allows users to interact with a computer with graphical icons and visual indicators. 17, 18, 22, 24, 25

LOS Line of sight, what other positions a viewer can see from their current position. 34, 38, 39, 45, 46

memory bound when the CPU is not being fully utilized because it is waiting for data to be retrieved from memory (for example, in a cache miss). 45

mesh A geometric shape made up of triangles. 20, 31, 60

Minimax An artificial intelligence algorithm to attempt to pick the best action based on an evaluation of the game state after trying a specific action. 7, 9, 41–44, 47–50, 55, 56, 59

Monte Carlo tree search A search algorithm which only evaluates a random sampling of the search space based on a heuristic. 9, 41, 43, 49, 51

quad A 4 vertex quadrilateral mesh. 18

raycast A ray that is sent out from a position and moves infinitely in a specific direction. 25, 31

screen space The space defined by the 2d screen, each unit is one pixel. These elements are placed on top of the world, and are relative to the screen and not the 3d world, therefore they stay static when the camera moves. Used for elements that you want to be shown on the screen reguardless of where the camera is looking, such as GUI elements. 11, 18, 25

shader A type of computer program that runs on graphics hardware (usually a GPU), used in computer graphics for various tasks such as post processing, lighting and more. 6, 22, 24, 26, 59, 60

SIMD Single instruction, multiple data is a feature in modern CPUs that allows the same operation to be run on multiple pieces of data, which is faster than doing the same operation for each piece of data multiple times. 57

struct A composite data type that defines a physically grouped list of variables to be placed under one name in a block of memory. 17–21, 24, 33, 34

texture atlas An image containing a collection of smaller images, packed together to reduce the size of the final image. The sub images can then be extracted from varying locations depending on which sub image is wanted. 18, 19, 22

UV coordinates The x and y coordinates of a texture that map to a specific vertex of a mesh. 18

vertex shader A shader that is operated on each vertex individually. 23

VTune Intel VTune Amplifier, a profiler for performance analysis of x86 applications. 45, 57

Wavefront .obj file A file extension for a simple 3d mesh file developed by Wavefront Technologies. 20

world space The space defined by the 3d world, each unit is one "block". When moving the camera, these elements move relative to the camera. Used for elements that you want to be within the 3d world, such as a 3d cube on a 3d map. 11, 17, 18, 25, 26, 59

zero-sum game A game where the gains and losses of one player is exactly mirrored as an opposite loss or gain for the other player (for example, chess is considered a zero-sum game). 41

References

[1] Mike Acton. *CppCon 2014: Mike Acton "Data-Oriented Design and C++"*. URL: https://www.youtube.com/watch?v=rX0ItVEVjHc (visited on 04/05/2019).

[2] Creative Assembly. *Total War: Rome II*. URL: https://www.totalwar.com/games/rome-ii/ (visited on 05/09/2019).

[3] Jeroen Baert. *libmorton - C++ header-only library with methods to efficiently encode/decode Morton codes in/from 2D/3D coordinates*. URL: https://github.com/Forceflow/libmorton (visited on 05/01/2019).

[4] Sean Barratt. *stb - single-file public domain libraries for C/C++*. URL: https://github.com/nothings/stb (visited on 05/01/2019).

[5] Alex Champandard. *Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI*. 2014. URL: http://aigamedev.com/open/coverage/mcts-rome-ii/ (visited on 05/09/2019).

[6] CrossDesigns. *Spike - The Robot*. URL: https://www.turbosquid.com/3d-models/free-3ds-mode-robot-spike/613288 (visited on 04/02/2019).

[7] glfw. *GLFW - An OpenGL library*. URL: https://www.glfw.org (visited on 05/01/2019).

[8] G-Truc. *OpenGL Mathematics*. URL: https://glm.g-truc.net/0.9.9/index.html (visited on 05/01/2019).

[9] Feral Interactive. *XCom 2*. URL: https://xcom.com/ (visited on 05/10/2019).

[10] Branimir Karadzhich. *bgfx - Cross-platform rendering library*. URL: https://github.com/bkaradzic/bgfx (visited on 05/01/2019).

[11] Louis Lafair. *Building an AI that Can Beat You at Your Own Game*. 2018. URL: https://towardsdatascience.com/pathwayz-ai-3c338fb7b33 (visited on 05/09/2019).

[12] Thomas Mueller. *What integer hash function are good that accepts an integer hash key?* URL: https://stackoverflow.com/a/12996028 (visited on 04/10/2019).

[13] MIT OpenCourseware. *Intersection of a line and a plane*. URL: https://ocw.mit.edu/courses/mathematics/18-02sc-multivariable-calculus-fall-2010/1.-vectors-and-matrices/part-c-parametric-equations-for-curves/session-16-intersection-of-a-line-and-a-plane/MIT18_02SC_we_9_comb.pdf (visited on 04/29/2019).

[14] Sebastiano Vigna. *splitmix64.c*. 2015. URL: http://xorshift.di.unimi.it/splitmix64.c (visited on 04/20/2019).

[15] Ozan Yigit. *Hash Functions*. URL: http://www.cse.yorku.ca/~oz/hash.html (visited on 04/20/2019).