Final Year Project - Multiplayer Strategy Game

Adam Rhys Boulton

15th April 2013

Abstract

This report looks into the background, design, implementation and success of my project to create a multiplayer strategy game. Throughout the report, the various problems presented by the development of such a project are thoroughly analysed and evaluated, from game design and playability conisderations to Artificial Intelligence optimisation and programming methods and patterns. This report also suggests future work possibilities and investigates the problems found during implementation.

Contents

List of Algorithms

1 Introduction and Background

The goal for my final year project was to design and implement a multiplayer strategy game, playable against both human opponents and opponents controlled by an Artificial Intelligence. The intention was to build an AI player who will exhibit an ability to think ahead of the current game position, and play moves according to a long term goal rather than simply looking for short term success. The finished AI engine is able to display different behaviours and tactics, customisable by the player, and also react to certain situations and change tactics accordingly. In order to keep a level of realism, the AI player is not guaranteed to play perfect moves, much like how a human player is unlikely to always play optimally.

Solved games are games for which there exists an algorithm that can produce perfect play from any position[5]. Differing types of games offer different types of challenge for an AI: for a game that is solvable, the challenge lies with creating an opponent that will provide an entertaining opponent for a human, rather than always winning. On a non solvable game, we try to create an opponent who can play as well as possible, but may not be unbeatable in most circumstances. Therefore, no matter whether a game is solvable or not, a developer can still create interesting AI opponents.

The design section offers a more complete overview of the design of the gameplay systems, but the game's core revolves around two small teams of units battling on a customisable piece of terrain. The units fall in to certain archetypes to make their function obvious: snipers are long range specialists, for example. The characteristics of each unit make it obvious to the player what they should be using it for rather than having to tell players how to play. The gameplay should be largely self explanatory for a new player, including the game's controls and winning states.

The main audience for this project would be those beginning to look into how to implement games and Artificial Intelligence in Java. Designing and programming a large project such as this is vastly different to implementing the smaller projects that encompass a programmer's first few endeavours. With a large project, the demands on a coder are far more severe, requiring a far deeper knowledge of the programming language being used, and good planning is essential. A small project might consist of a single routine that must be created and implemented, but a large project such as this consists of a multitude of diverse systems, structures and algorithms which must all be implemented in a way such that each individual component is seamlessly compatible with each other component. Each of these components could be as large, if not larger than the projects a novice programmer has tackled in the past. Failure to plan ahead and ensure total compatability could easily result in a broken, messy and frustrating implementation.

By heeding the information contained in this report, another user should be able to dive into a similar project with a strong awareness and understanding of the pitfalls that litter these kinds of undertakings, and implement a project that builds upon the conclusions drawn throughout this report, building a system that is expansible and flexible, complemented with an AI Engine with a capacity for deep thought and excellent optimisation.

Others with an interest in Artificial Intelligence will also be able to look into the evaluation of the AI Engine's performance and compare it to other approaches to similar games: areas where my implementation improves on others could be adapted to help other projects, and the flaws I uncovered avoided. Other implementations offer solutions both more and less complex than my own, and the differences in their efficiency and ability to play well are certainly worthy of analysis.

I would also consider myself a beneficiary of this project: since the project's outset, my have vastly expanded my knowledge of Object Oriented Programming and design, as well as in the field of Artificial Intelligence. Over the past year I have repeatedly had the satisfaction of seeing my designs successfully come to life on screen after hours, days and weeks of hard work.

The project itself covers a rather wide scope: implementing a game means bringing together systems, algorithms and theories that may not usually interact together as a single package. The design phase of the project yielded a number of questions that had to be answered, and how I answered them would affect the rest of the project in its entirety.

The design phase of the project revealed a number of challenges that would demonstrate the scope of the project. Interface design plays a major role in any video game: menus should be clear to the user, and the user should be able to achieve their goals with as few clicks as possible, without overloading them with information. As mentioned previously, the design of the game is also important: the game needs to be complex enough to build an interesting AI opponent, but also intuitive enough for players to quickly understand the game's rules. The underlying framework of the game is also a major question: issues such as pathfinding (finding the shortest legal path between two points) and collision detection are not easily solved, each with hundreds of possible approaches, none of which offer a catch all solution perfect for every project. The same is true for building an AI engine itself, as every game has a completely different set of rules, and therefore demands a completely different style of opponent with it.

In order to achieve success, I took an approach to ensure I had thought of as many requirements as possible before implementation. This meant everything from drawing mockups and draughts of the interface, to considering the exact implementation of each feature before I started programming. The choice of pathfinding algorithm, AI engine approach and collision detection would all have major bearing on the underlying implementation of the game itself. I knew I needed to think of each component and class ahead of development to ensure compatability with every module. This included sketches of classes and their interactions, from the grid based movement system, all the way through to the way units are represented in the game. Failure to think of major components in advance could easily cause headaches at later points in development: sometimes a simple solution is not the best one, as the simple solution could cause compatability issues in the future. In order to help with the design spent some time experimenting with various different approaches in Java, before deciding to use a number of classes from the Slick game library as a base for my game. The Slick library classes used are discussed in more detail in the Design section.

There are a few assumptions on which my work is based. I have to assume that a human player will have a reasonably limited amount of patience: if my AI player were to spend too long making a move, a player would normally become bored of waiting. As a result of this, the AI needs to be extremely well optimised to ensure it finds a good move in the shortest time it possible can. A further assumption is that a player will not find a perfect AI opponent especially fun to play against: the AI needed to play well, but be beatable by the player. A good way to achieve this is to offer multiple AIs, each of which will play slightly different to the last.

The barometer by which the project's success will be measured is whether or not I achieve everything I set out to do at the start of the project: whether the game is suitably complex yet easily understandable, whether the AI opponent is well optimised and a strong player, and whether the underlying implementation is has a clean structure and makes best use of all the options Java offers the programmer. Early on in the project I established a list of goals and objectives for the project, and these are listed in **Appendix A**.

My final implementation achieves all of these goals. The following sections explain how I managed to achieve each section through rigourous design, implementation and testing, and an evaluation of how fully each goal was achieved. There were also a few other additional features that I implemented, including a map editor.

2 Design

As mentioned in the introduction to this report, the design phase is hugely important in implementing a large software project. For example, imagine a simple program that simulates a function f. At the initial implementation, the program is simple with limited functionality. Extra functionality is easily implemented - colouring certain features, adding text to display certain variables, and so on. However, one has to consider the effect this would have on the structure and "tidiness" of the underlying code. Imagine if we wanted the system to calculate and display a set of values: we'd need routines to calculate those values. If we already had a similar function to those calculations, we could copy that function and use it for a new calculation. But if the initial implementation had a bug, so will every other implementation. Maybe we want to toggle visibility of certain elements: this could easily be handled with an if statement. But if we keep adding elements without prior design, a project quickly becomes unwieldy, like a tower without a proper base. And if the base needs to be changed, what happens to everything that has been built on it? What if we need some new functionality that would require a rewrite of one component, and subsequently stop other components from working? Thorough software design helps to avoid eventualities such as this, by considering a system as a whole, rather than starting at an arbitrary point and building around that point.[12]

Poor design can also have repercussions for the user. A badly designed system can needlessly obfuscate systems and functionality from a user, and a programmer may not notice these flaws due to it being obvious them how something works. When someone wants a house built, they first talk to an architect. They wouldn't let an engineer both design and build the house, so why should we build a software project without first properly considering its design? Design does not just mean designing an interface: it is the overall conception of the product. The feel of a product is only a single part of design, which is why we must consider requirements, interfaces, and underlying architecture.[7]. The design needs to give a clear picture of the planned system, both in terms of visual feel and capability required.

On the website for the book 'Code Simplicity' by Max Kanat-Alexander, the science of software design is described as such:[6]

- "An explanation of the purpose of software."
- "An explanation of the goals of the science."
- "A series of fundamental truths on which to base decisions."

"And this would allow us to achieve:

- "The ability to make decisions that achieve the stated purpose of software."
- "Some way of understanding what causes errors (decisions that do not achieve the purpose) in software design."
- "A method (or methods) of preventing future errors."
- "A method (or methods) of fixing errors that already exist."

In short, good design helps us create software that is useable and understandable by the intended audience at the level of interaction, and below that level, a strong modular design that allows expansion and customisation, easily readable and understandable by someone who know the language the software is written in. Software design provides a design view across every layer of a system, rather than just the layer seen by the engineer, or the layer seen by the user.

2.1 System Requirements

Before designing a system, a developer needs to formulate a set of requirements to fulfill. Developing without a clear set of goals can be dangerous: resources could be focused trying to solve issues that are of lesser importance than others. With a clear set of requirements, I know exactly what I want to achieve during my implementation. I can derive a set of requirements from my list of goals, so define a complete description of the behaviour of the system.

2.1.1 Overall Description & Definitions

The overall description of the project is largely covered by the list goals, which provide a good overview of the total features and functionality of the game: a strategy game, playable by two players (two human players, or one human player versus an AI), played on a two-dimensional grid featuring a variety of units. The winning player is the one who first defeats all of the opposing teams units, or who has the most units remaining when the pre-set turn limit is met. Gameplay takes place in a turn-based format: each player takes in turns to make a single move before control is passed to a second player. This section covers the definitions for the project, as well as some external requirements. A full list of system requirements is in **Appendix B.**

The definitions that must be considered are:

- **Pathfinding** - the method for finding a path from any co-ordinate in the game space to another. These algorithms often form the core of an AI opponent, as an AI who cannot find his way around a map would not be a particularly interesting or challenging opponent.[13] A good pathfinding system therefore needs to be included as part of the system's design
- **Collision Detection** - units attack each other by "shooting". Given that game maps can include walls, it makes sense that these walls should protect players from being shot at by other units. Calculating a collision between bullet and wall is somewhat trivial if we shoot in straight lines or pure diagonals (equal change in x/y). Collisions on other diagonals are more complicated. Collision Detection in this game refers to the method used to calculate these situations.
- **AI Engine** - the system by which the AI opponent chooses which move will be best to play. There are several sub definitions, each of which must be taken into account during the design
  - **Move generation** - the method by which potential moves for a player are calculated
  - **State generation** - the way by which new states are derived from potential moves
  - **Evaluation** - the methods by which potential states are scored to determine how strong a position is for a player

- **Unit** - if this were a board game, a unit would be equivalent to a piece. Each has their own behaviour and abilities that a player can exploit. The features of each unit will have to be considered during the design phase, but should remain flexible and changeable so that they can be adjusted for balance once the game is operational. If a unit is overpowered, it should be easy to scale back his attack or movement range to improve balance.

- **Tile** - each space on the 2D grid is referred to as a "tile". Tiles can have different properties to change how units can interact with them: units can move over trees and grass, but not over walls and water. Units can shoot over grass and water, but not through walls and trees.

2.1.2 External Interface Requirements

- **Information -** the game must feature a full GUI that will display relevant information to the user without any extra effort on the user's part. Information should not be relayed to the user through text output to the Java console. This can be used for debugging, but any information meant specifically for a player should be presented in a way that is as unobtrusive as possible.

- **Feedback** - it must be clear that when the user clicks on an object or element on screen, that their click is having an effect. This could be as simple as a button depressing visually when clicked, or drawing a box around a unit when a player clicks on it. Every action needs to have a linked reaction. If a player selects a move that is not legal, they need to be informed, and given a reason why that move is not legal, else they may get frustrated. When the AI is making a move, the player will not be able to interact with the game. This needs to be communicated to the player: a message saying "please wait, AI is thinking" would be one way of doing this.

- **Clarity** - it should be clear what each element of the screen does as first glance. Buttons should be self descriptive: one saying "exit" should exit the game, and one saying "close" should close the current window. Labelling should match the user's expected reaction from pressing such a button. If a button labelled close exits the system, the user could get confused. Users should also be able to navigate the interface without external help or assistance.

- **Consistency** - fonts and font sizes, colours and so on should be consistent across all screens of the interface. Mixing fonts and colours could cause unwanted confusion for the user

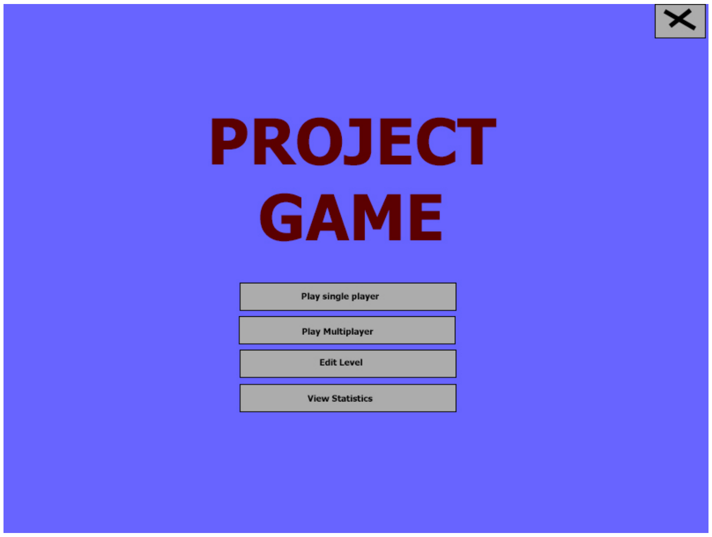
- **Resolution** - the game window should dynamically resize itself based on the size of the game map being played. No matter how small the map being played is, the game should not need to hide elements that would otherwise be visible on a larger map. The user should never need to manually resize the game screen.

- **Usability** - the standard colours for units will be red and blue, two very distinctive colours and the grass tiles are green. This ensures that each colour stands out, rather than objects appearing to blend together.

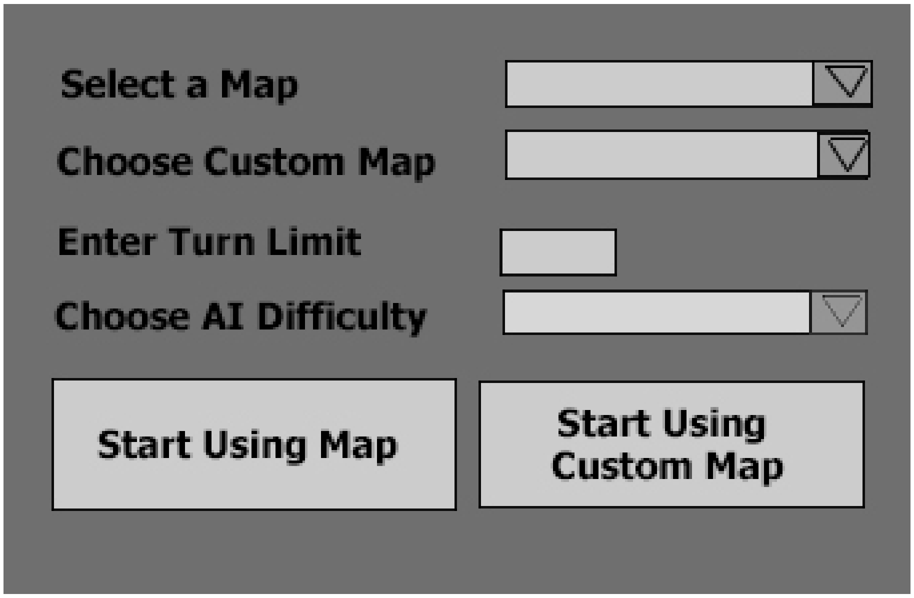
2.2 Interface Design

Navigation through my game's interface will be made as simple as possible by having as few screens as possible. In this section I have several mock-up designs for the interface, along with explanations of their contents, and the general flow of the interface.

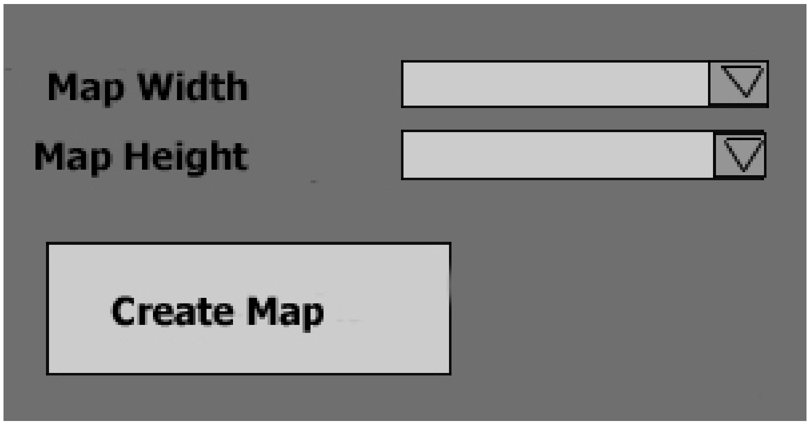
2.2.1 Main Menu + Popups



The main menu is kept as simple as possible. The blue background is simply a placeholder for a more interesting background image. The play multi player and edit level button create the following windows.

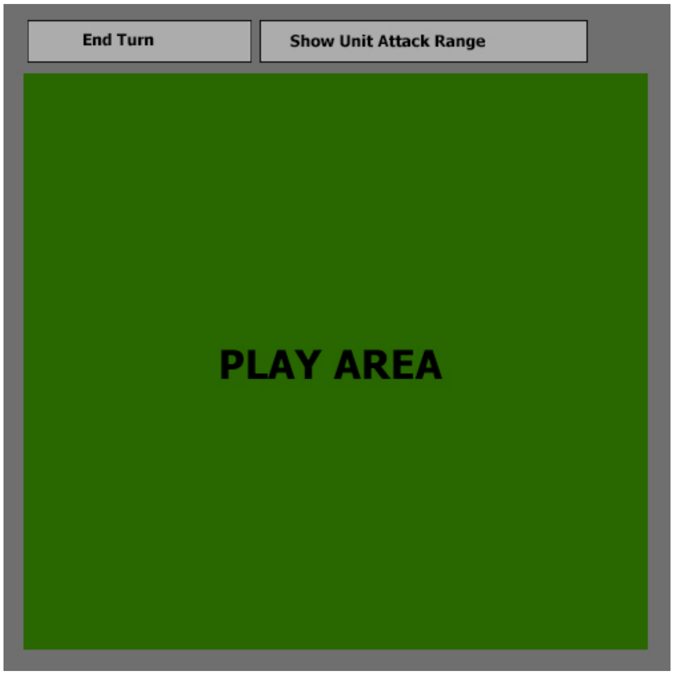


Drop down boxes are generally better for selecting options which have a limit as there is no doubt to the user as to what they are allowed to select.



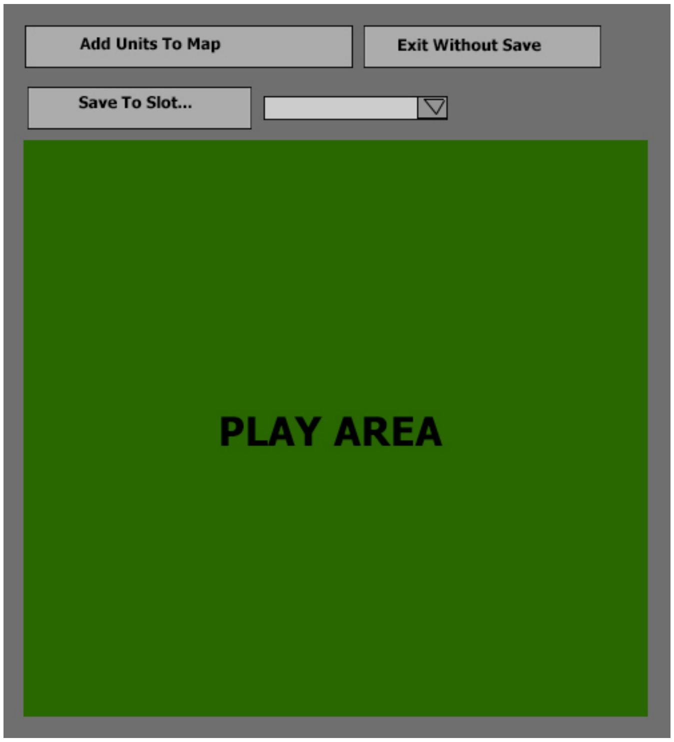
2.2.2 Game Screen

The gameplay screen is kept as simple as possible. Most information relevant to the player is displayed in the play area, and they can see how many units remain etc, as a result. Two further buttons exists to end turns and to trigger attack range calculations. Closing this window returns you to the main menu.

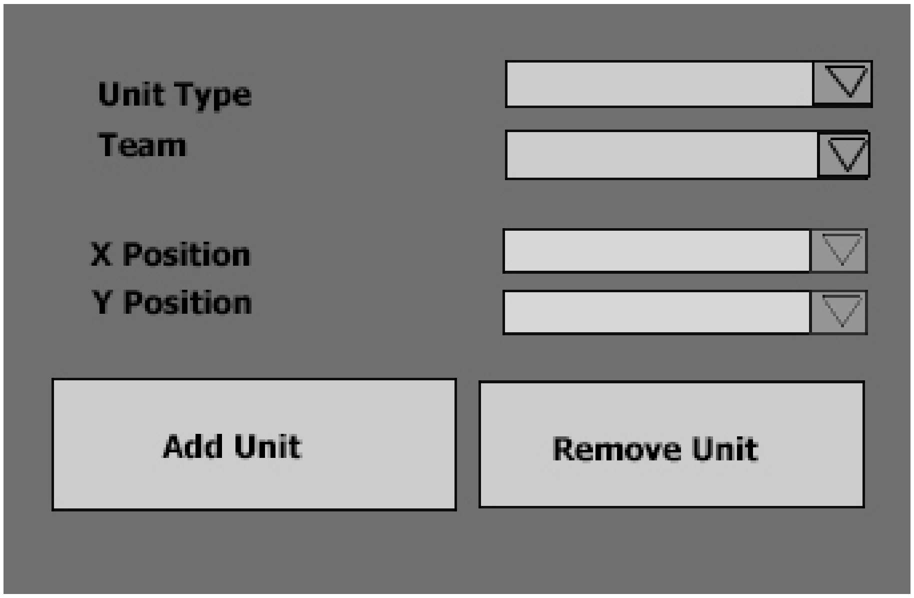


2.2.3 Map Editor + Popup

Largely similar to the main game screen, however the buttons are different: add units to map creates the dialogue box below this figure, and save to slot performs serialisation of the map to the save slot selected in the drop down menu. Closing this window returns to the main menu.

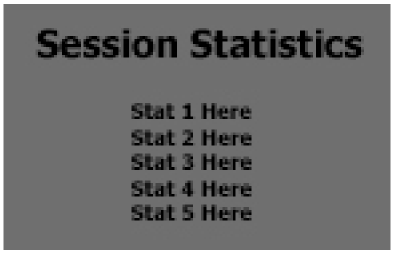


This window allows us to add units to an editor created map. We can only add units if all four boxes are filled in, and removing units only requires the x Position and y Position drop downs to be filled.



2.2.4 Statistics

This is a very simple popup that displays the statistics accumulated over the current game session: total turns this session, red kills this session, blue wins this session, and so on. Closing this window returns to the main menu.



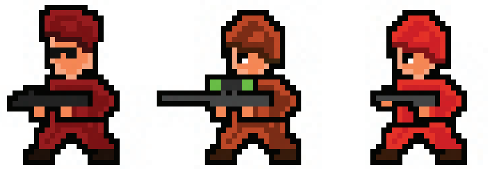
Session Statistics

Stat 1 Here
Stat 2 Here
Stat 3 Here
Stat 4 Here
Stat 5 Here

2.3 Visual Design

The graphics for the project game need to be functional rather than especially attractive. I did not want to spend a huge amount of time on the visual elements of the project, as the main focus was to build a working game with a strong AI opponent. My main focus when designing the visual elements in the game was that each of them is distinct, and makes sense: for example, the user should be able to easily tell a soldier and sniper apart, and the grass should be green and not blue. To create the graphics I used Adobe Photoshop to create transparent 32x32 pixel images, and drew the tiles from scratch using pixel editing, which gave me full control over the look of each unit.

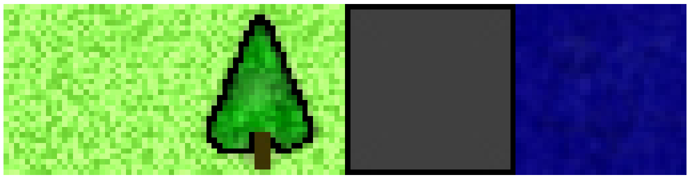
2.3.1 Units



Above we have the 3 designs for the red team's units. Despite their small size, each is distinctly different in several ways. The standard soldier unit, furthest on the right, looks the most like a standard unit: a simple uniform, weapon and helmet, with very few elements that stand out compared to the other two designs. It suggests to the user that this unit will be the average unit, similarly to how in chess a pawn might be considered a standard piece because it has the most basic shape of all the pieces. In contrast to the soldier, the commando unit on the left of the image has a number of features that make him distinct. Unlike the other two units, he appears to wear a beret rather than a helmet. The colour of his uniform is slightly darker, and he appears to be wearing some form of sunglasses as well. The aim of this design was to make a unit that looked more aggressive than the others: with a larger move distance but shorter attacking range, the commando is best used to quickly close distances and flank enemies, and his design suggests to the player that he should be used aggressively.

The last of the units, the sniper, stands in the middle of the image. He shares some similarities with the soldier, as he wears a helmet. However, his uniform is a lighter colour in places, turning toward a more brown colour in others. He also carries a far larger gun, which appears to have some form of sight attached to the top. These two features suggest that the unit might want to blend in and attack from range, which is exactly what I want because the sniper has the largest attack distance of any unit, but is not very mobile, which leaves him vulnerable to commando units.

2.3.2 Tiles



Above are the designs for each of the tiles in the game. As with the units, they are designed to appear as distinctive as possible. The grass and water tiles at opposite ends of the image are deliberately not a solid colour. Initial designs were a solid colour, but I found that due to the slight colour contrasts within each of the unit designs, they looked out of place. I used photoshop's noise filter to adjust the images so that there is more of a range of colour in the tiles, which made them appear more visually pleasing when placed alongside the units. The wall tile is the only tile which is a solid colour, so that it is clear to the player that it is a solid object that cannot be fired through. The tree tile has a tree superimposed on top of a grass tile: this allows trees and grass tiles to be placed against each other seamlessly.

2.4 Gameplay Design and Features

Earlier in this report it was explained that the game needs to be complex enough that it is non solvable, but also simple enough so that its rules are easy to grasp and give the game a "pick up and play" quality. Much of how the game plays is dictated by the units and the tiles, but there are a few other rules that will also have an effect on players.

2.4.1 Tile Types and Their Properties

Tile types in the game change how units can interact with the map and with each other. There are four types of terrain. Grass serves as the default til type, and has no special properties. Grass tiles can both be fired over and moved over by units. Because of this, they are the most common tile that users will encounter, making up the majority of almost all maps.

The wall tile is at the opposite end of the scale to a grass tile: a wall tile cannot be fired through or moved over. This allows a map design to be more interesting: creating pathways for units to follow, rather than just having plain, open ground arenas for players to face off in. As well as making maps more varied, wall tiles are by far the most effective type of cover, and can be used to help players to protect their own units and prepare attacks and ambushes. Walls can also be used to break sightlines on maps that have large open spaces: despite the sniper having a weakness in that it cannot move huge distances, a poorly designed map could easily make snipers overly powerful by allowing a sniper to cover too large a portion of the map.

The two remaining tiles fall in the middle of the scale. The water tile blocks unit movement much like a wall, but does not block shots. The tree tile is the opposite: the tile does not inhibit movement, but will block shots. Water tiles can be used by a player to help set up a trap or ambush, as it reduces movement options for units, but does not protect them from attack like a wall. Tree tiles can help 'hide' a unit: they do not completely block shots, but they do make a unit harder to target. This effect will be achieved during the distance calculation for an attack: if a unit has an attack range of 5, when a player tries to attack another unit using the selected one, the game will check first if the shot is within legal range, then if the shot is blocked by a wall. Each tree in the path of a shot will subtract a point from that units attack range: so if a sniper spies a unit 8 squares away, normally they would be able to attack if there is no wall in the way. If there is a tree in the way though, it would reduce the sniper's effective range to 7, and stop the attack. This means trees offer less protection than a wall, but clever use of these tiles allows players to protect a unit even while it advances.

2.4.2 Unit Types and Their Properties

During design, I settled on having three different types of unit: the soldier, the sniper and the commando. The first of these three is the soldier. The soldier was intended to be the “standard” unit available to players. In most of the maps I included with the game, the number of soldiers is greater than any other unit, often larger than the rest of the unit types added together. As the most common unit, players will identify it as a standard unit. The soldier can move 5 squares in a single turn, which is lower than a commando but higher than a sniper. He can attack in a radius of 3 squares: this makes it difficult for a soldier to defeat a commando or a sniper for varying reasons. In the former, the commando can easily escape, and in the latter case, the sniper could easily pick off an approaching soldier. This makes it important to use cover to protect soldiers, and also to use them to create traps. Their strength comes in numbers: on their own they are fairly weak units, but clever use of soldiers can help break trap enemies.

The second unit type is the commando. The commando is a less common unit than the soldier, emphasising to the player that he is a more valuable unit. At 2 squares he has a shorter attack range than a soldier, meaning he must be extremely close to an enemy to attack him. However, this is mitigated by his extremely high movement range of 8. Commandos should be effective as hit-and-run units, able to quickly get rid of units that have been placed carelessly, before easily retreating out of range to safety. Despite their power, a player who is careless could still find a commando being trapped by a well placed soldier or sniper.

The final unit type, the sniper, has by far the smallest move range of just 3. Coupled with an extremely large attack range of 8, the sniper is best used to close off certain areas of a level to the enemy. Any enemy that crosses into the range of a sniper is immediately at risk, however the closer an enemy gets to a sniper, the more danger the sniper is in, as the low movement range immediately starts to work in the opponent's favour. Therefore, a sniper must be positioned very carefully or supported with another unit, else the advantage of having such a unit might easily be lost. The sniper is of similar rarity to the commando, which again makes it clear to the player that these units are more valuable than others. Both the sniper and commando, whilst extremely strong in one regard, are very weak in other, which helps keep them balanced and asserts them as something of a "glass cannon" archetype.

2.4.3 Win Conditions and other features

The game will track how many units each player has. If either player's unit count drops to 0, the game will end, and the winner will be the player who still has units remaining. As an alternative option, players can select a maximum amount of moves before the game will end, which helps reduce stalemate situations. If the limit is hit, the game compares how many units each player has remaining, and whoever has more units will be proclaimed the winner.

In order to prevent frustration for players, an option to preview a unit's attack range will also be in the game: a unit can be selected, and all the cells that can be attacked from the current location will be shown. This will help give players some idea of what their unit will be able to do once they move it.

2.4.4 The Game Board

The approach for the grid implementation was initially discussed in section 3 of the interim report. The terrain array will be represented by a 2 dimensional array of integers. Different values will represent different types of terrain, and based on the integer contained in a cell, the different modules of the game will be able to choose what to do: the collision detection will be able to tell if a cell is a wall or a tree and use that knowledge to block a shot, for example.

There will be a second grid to keep track of unit positions. This grid will consist of UnitNode objects that as well as having a pair of co ordinates, will also point to an instance of a unit. This will allow methods to check cells to see if there is a unit present there or not with ease, an essential feature for the AI and pathfinder modules to work effectively.

2.4.5 The Map Editor

The map editor needs to be kept as simple as possible. The above design for the game board would make implementing a map editor fairly simple: clicking on a cell could increment the type of terrain it contains. In the interim report I discussed exporting the grid of integers and unit locations to a text file, and creating a method to parse this information back into the game when loading map files. However, a better method for implementing saving and loading objects from a file is already built into Java. The serializable options for Java allow any object to be saved to a .ser file, and then retrieved later on. As long as all the objects that are contained within the object being serialized are also serializable, it can be successfully saved. When an object is retrieved, it is retrieved as a generic object, so it will need to be cast back into the type of object it was initially. I plan to make the class that holds Game Maps serializable so I can save everything related to a map, including unit instances being used by the game.

2.5 Major Challenges

In the interim report, I identified three major problems that would need to be solved during the implementation of this game. I decided on the best approach for each one. This section covers my design and how I planned to implement each of the solutions.

2.5.1 AI Design

My AI will be a minimax and heuristic based opponent. Minimax is a popular choice for developers implementing a computer controlled opponent in games that have a clearly defined set of rules. Because of the clear rules, it is easy to compute the possible moves at any given game state (in some cases this may not be true: some games have massive potential move sets that would make complete calculation too slow to be practical. The midgame for chess for example, as many as 10 pieces could have 5-10 possible moves each, meaning 100 potential moves at the first level) as each player can also see where both the opponent's and their own pieces are located at any time[11]. The general operation of Minimax is explained in the interim report.

In order for a minimax effectively, we need to analyse each state to compare how "good" it is in relation to others on the same level. This is accomplished using a heuristic that calculates a numerical score based on the gamestate it is analysing. The nature of Minimax therefore means I need four main designs: the minimax algorithm itself, the way by which potential turns are generated, the way from which new states are generated as a result of the potential turns, and the heuristic to evaluate states.

Algorithm 1 shows a pseudocode implementation of a Minimax algorithm that also incorporates Alpha-Beta pruning, which was discussed in the interim report (**Section 2.3**): we set a value for alpha and beta, typically negative infinity for alpha, and positive infinity for beta. At the max level, alpha is set as the maximum value of the nodes on that level, and at the min level, beta is set as the minimum value of the nodes. By using these initialisations, we know that if we ever encounter a Min node with a beta that is less than alpha, we do not need to search below that node since the value can only ever decrease or remain at that value, as no higher value was propagated up from below. This allows us to prune branches of the tree that are not promising, which speeds up searches significantly[9].

Algorithm 1 Alpha-Beta Minimax

```
chooseMove(GameInstance)
        state = current Game State;
        List<Turn> turns = list of possible moves
        bestScore = 0
        for (each Turn t in turns)
                nextState = new state based on t
                returnedScore = getMin(nextState, 5, -infinity, +infinity)
                if (returnedScore > bestScore OR bestMove is empty)
                        bestMove = t
                        bestScore = returnedScore
        return bestMove


getMin (state s, depth, alpha, beta)
        if (depth is 0 or state is a final state)
                return the heuristic score for s

        List<Turn> minTurns = list of possible moves based on s
        res = +infinity
        for (each Turn t in minTurns)
                nextState = new state created from t and s;
                returnedScore = getMax(nextState, depth-1, alpha, beta)
                res = smallest of res and returnedScore
                beta = smallest of beta and returnedScore
                if (beta <= alpha)
                        break
        return res


getMin (state s, depth, alpha, beta)
        if (depth is 0 or state is a final state)
                return the heuristic score for s

        List<Turn> maxTurns = list of possible moves based on s
        res = -infinity
        for (each Turn t in minTurns)
                nextState = new state created from t and s;
                returnedScore = getMin(nextState, depth-1, alpha, beta)
                res = largest of res and returnedScore
                alpha = largest of alpha and returnedScore
                if (beta <= alpha)
                        break
        return res
```

Algorithm 2 describes how generation of the list of possible moves will be implemented. The variable "firedFirst" is a boolean. It is essential to the operation of the algorithm because when a new state is generated based off a turn, the engine will need to know whether to move a unit then attack, or attack then move a unit. If the unit moved and then attacked, but the engine tries to attack then move, the co-ordinates the game looks in to find the unit that needs to attack would be empty, as the co ordinate for the attacking unit would be based on the units position after the move.

Algorithm 2 Move generation

```
generateMoves(state s)
        For (each friendly unit that is alive)
                calculate all possible attacks that can be made
                set firedFirst to true
                add attacks to list potentialMoves
                add attacks to list firedMoves
        For (each move in firedMoves)
                calculate all moves that can be made from the unit that
                fired's location
                add moves to list potentialMoves
        For (each friendly unit that is alive)
                calculate all possible moves that can be made
                set firedFirst to false
                add moves to list potentialMoves
                add moves to list nonFiredMoves
        For (each move in nonFiredMoves)
                calculate all attacks that can be made by the unit
                that just moved
                add moves to list potentialMoves
        Add a move where the there is no move and no shot
```

Algorithm 3 describes how a new state is generated from a turn.

Algorithm 3 State generation

```
generateState(turn t, state s)
        if (t.firedFirst is true)
                remove the unit that is attacked from the game
                if (there is also a move to complete)
                        move the unit from its current location to a new one
        else if (t.firedFirst is false)
                move the unit from its current location to a new one
                if (there is also an attack to complete)
                        remove the unit that is attacked from the game
        else if (there is a blank move)
                do nothing
```

The heuristic used to **evaluate the strength of a state** will be based off the following assumptions. Some of these assumptions were discarded during implementation due to being unnecessary

- A winning state is good (add an extremely large amount of points to the state's score)
- A losing state is bad (remove an extremely large number of points from the state's score)
- Killing a sniper or commando unit is extremely beneficial (add a very large number of points to the state's score, but less than a winning state)

- Killing a soldier unit is very beneficial (add a medium number of points to the state's score)
- Turns where a player moves are generally more productive than moves where there is no action at all because it likely shows an attempt to move units away from harm, or into position for attack (add a very small number of points to the state's score)
- Turns where a player moves and fires are also more productive (add a very small number of points)
- Situations where the player has more units remaining than the opponent are generally beneficial (add a large number of points to this state's score if the player has more units, and subtract the same number if the player has fewer units)
- Add a small number of points for each friendly unit that is in play
- Subtract a small number of points for each enemy unit that is in play (in this and the previous case, snipers and commandos will be worth more than a soldier. By adding one and subtracting the other, we can see if a situation could be advantageous for a player, even if they have fewer units)
- If the move has left the player in a position where their opponent could kill one of their units on the next turn, subtract a medium number of points.

Different AI behaviours will be able to be produced by tweaking this heuristic: if I increase the rewards for killing units and decrease the penalties for losing them, the AI player should be notably more aggressive, as it views taking units as the best thing it can do. If I do the reverse, the AI will become defensive, doing its best to protect its units.

Further complexity can be added through situational heuristics: for example, if the AI finds itself in a situation where it suddenly has fewer units to the opponent, a different heuristic could be activated where the AI attempts to maximise the distance between its units and the opponents. This will be a very good way of producing a believable AI opponent.

An AI opponent will be more fun to play against if it does not always play perfectly. A good way to stop the AI playing perfectly would be to implement a random element to the heuristic: after all the calculations have been completed, a random number between two values could be generated, and added to the score. The number should have the potential to be sufficiently large that it could push sub-optimal moves up so that they receive higher scores than the optimal move, or sufficiently small such that high scoring moves could be pushed down so they receive lower scores. Changing the minimum and maximum size of this random addition would be a good way of implementing a difficulty level: the larger the random addition/subtraction, the greater the chance that a bad move will be played by the AI. This could be combined with changing the maximum depth that the AI is allowed to search to: in theory, an AI that thinks 10 turns ahead will be stronger than on which only thinks 5 turns ahead[1].

2.5.2 Collision Detection Design

As mentioned previously, detecting if a wall blocks a shot is easy when considering only pure diagonals or straight horizontal/vertical attacks, but more complicated in any other situation. As described in the interim report (**Section 3.1.3**), we use the attacking unit and target unit's positions to calculate a line, and intersect this line with the co-ordinates of the walls on the grid. It should be noted that this algorithm will only be used in situations where neither the change in x or the change in y are zero. If we encounter either being zero, this means we have a horizontal or vertical shot, and can simply check all the cells in a straight line between the two units for walls instead. This algorithm does not work if change in x or change in y is zero.

2.5.3 Pathfinding Design and Algorithms

My interim report (**Section 2.2**) identified the A* pathfinding algorithm as the best pathfinding solution for my game, and relevant pseudocode. A* combines a best first approach with a heuristic one. For each node in a grid, we have the cost of getting from the start node to the current node (the node we have reached in the search), g which is the estimated cost from the current node to the goal node h (based on an admissible heuristic that will never overestimate the actual cost), and the sum of g and h to create a best estimate f of the cost of the path going through the current node. The algorithm also uses two lists, an 'open' list that contains all the nodes that have not yet been examined, and a 'closed' list that contains all the nodes that have been fully explored.

I chose the A* pathfinder because it will always find the shortest possible path from point A to point B if an admissible heuristic is used, and will always move toward the general direction of the goal as long as the estimated cost is lower. A* is also relatively quick to compute, and as my units only move 3-8 squares of movement range, it is an ideal fit for my project. It should be noted that whilst in some cases a pathfinding table could be created, containing all the possible moves from every square to every other square, in my game this will not be possible, since units will block the movement of other units. Because of this, the table would have to be recalculated every turn, which would be extremely inefficient.

2.6 The Slick2D library

The Slick2D library[8] is a set of tools and utilities wrapped around LWJGL and OpenGL bindings to make 2D Java game development easier. I found Slick2D while researching the implementation of an A* pathfinder algorithm. Rather than using the whole toolset, I made use of a small number of classes to help begin the design and implementation of the project. The versions of these classes came from an example on Slick2D developer Kevin Glass's website[4]. These classes provided strong bases for storing and displaying game maps, as well as re-usable pathfinder code. Most of these classes would need to be hugely adapted during the implementation, leaving them mostly unrecognisable, but they provide an excellent launching point for development. As well as my own code, the classes from which my code was built and adapted are also included in the appendix/other files of this document.

3 Implementation

The design section of this report describes the system at a high level. This section looks at the actual implementation of the system in greater detail. The purpose of this section is not describe individual segments of code: the full program listings including comments are included in the appendix/additional files of this report. Instead, this section outlines and explains the more unusual and interesting techniques used in implementation, and major deviations from the design.

3.1 The Game Package

The final implementation of the game features four packages: game, pathfinder, players and units. The game package handles the majority of the core game logic, its interface and the structures that form the core of the game: GameMaps, Turns and so on. The package consists of the classes described in the Design section as well as a number of additional classes:

- **AIPopup** - creates a frame that pops up when the AI is thinking
- **ChooseUnitWindow** - used by the editor to create a window that allows players to add units into the game
- **EndGameWindow -** window created whenever a game hits a win condition or the turn limit expires

3.1.1 The Game Interface

This class is not a Java interface; rather the name refers to the fact that the class is responsible for the game's user interface. This class is responsible for launching the entire game. Before the main method, a few important variables are created:

- **mapSaved** - this static boolean variable is used to check that a custom map saved locally exists. As well as saving custom maps to file, custom maps can be saved for temporary use and played by selecting "custom map" on the map selection. If no custom map exists to load, an exception occurs. A custom map will only be allowed to load if this variable is set to true. Setting this variable to "static" means that it does not need to be instanced to be accessed, and can be treated as a constant: if a map has been saved locally or loaded to file, this variable can be updated to reflect that. Actual saved maps are stored in the savedMap variable in the editor (also static), but this variable is in GameInterface as it is the only class that actively uses the variable's contents.

- **choosewidth** and **chooseheight** - these static variables are used so that the MapLayouts createMap method knows what size map to create when building a map for the editor. When creating a map, the editor uses mapselection = 0 to get a blank map, and the height and width of that map are set using the value stored in these two variables. The two variables are manipulated by the MapParams class, which is called so that players can choose a custom map height/width when loading the editor. Without these variables, users would not be able to customise map sizes and would be forced to use the default 15x15 style.

GameInterface makes use of layeredPanes to organise content. Layered panes are extremely useful and used throughout the game's interface. In this window's case, a JLabel is used to display a background image. Several buttons are also placed in the pane to allow the user to navigate the system. Without a layeredPane, these buttons would be 'stuck' within the background label and only visible when hovered over by the mouse. The layeredPane allows me to place window components on top of each other, and likewise, btnMapEdit opens the map editor paramaters window. Selecting either of these two windows keeps the parent visible so that the system remains running if these pop up windows are closed.

The combo boxes in MapParams required a slightly different approach to most combo boxes in the system: normally we can simply retrieve the selection index and use that to manipulate some function. The combo boxes here change choosewidth/height, but they do not start at 0. A switch statement could have been used to translate from index to actual value, but a more efficient method was to create a pair of string arrays 15 units long. The arrays were looped through to fill them with values from 10 to 25. When btnGo is activated, choosewidth and chooseheight are set to be the values pulled from the combo boxes. ParseInt is used to convert from string to integer, giving me the exact value required from the selection, rather than needing lengthy switch statements to convert.

3.1.2 The GameMap class

The GameMap class is hugely important and responsible for a large portion of the game's operation, both for the pathfinder and AI engine. This is one of the classes adapted from the Slick2D library, though it is extremely different at this point. Many of the classes variables are self descriptive, however some are of particular note:

- **serialVersionUID** is used as a unique identifier for a serialized object. When an object is serialized, it becomes a generic object. When these objects are loaded, they must be cast back to whatever type they were initially. This variable helps check that the object being cast is the correct type.

- A number of integer values are defined related to tile types (GRASS) or units (SOLDIER_RED). These are used by the graphics package so that it knows what to draw in a certain tile.
  - In the Game class, the values related to each tile are used as an index to store retrieved images in the tiles image array. For example, GRASS has a value of 0. In Game, tile[GameMap.GRASS] will be index 0 of the tiles array, and then "res/tile/GRASS.png" is loaded into that position of the array. Later, when the graphics package finds a 0 (GRASS) in the tiletypes array (which holds the terrain map), it draws the image found at index 0. Similarly, units have a imageloc variable which matches the definition in GameMap. SOLDIER_RED is set to 4, and the RedSoldier class has imageloc of 4. When the graphics package loops over the unit locations array, if it finds a unit it checks the imageloc variable so it knows which image to draw from the tiles array. imageloc for a red soldier is 4, so it draws the tile in index 4 of tiles, which would have been set to load "res/unit/SOLDIER_RED.png" much like the grass tile was

- **turnused** is used by the AI to identify the Turn object from which a state was generated. This allows the scoring function to analyse both states and Turns.

- **tiletypes** and **unitpositions** hold the terrain grid and unit grid respectively. Tiletypes is simply a 2D array of integers: those integers are interpreted by classes such as the collision detection and graphics package. For example, the collision detection knows a 3 in this array represents a wall and acts accordingly. Unitpositions is a 2D array of UnitNodes, an inner class explained below. It tracks unit locations.

- Similarly to the above classes, **nodevisited** is also a 2D array, but one of booleans. It is used by the pathfinder to check whether a node has previously been visited in a search

During much of the implementation I used temporary, pre-defined unit instances. When the time came to implement the level editor, I needed a way to generate unit instances from a set of parameters. To achieve this, MapLayouts' createUnits function is called to build a list of unitInitialiser objects (unitInitialiser is an inner class in MapLayouts). This list is iterated through, with each object being used to create a new unit instance, which is then added to a new list, builtUnits. This new list is iterated through, and each unit's startPositionX and startPositionY variables are used to place the unit at the correct location in the grid.

A second GameMap constructor is used to clone an old one. This copy constructor, and many others, exist as a solution to a problem encountered when implementing the AI. When testing the initial attempts at AI implementation, it was observed that units were disappearing from the board after the AI took its move. These units were not being killed as their isAlive variable remained true, and the turn the AI was playing was not attacking any units, only moving. After much testing it was concluded that the cause was the method used to create new states for the AI. New states were built using “GameMap newMap = oldMap”. This meant that the new map’s references to units, unit locations and so on were still referring to the old map’s. When new turns were generated from the new map, changing the new map was also changing the original map from which it was derived. The copy constructor fixes this by taking an old GameMap as an argument, and creating a new map with its own set of object instances with which to manipulate. With the original method, when the method that creates new states from Turn objects removed a unit from a location, it was actually referring to the actual game’s unit. The method means that the object being removed from the map is seperate to the ones in the initial state.

The copy constructor was one of the biggest problems I encountered during implementation, and delayed development for several weeks since no work could be done on the AI portion of the software until the problem was found and rectified.

The UnitNode inner class is serializable, much like GameMap, since all the objects within a serialized object must also be serialized, hence it declares its own serialVersionUID.

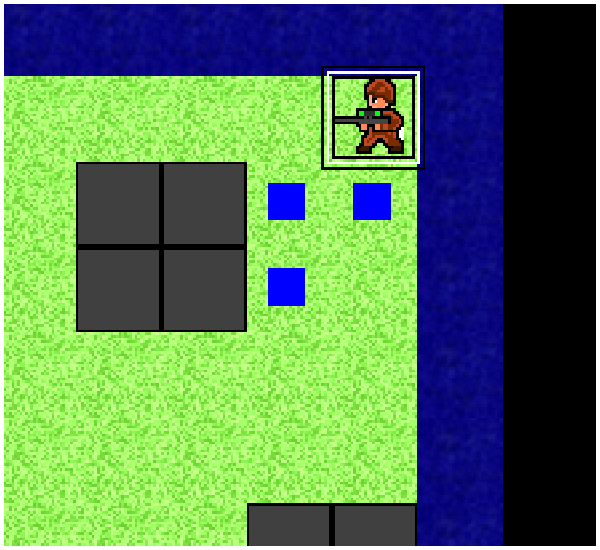
3.1.3 The Game class

This class is responsible for much of the game logic, providing several core methods, and is the longest class in the system. Like GameMap, it is based on a Slick2D class, but has been expanded to the point where it may be unrecognisable. The class opens with a number of variable definitions, one of the most important being**, buffer** an image object used to store partially built images. Since the graphics package loops through the map to decide what to draw and where, the buffer stores the game board as it is built so that the whole image can drawn in one go, rather than being drawn tile by tile in front of the user

Graphics

The class gameGraphics is responsible for drawing the current map to the screen. It creates a Panel which can be added to the Game window, on which an image representing the map is drawn. It first loops through the gameMap and uses the getTile method to get the type of tile at each co-ordinate. Since each tile is 32x32 pixels, it is then drawn at the x/y location used to check the tile type, multiplied by 32.

Following this, the same process is repeated for units. The game then checks the path variable to see if there is currently a path being stored. If there is, each tile on the path is filled with a small blue rectangle.



If the button that allows us to see a unit's attack range is activate, a seperate loop draws red squares in all the cells the selected unit can attack. A final if statement in the class creates a trio of rectangles around the selected unit by finding the unit's x and y co-ordinates, and drawing a series of smaller rectangles at those co-ordinates.

AI Functions - getLegalMoves and getLegalMovesMin

Game contains a number of functions crucial to the AI. getLegalMoves and getLegalMovesMin are used by the AI Player objects to generate a complete list of moves that the AI can perform from a given state. The process is based on **Algorithm 2** from section **2.5.1.** The system first creates a new GameMap that is a copy of the old one, so that we can modify a map without modifying the original. We also define a few lists, **candidateMoves, firedMoves, nonfiredMoves and furtherMoves.**

These lists allow us to carry moves from one loop over to the next. For example, the first loop handles moves where the player attacks. These moves are added to both candidateMoves as a potential move, but also firedMoves, which is then used by the next loop to create additional turns based on the moves already added, saving the need to re-generate moves already calculated.

Another loop that was not mentioned in Algorithm 2 is added here. If a unit has a move range of 7, and they move 5 squares and attack another unit, they may want to use the remaining 2 squares to move again. This is possible for players, so to make this possible for AI players, an extra loop exists to calculate the distance that can still be moved by subtracting the distance moved so far from the unit being moved's maximum move distance. This value is then used as a max distance to search for further moves.

getLegalMovesMin performs the same operations as getLegalMoves, however it used to generate turns belonging to the AI Player's opponent. If the min section of the AI algorithm also used getLegalMoves, none the moves assessed would actually be legal moves for the opponent, since it would be as though the opponent was trying to move the AI's pieces.

AI Functions - getNextState and doAIMove

Both these classes are similar and based on **Algorithm 3**. Get next state uses turn objects to create new states for the AI. The other method, doAIMove differs in that rather than generating a new GameMap object to manipulate, it applies the moves and attacks to the actual state in play. This method is used **only** when the AI has finished thinking.

Mouse Action Handlers - handleMousePressed and handleMouseMoved

The former of these two classes is responsible for what happens whenever the player clicks their mouse. The first thing either of these classes needs to do is to translate the window location clicked into an actual grid co-ordinate. To do this, the size of the border around the game is subtracted from the x and y co-ordinates, then divided by 32(the size of each tile in pixels). This gives us a co-ordinate we can use to interact with the grid.

The handleMousePressed class has the following responsibilities:

- Selecting units
- Attacking units
- Moving units

If a unit is selected and there is no unit in the location clicked, the listener assumes that we want to move the selected unit there, and attempts to build a path from the selectedx/selectedy co ordinates to the location clicked. If this is the first time the player has moved on this turn, moveDistanceTracker is updated to equal the selected unit's maximum move range. If a path exists from the two sets of co-ordinates, the unit is removed from its old location and placed in the new one. The distance moved is then subtracted from moveDistanceTracker. This allows us to see if the unit that was moved used all of its allowed move range. If it has not, the player's allowed move range is then updated with the remaining distance allowed so that they may move the unit again. MovedOnce is updated to equal true so that future moves will not reset the allowed move distance. Once the entire alloted distance has been used, turnMoved is set to true, blocking future movement.

The second mouse listener uses the pathfinder to get a path to the mouse pointer for a unit that is selected. The variables lastFindx and LastFindY are used to avoid constant re searching of the game map: if we move the pointer from the left side of a tile to the right side of a tile, the co-ordinates receieved by the listener will be same every time, so there is no need to recompute a new path until we find ourselves in a different location of the map.

3.1.4 The MapSerializer class

This class is responsible for saving and loading maps to file. In Java, any class can implement the serializable interface, as long as it then declares a serialVersionUID. This is necessary because when an object is deserialized by Java, we need to check that the sender and receiver of a serialized object are compatible for serialization. This is done by comparing the serialVersionUIDs. If a class does not define one, Java will create one automatically, though it is preferred to define one explicitly.[10]

The class defines two non constructor methods: SaveMap takes a GameMap as an argument (this is the map object that will be saved), and an integer, so we know how to save it (the integer represents the save slot chosen by the user). LoadMap simply takes an integer as an argument so we know which map to load. It uses a switch statement like SaveMap to decide which map should be loaded.

3.1.5 The AttemptShot class

This class implements the collision detection algorithm from the interim report (**Section 3.1.3**). The overall class was longer than expected for a number of reasons, largely to combat bugs that occured during testing. Initially the class had a single method, attemptShot, but it became clear that I would need further methods after the implementation of the AI. The standard attemptShot class uses four integers and a GameMap as arguments. The four integers represent two pairs of co-ordinates: a location for the selected unit and a location for the target.

The first deviation from Algorithm 4 comes from a set of variable definitions. It became apparent during testing that the results of the algorithm were extremely unreliable, producing unlikely shots from some angles and other angles blocking shots that should not be blocked. The image in **4.8** visualises this phenomenon. I traced part of this problem to a fault in the original algorithm: in the algorithm, calculations were being essentially being performed from the top left corner of a cell to the top left of the target, as opposed to coming from the center. To fix this, the source and target integers from the arugments are converted into doubles, and each has 0.5 added to them so that calculations simulate the center of the square.

Another problem with the version of the algorithm in the interim report was that it did not consider shots with a changeX or changeY of zero, and these would produce divide by zero errors. To avoid this, shotGradient is initialised to 0 initially, before calculating the gradient if both changeX/Y are not 0.

We use a euclidean distance calculation using the target and source co-ordinates, then find the square root to give the exact distance between a pair of units. The variable intRootShotDistance converts the result to an integer, rounding it up. We round the value up so that if a unit's attack range is 3, and RootShotDistance is 3.1, when we convert to an integer it rounds to 4 to ensure the shot is disallowed.

If the shot is within range, there are two possibilities: either the attack is a straight horizontal/vertical attack, or diagonal. If the attack is horizontal, we use the value in changeX/changeY to find out which direction the shot is travelling in. We then check every cell in that direction between the source and target for wall tiles, and if we find one, we return false.

For attacks where both changeX and changeY are not 0, we use seperate loops depending on the shot direction. For example, the first loop activates if sxdb $<$ txdb and sydb $<$ tydb. In such a case, the shot is travelling upwards and to the right due to a positive change in both x and y. We loop through every cell between the two units, and carry out the calculations described in the interim report. These calculations were also used to detect when trees are in the way of a shot. If trees are found, we add one to a counter and add this to the total shot distance to see if it has any effect on the legality of the shot.

The interim report did not specify a need to check the direction of the shot, or to only search between the source and target. This was made necessary because during testing, I noticed that shots were being blocked by walls that were behind either the source or the target, and therefore could not possibly be in the way. Eventually the reason this was happening was discovered to be that the walls were technically on the line connecting the units, even if they weren't in the way. Only testing cells between the two locations helped to fix this problem.

A further addition came from the addition of the grenadier unit in implementation. This unit was added to provide more gameplay variety, and can attack through walls. To ensure compatability, an if statement was added so that the wall calculations are not reached if we are testing attacks for grenadiers. With the grenadier, we calcalate a small circle around the unit using the euclidean distance method, then check that the target unit is somewhere between this value and the unit's maximum range.

The AI module ncessitated an additional method, testShot. This method is largely identical, however it also takes integers for shot range, and an extra pair of sx/sy values. Oldsx/oldsy are used during the check for whether a grenadier is being tested: when the AI is building a state from a turn where the player has moved then attacked, the sx/sy values for where the shot is coming from will be incorrect because technically the unit has not moved there yet (the state generation method is what performs the move). Oldsx/sy represent the original location of the unit so we can find what type of unit we are looking at.

The shot range argument is used for similar reasons: rather than looking in a cell to find a unit to grab a shot range from, we simply pass the desired range as an argument instead.

3.2 Other Notable Game Implementation

Turn

This class underwent some changes from the original design. The first was an extra set of variables to be considered when the AI player made a move, attacked, then moved again. This required four more variables to work, and also an extra constructor so turns could be created with these new properties.

AIPopup, ChooseUnitWindow and EndGameWindow

These classes were mentioned at the start of this section. AI popup is the simplest and siply defines a window that pops up with a message informing the player that the AI is thinking. ChooseUnitWindow uses a number of dropdown boxes to allow users to select a unit type, team and position, then add it to the map. When players click btnAddUnit, a new unit instance is created based on the type and team, then placed at the x and y co-ordinates in the map. Another pair of combo boxes define a location from which to remove a unit, and when btnDeleteUnit is pressed, that unit is removed from the cell specified.

EndGameWindow is called at the end of a turn if end game conditions are satisfied. The window displays a game over message and who the winner of the game was, as well as a selection of statistics that are counted as the game progresses.

The MapEditor class

Much of this class is similar to the Game class, however there are a few notable differences. A number of extra buttons exist which handle map serialization, and the combo boxes provide the information for how to save the map. There is no mouse movement listener, instead there is a mouse click listener which simply checks the type of tile in the location clicked, then cycles it to the next type and redraws the map. This allows real time changing of the map's terrain by users.

3.3 Pathfinder Classes

3.3.1 The AStar class

This class implements the A* pathfinder. The main method used by this class is findPath, which users a mover and two sets of co-ordinates to determine a legal path from point A to point B using a given heuristic to determine the cost of the path and nodes. This class is largely unchanged from the Slick2D implementation, as it appeared well optimised and produced extremely fast results from the moment is was implemented. The class uses a pair of lists to keep track of nodes which have been fully searched or are candidates for being moved to, the open list being automatically sorted to ensure the most promising move is always at the front of the list.

The main algorithm closely mirrors the pseudocode given in the interim report section **2.2:** the while loop covers step 4 of the pseudocode almost exactly, and the system is set up the same way as steps 1-3.

3.3.2 Other Classes

The other classes in the pathfinder package are mainly support classes used by the pathfinder: a pair of classes define an interface for heuristics and a heuristic itself (by using an interface we ensure that any custom heuristic created will be fully compatible with the rest of the system), other classes such as Node and Path form data structures that are used in the algorithm (Paths are returned following a successful search, and are essentially a sequence of co-ordinates that can be used by the mouse movement listener to draw the path found on the screen, while Nodes are used as part of the pathfinder itself: each node can have a parent so it is clear how that node was reached, and has a cost for moving through that type of cell (calculated using getMovementCost) and a cost for how close the node is (using getHeuristicCost)). Another class defines an interface for GameMaps, again to ensure full compatability amongst modules.

3.4 Players Classes and AI Implementation

3.4.1 PlayerInterface and HumanPlayer

The PlayerInterface's primary purpose is to allow both human and AI players to be treated the same way. They integrate the same methods for the most part, but their operation is very different. The purpose of the interface is to allow the system to use the same methods to access variables or functions belonging to AI players or human players. The currentPlayer variable in the Game class is a good example of this: it can hold either a human or an AI player.

The HumanPlayer's chooseMove method is used to enable the mouse Listener. The mouse listener is enabled/disabled by grabbing the GlassPane that exists on JPanels and setting it to visible. This acts as a layer on the window that stops the user from being able to interact with the Panel below.

By treating players as objects, I achieve two things: I do not need to create a seperate class for multiplayer and single player games: I can simply initialise two human players, or one human and one AI. It also provides an elegant method for progressing the game: we use the chooseMove method, then call the swapPlayer method to call the other player's chooseMove method. This process repeats until the game ends.

3.4.2 AIFactory

This class cats as a means to create an AI Player object based on an input integer. When a new game is created, the player chooses an AI opponent to compete against, which is translated to a single integer. This class uses that integer to decide which type of player object to create.

3.4.3 AI Implementation

The various AIPlayer classes represent the implementation of the game's Artificial Intelligence engine. As explained in the design, I have used a Minimax algorithm to create the AI, however it has also been optimised to produce better performance. These optimisations are essential: on a simple game with few possible moves at each stage, the number of potential turns and states generated is low. Even at extreme search depths, the generated states will number in their hundreds. These games can therefore reach extreme depths very quickly with little optimisation. This is not the case with my game:

- The typical soldier unit can move 5 squares in any direction, in any combination of moves (left, up, left, up, up for example)
- On a blank grid, this means there are 60 possible moves the player could make, not including attacks
- If the player could attack, you could have the 60 potential moves, and maybe 10 could produce attacks as well, giving us 70 possible moves. If we can attack before we move, we gain the attack, plus every move that could be made following that attack, creating another 60 moves for a possibility of 130 possible moves from a single unit.
- In reality, walls will be blocking the unit's progress, and situations where a unit can attack before moving are rare, so for the purpose of the example, we will use 50 as the average number of turns generated by a single unit
- An average game features 3-5 units per side. If we take 3 as a best case average, then we get 150 possible moves at the initial search level
- When we advance to the next level of the search, we need to consider which moves are now possible as a result of those moves. Each of those 150 moves could produce another 150 moves of their own, for a total of 22500 potential moves
- When we flip back to the initial player's moves, we might produce another 150 moves. This brings the total number of states generated to 3,375,000 at just 3 levels of search depth

With numbers like those, it is clearly essential that we optimise the algorithm so that not all of these states need be searched.

The optimisations built into the algorithm are:

- **Alpha Beta Pruning** - alpha beta pruning is used to help find an optimal solution while avoiding subtrees that will never be selected. This has been explained in the interim report (**Section 2.3**).
- **Iterative deepening** - this is a method used to improve a search by searching to continually deeper levels. We set a maximum deapth and search to this depth. If no solution is found by this point, we increase the

search maximum depth and search again. It allows us to reach the same result with less memory usage, however we discount all the computation done previously if we use this method.[2]

- **Aspiration search** - this allows us to improve the alpha beta pruning by considering only a small search window. Normally in alpha beta pruning we initialise alpha to negative infinity and beta to positive infinity. However, we can use the score from the previous depth level to estimate a better alpha and beta. If s is the best score found at the previous search level, and we define a window w, we can define alpha as $s - w$ and beta as $s + w$. With a small enough value for w a large portion of the game tree can be pruned. We have to ensure we set this value properly, as if $s* \geq s + w$ or $s* \leq s - w$ where $s*$ is the best score found at the current depth level, we may have pruned too much of the tree, and we will have to search again with the normal initialisation of alpha and beta. This causes us to lose time, but through trial and error we can find a value for w that ensures this rarely happens and make the overall algorithm faster

The implementation of the alpha beta algorithm is largely similar to **Algorithm 1.** The methods used to create new turns and states have been discussed in section **3.1.4.** Since iterative deepening could potentially run for a very long time, we define 3 variables, timeStart, timeNow and timeCheck. Before we start the search timeStart is set to the current system time in milliseconds. At the beginning of the search timeNow is also initialised to the current system time, and timeStart is subtracted from it and stored in timeCheck. If the result is larger than 5000 (5 seconds) after we begin anew, the search breaks.

Also before searching we set initial values for alpha and beta using integer.MIN_VALUE and integer.MAX_VALUE, and define our window, which represents w. If we have found a bestScore, we can use that to create estimates for alpha and beta, which are stored in aspAlpha and aspBeta. If after considering every turn in AIMoves, we have a bestScore outside the aspiration window, we have to re search with standard alpha and beta initialisations, which will take significantly longer.

State Evaluation

The method used to evaluate states has changed slightly from the design. In the design I had ideas such as the following: "If the move has left the player in a position where their opponent could kill one of their units on the next turn, subtract a medium number of points." I realised this was unnecessary: by the nature of minimax, such things should naturally be taken into account anyway as situations where the player leaves themselves open to attack will be avoided naturally if the penalties for losing units is large enough. This left me with a much cut down evaluation function, where states are assigned scores based on two methods.

The first method adds or subtracts a score based one whether the player has more or less units remaining than the opponent. Following this, we add an amount to the evaluation based on how many units the player has alive, and what type of unit they are. The second method also adds and subtracts a score based on unit advantage, but then adds points by killing enemy units, resulting in a more aggressive play style.

It also became apparent that I needed a pair of evaluation statements: one for min turns, and one for max turns, similarly to how I needed two methods to generate potential turns. If I used the same evaluation function for both players, when examining an opponent's move, the AI controlling blue units would think that a good situation for their opponent was having more blue units than red units. Therefore, two methods are used, with slight differences whenever specific teams are involved.

The average evaluation therefore works as follows:

- An excessively large number of points is added for finding a winning state and subtracted for finding a losing state
- Having more units than the enemy results in a large points boost, having less results in a large deduction
- A number of points is added for each friendly unit that is currently alive, with snipers etc worth more points than a soldier
- OR
- Killing enemy units gives a large number of points depending on the type of unit killed
- A random amount is added to the score
  - Easy opponents have larger random values. This helps make bad moves look more attractive to the AI, and good moves worse. The larger the random value, the more pronounced the effect
  - Easy opponent search to lower depths
  - Easy opponents deliberately have evaluation functions that evaluate bad situations as being good situations, such as having no penalties for losing units

Some AI opponents have conditional behaviour. These are different scoring functions that are used in certain situations. These often triggered using an effective strength mechanic:

1. Using the numerical score value for each unit, calculate a score for each team
2. Subtract the enemy's score from the AI player's score and store as effective strength
3. Initialise a window (for example, 100)

4. Multiply the window by the total number of units in play
5. If the effective strength difference is positive and larger than the window, trigger aggresive conditional behaviour.
6. If the effective strength difference is negative and smaller than the inverse of the window, trigger defensive behaviour

For example, if the window size is 500 and the AI's strength is 1100 and the enemy's strength is 700. 1100-700 is 400, so the AI does not have a big enough advantage to start using a more aggressive evaluation function.

If the window size is 500 and the AI's strength is 1000 and the enemy's strenght is 1600, the effective strength is -600. The inverse of the window is -500, so the defensive behaviour will trigger because we are outside the window.

I also included an alternative method late in development. This method used a scale to adjust how large the bonus or penalty is for having more or less units remaining. This method used a flat value of 100, multiplied by the the difference between player unit counts multiplied by the number of units the player has lost. For example, if the player has 2 units less than the other player reamining, and has 4 left having started with 8, a scale value of of 800 will be produced. This value is then subtracted from the reward/penalty for having more or less units. This means that the further a player falls behind or pulls ahead in terms of numbers of units, the more noticeably offensive/defensive they will play.

Interface Adherence

Like HumanPlayer, the AI Players also use a swapPlayer function. This is largely similar to the one in HumanPlayer, except after updating unit positions, it performs the bestMove returned from the chooseMove method using the doAIMove method, then flip control of the game to the opposing player.

3.5 Units Classes

Every unit in the game implements the UnitInterface. Because of this, units can be passed around by the game through numerous methods no matter what type of unit they are. This makes adding new units to the game very easy, as demonstrated by the addition of the grenadier class. Each unit type has a few variables to help make it distinct, such as teamID, moveDistance, shootDistance, and so on. Units can be initialised as a default unit without arguments, using another unit as an argument to clone that unit (essential for cloning GameMaps), and also using a unitInitialiser.

During testing I noticed a tendency for games to turn into cat and mouse affairs where units hit behind walls endlessly. I created the grenadier unit to help break these stalemates: the grenadier can attack through walls, but cannot attack units directly next to him, and only over a limited range. These units added an extra layer of depth to the gameplay described in the next section.

4 Results and Evaluation

This section examines the extent to which I achieved the goals set out at the start of the project. I will address each goal in order, the criteria for its achievement, and how well I feel the goal was achieved. I feel I have achieved every goal I initially set out to achieve. Not every test can produce a qualitative result, some outcomes must simply be tested through continuous use.

4.1 A two dimensional grid on which units can move around

Testing that this goal has been achieved is relatively simple:

1. Players should be able to click a unit, then click another cell in order to move the unit there
2. The only situations where a unit cannot move should be:
    (a) The destination is already occupied by another unit, enemy or friendly
    (b) The cell is outside the maximum move range of the selected unit
    (c) The destination cell is a wall or water

It is clear from testing that the above criteria are all satisfied: by playing the game repeatedly, moving units about and attacking others, we can see that any cell that a unit should legally be allowed to move to based on the rules of the game can be moved accordingly. These tests cover all of the grid's fundamental functionality in a few tests. The same also applies to AI players: the AI does not play moves that an equivalent human player would be prevented from making. This is because the algorithm from which potential turns are generated uses the same pathfinding and collision detection classes as are used when a player chooses a unit to move or attack.

4.2 Different grid cell types

Whilst this goal also had three sub goals, I will first look at how the overarching goal was achieved. As noted in the design, I planned for four different types of grid cell, and the system implemented this almost exactly as the design specified: different tiles are represented by a different integer in the range from 0 to 3 and other methods simply check the integer to see how they should handle that tile. The cells behave differently: players can move over grass, but not over walls, and so on. As a result each tile feels suitably different, and none feel superfluous or merely cosmetic.

4.2.1 Users will be able to generate their own designs to play on

1. Users should be able to make a custom map from scratch
    (a) Custom maps should be customisable in height and width

    (b) Users should be able to manually place units on the map
    (c) Any map that is pre-provided with the game should be recreatable in the editor

2. Users should be able to play the custom map against other players or AI
3. Users should be able to save maps to their computer to reuse at a later date

The implementation described how this goal was met: clicking on cells in the editor cycles through tile types, a dialogue box is used to add unit instances at specific locations, and so on. Custom maps use the same structure as regular maps, making them completely compatible with the AI and pathfinder, rather than requiring additional implementation. Java's serialization interface allowed the saving of maps to a computer for later use.

4.2.2 Grid squares will be visually distinctive

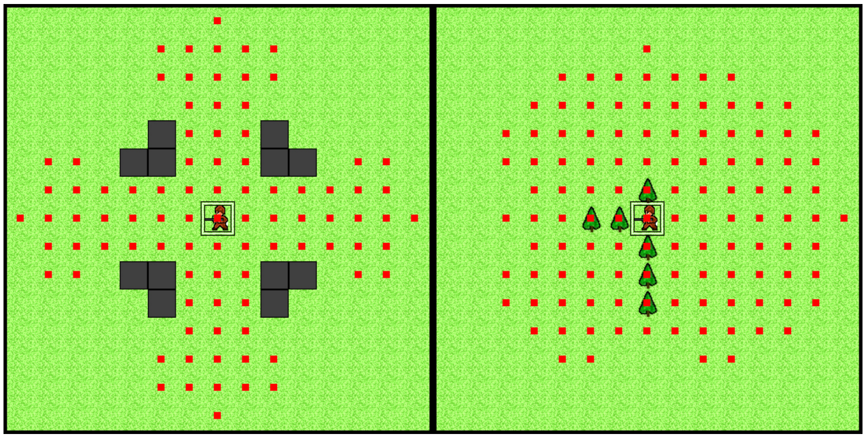
The design of grid squares was covered in the design section of this document: each was given a visual design that ensured each was clearly distinguishable thanks to colour or contents. The exact images shown in the design section were used for the in-game graphics.

4.2.3 Players should be able to shoot at any angle and the game will calculate obstructions

This functionality is achieved as a result of the collision detection module in the game. As noted in the introduction and design, this turned out to be one of the main problems that required solving during the project. During the initial conception of the project I considered having only 8 directions of attack, but decided this was not condusive to good gameplay and would lead to cat and mouse situations where two units continually duck out of the way of each other's attack lines.

Whenever a player has a unit selected and clicks a unit belonging to the other team, the collision detection is called, and loops through every cell between the source and target. Cells containing grass or water are simply ignored. Whenever a wall is tested as being on the line of attack, the algorithm immediately returns a value of false. If a tree is found, the same calculations for detecting if a wall obstructs a shot are used, but the algorithm adds 1 to a counter rather than returning false. Once every cell between the source and target has been checked without finding a wall, the number of trees is added to the shot distance, and if the shot fails this final test, a value of false is also returned.

The algorithm will only return true once every single cell that could potentially block a shot has been tested and trees have been taken into account. To check consistency, a number of symmetrical maps were created with a unit in the center to ensure that all the shots that were listed as blocked were the same in every direction, and appeared logical. This image can be seen below:



4.3 Several types of unit

My initial design covered plans for three types of unit: the soldier, who occupied a 'jack of all trades, master of none' role, and the commando and sniper, who occupied more specialised roles with clear strengths and weaknesses. Each of these three unit types were implemented exactly as they were in the design, along with the balance considerations (a unit that attacks a long way must not be able to move a long way as a means of balance), but during implementation and testing I decided upon adding a fourth unit type to improve the variety of gameplay and also avoid stalemate situations where neither player would move units from out of protective cover.

This new unit had to be balanced differently to the other types of unit, as its strength was noticeably different. Rather than having a bonus in move range or attack range, it had the unique ability to be able to attack through walls. As the only unit capable of this, it had the potential to be very overpowered. This potential was mitigated by a number of balancing features. The unit's move range was made to be one shorter than that of the soldier. This allowed him a good range of movement options, but was low enough that commandos would easily be able to close the distance on these units, and snipers would be able to kill them easily if they crossed into their field of view. An additional balance step was made in introducing a minimum attack range for the the grenadier. The grenadier cannot attack a unit positioned in a cell directly next to its own: this simulates the idea of a grenade attack, as throwing a grenade too close to one's self risks severe injury. This makes the grenadier seem more realistic, and also cements its role a support unit: it is poor in straight up combat with any other class, but can be used to force overly defensive players to abandon their safe positions.

4.4 Multiplayer Functionality

This was one of the first goals I achieved during implementation, even before pathfinding and the collision detection were fully implemented, I successfully had two different teams of units taking turns to attack each other. The test to ensure that this was achieved was simple:

1. Only units belonging to the current player can be selected
2. Only units belonging to the opposing player can be attacked
3. After a turn finishes, control switches to the opposing player

We can see how this was achieved by looking at the implementation of the mouse listeners, which handle moving, selecting and attacking units. As a result players can only move and attack units they are supposed to be able to. Switching players is handled by having players represented by player objects which can call the opposing player's chooseMove method.

4.5 Robust pathfinding

Pathfinding was outlined in the interim report and this report's design section as one of the main challenges to solve in the project. As noted in the design and implementation, the pathfinder code I used is based on a number of classes from the slick Java game library. These classes were adapted to work with the continually evolving design of the game, and classes such as GameMap are almost unrecognisable compared to their implementations in the slick library. The slick library A* pathfinder suited my needs as in the initial report I identified that I felt the A* algorithm would be the best way to achieve a fast and accurate pathfinder. Further requirements for the pathfinder stipulated that the pathfinder should always find a valid route between two points (valid in terms of the rules of the game), and also be fast enough in computation that the path from a unit's position to the location of the mouse pointer can be displayed in real time. The criteria to test this were:

1. Any cell within the legal number of moves for the selected unit should be reachable if it is not blocked by a wall or water
2. The path should be displayed on the screen without delay whenever the position of the mouse changes cell

The first of these criteria was tested by extensive play of the game. Whenever I selected a unit and tried to move it somewhere, but the game would not compute a path to that point, I manually counted the squares between the source and the target. I never encountered a situation where the pathfinder had made a mistake. The second criteria was tested by selecting units in any game, and rapidly moving the mouse about the screen. Despite the constant need to update the path calculation, the game never slows down or crashes when I do this, and displays the correct path without any noticeable delay.

4.6 Artificial Intelligence

The artificial intelligence section of my initial goals had related sub goals defined, so I tested each of these individually.

4.6.1 AI Difficulty settings

I achieved adjustable AI difficulty settings by creating the different AI presets for players to play against described in the implementation. Through adjustments to the scoring methods, different styles of AI can be created: high scores for taking units and low penalties for losing units results in aggressive players, and the reverse results in defensive ones. Situational behaviour is triggered by the effective strength calculations. Easy players had restrictions on search depth, had deliberately poor evaluation functions (high scores for taking units but no penalty for losing them for example), no situational behaviour and large random score changes. Medium removed the depth limit, and used more logical evaluation functions, added situational behaviour and reduced the size of the random addition. Hard mirrors medium, however the situational behaviours are easier to trigger, and will also apply the situational behaviours to the player, meaning the AI can react to how a human player might play in different situations. The differences in the way the AI opponents play are noticeable, which makes the game accessible to players of any skill level. However, with more time to work on the project I would have liked to spend more time differentiating the way different AI opponents played.

The alternative method for situational behaviour using a scale implemented in AI_MEDIUM_Standard_Other produces slightly less pronounced behaviour differences. This could be improved by changing the initial multiplier on the scale from 100 to a larger value such as 250 to make the differences more notable. The performance of this method is also somewhat dependent on the game having large amounts of units, since if only small amounts of units are present the difference in unit numbers will be small. A further weakness of this scheme is that it favours simply having more of any type of unit left, rather than treating individual units as being more valuable than others. Because of this I stuck with the initial design of the evaluation, but kept this method as an option, since it could be expanded in the future to be more effective.

4.6.2 AI thinking ahead

Minimax is a recursive algorithm, and the appearance of thinking ahead is achieved through looking at every single possible state that can be reached from the current state, all the states reachable from those, and so on, before evaluating each of the possible states and choosing the move that would result in the highest score for the AI and lowest for the opponent. I made good progress in optimising the AI to allow it to think further ahead, though I feel that with more time to work on the project I could have achieved a system that would allow the AI to think to further depths, which would be useful on larger maps such as bridge and fortress

4.6.3 AI mistakes/situational behaviours

A key feature of making the AI seem more human was to have it make mistakes and react to game situations. In 4.6.1 I described how different scoring functions can effect how well an AI opponent plays. These also form the basis of making the AI seem more human: a human opponent would recognise that they were currently losing, and adjust their tactics. Having different scoring functions activate according to certain situations allows me to emulate this kind of behaviour. The random score fluctuation allows the AI to make mistakes. For example, imagine a bad move scores 250 points, an average one 500 and a good one 750. With a random fluctuation of 250, the order of these moves can be completely changed: the best move could lose 100 points and be left on 650, meaning that if the middling move gains 150 or more points, it will be played instead of the best move. The larger the potential change, the greater the chance the AI will make a mistake. This can be displayed by adjusting the random fluctuation so that it is always extremely large: doing so makes the AI appear to play completely randomly.

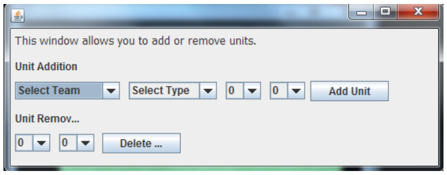
4.6.4 AI response times

As a functional requirement I listed that the AI needed to make moves within 5-10 seconds. To achieve this I implemented iterative deepening as part of my AI optimisations; this is discussed in the implementation section of the document. Iterative deepening selects an initial search depth to search to, and returns a result, then checks the time elapsed since the search began. If the search is still within the time limit, it will progress to the next possible search depth. This means that the algorithm will always be allowed to completely search the depth it is currently at rather than being cut off. In most cases, the AI finishes its search within 5 seconds, occasionally taking slightly longer due to having started a new search at, for example, 4.5 seconds. Occasionally a search can take significantly longer than the 5 seconds I aimed for, and this is something I would hope to rectify in the future through additional optimisations.

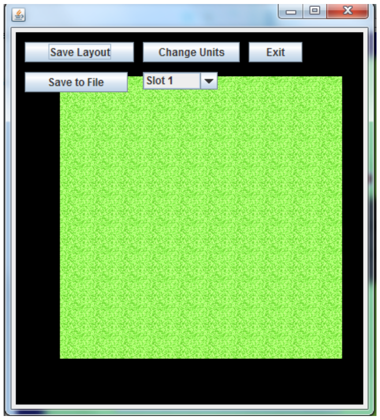
I mentioned during the implementation that at a depth of 3, there could be up to 3,375,000 possible states. Frequently my AI players reach a depth of 4+, which means potentially 506,250,000 game states have been considered by this point. This is proof of the effectiveness of the optimisation work carried out.

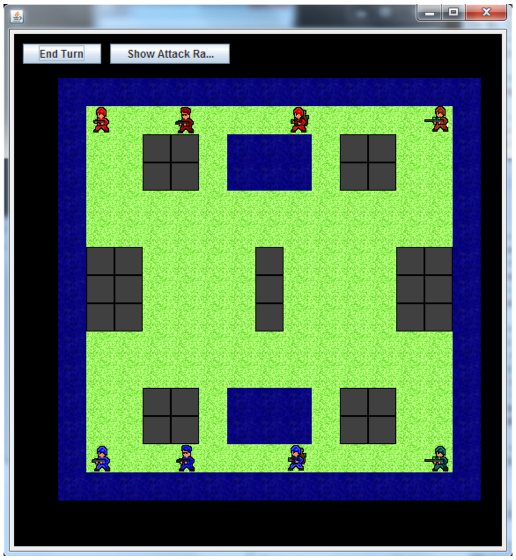
4.7 User friendly interface design

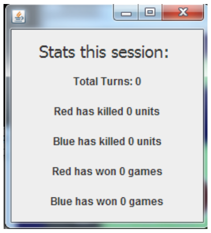
The design section featured a number of hand drawn interface designs and explained why I felt they were good, user friendly layouts. During implementation I stuck very closely to these designs, and as a result I feel that I achieved a good user Interface. Every button's function is clear, and dialogue boxes appear before major decisions, which allows users to back out if they make a mistake. In the design I described what made my designs user friendly, and the images below show how closely I mirrored the design in the final implementation:



This window allows you to add or remove units.







Stats this session:

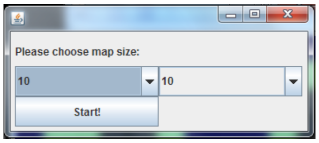
Total Turns: 0

Red has killed 0 units

Blue has killed 0 units

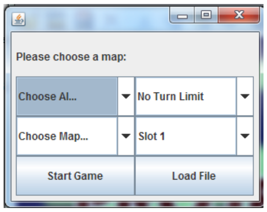
Red has won 0 games

Blue has won 0 games


Please choose map size:

10

10

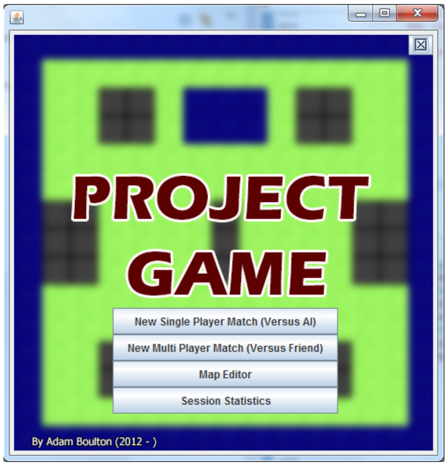

Please choose a map:

Choose AI...

No Turn Limit

Choose Map...

Slot 1


PROJECT
GAME
New Single Player Match (Versus AI)
New Multi Player Match (Versus Friend)
Map Editor
Session Statistics
By Adam Boulton (2012 -)

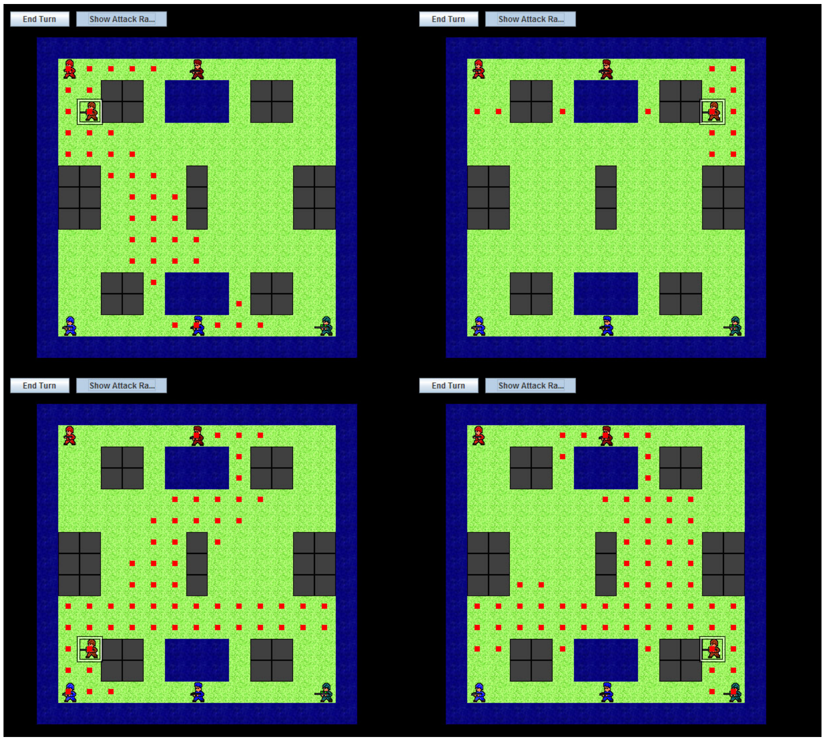
4.8 Overall Project Success

I feel that for the most part, the project progressed smoothly, however there were a number of hiccups, many of which have been described within the implementation section. In these cases I eventually found my way around these issues, but certain factors may have made them more difficult to overcome. Whilst I had a good understanding of the Java language when I began and finished with what I believe to be a much more comprehensive knowledge of the language, certain aspects of the language made progress difficult at times. A major sticking point was the times before I realised I needed to create and use copy constructors. Before this, whenever the AI would make a move, units appeared to randomly be disappearing from the map. It was only after thoroughly examining my code and reasoning that the move and state generation algorithms were having an effect on the actual game board that I was able to fix these bugs. Unlike languages such as C++ where copy constructors are implicit in the language, in Java every object that is manipulated is a handle, so copies need to be defined explicitly.[3] This was a major sticking point for me and held up development significantly until the problem was solved.

Another issue came from an initial misunderstanding of how static and non static variables operated in Java. Early on I made a number of variables in the GameMap class static, including the map width and height, which caused problems when I implemented the map editor: since static variables belong to the class rather than an instance, if I had an edited map that was a different size to the default map size there were problems due to the GameMap copy constructor getting the map size wrong, causing out of bounds errors. After these problems, I made sure to browse through my code and eliminate any potentially problem-causing static variables that I had created earlier in development.

I feel my chosen methodology allowed the project to proceed smoothly: I always knew exactly what goals I was working toward, and I often had several small tasks on the go at once: not so many that I was overwhelmed with work to do, but enough such that if working on one area of the project started to cause frustration, I could take a break from it and work on something else instead.

Further problems were encountered during the implementation of the collision detection algorithm. Despite the maths behind the algorithm being solid, on initial implementation the algorithm exhibited some extremely strange behaviour, which extremely inconsistent targeting calculations.



We can see from the above image the the selected unit's possible attacks were wildly inconsistent depending on the direction of the attack. Those screenshots were taken some time after the intial problem I encountered: wall tiles found behind the attacking unit or behind the unit being attacked were technically falling on the line between the two units, and were being interpreted as blocking the shot by the collision detection. The fix for this problem was described earlier in this section and in the implementation, and even lead to a slight performance improvement as there was no longer a need to check every cell in the grid. The above problem was caused by a number of errors: in the top right example we can see the pure horizontal shot to the right is passing completely through walls. This was caused by using a $<$ in a calculation rather than a $>$. The rest of the inconsistencies were caused by the calculations for where a unit could see from were running from the top left of a cell rather than the center, explaining why in the bottom left image, the unit can attack so far to the right. A further error was that there were a few calculations involving division that would result in decimals in the process, but the results were being stored in integers. These variables had to be changed to doubles, and since the algorithm took integers as arguments, these arguments also had to be cast to doubles.

For the most part however, I was both very happy with how the project progressed and with my final results.

5 Future Work

Whilst my project has achieved much of what I intended to from the beginning, there are still a number of features that I would like to have included if I had had more time to work on the project. Certain parts of the project took a lot more effort to implement than I expected, but on the other hand, some parts were a lot smoother than others. Some development slowdown was encountered by the fact that making changes in one area of the code can have unexpected changes in others: this can be seen from when I realised I needed to implement copy constructors to clone objects, for example.

The first future improvement would be to change the implementation of tile types so that they are class based, much like units. This would require overhauling a reasonable portion of the GameMap class, but I think it would make the game more expansible in the future. Due to the way the game is coded, adding new tile types could currently be difficult, as I would need to adjust a large number of classes to fit the new tiles in to the game. Changing to a class based tile structure using an interface to define a generic tile object would make the game easier to develop in the future. One idea for a new tile type would be a hill or a mountain: standing on a hill tile could increase a unit's attack range or allow him to shoot over nearby wall tiles, but units not on a hill tile would find they block shots much like a wall would.

The game uses the Java serializable interface to save maps to the user's computer for use later. Whilst this is a quick and easy way for users to save maps, a major limitation is the number of slots available. I felt 5 was a reasonable number of slots, and this number could easily be expanded, however adding a full save dialogue would allow an unlimited number of saved maps. To do this I would likely need to find a way to create my own file type: possibily a text based format that could be parsed to build a GameMap object, similarly to how I initially planned to implement saving maps. A further improvement of saved maps is currently there is option to edit a saved map once the map has been closed. Allowing users to open saved maps in the editor as well is in the game proper would make the game a lot more customisable.

The current map editor is great for building maps, but it could still be a more user friendly experience. The MapLayouts class has a function that allows me to fill an area with a selected tile type. Editing large areas of terrain can be quite time consuming in the map editor, but a flood fill tool could cut down the amount of time required to edit the map considerably. I could adapt the method in MapLayouts that allows filling for use in the map editor, and possibly implement it as a seperate pop up window to the add units dialogue. Users could specify a type of tile, and a start x/y co ordinate as well as a finish x/y co ordinate, and everything between those points would be filled with the specified tile type.

There are a number of possible gameplay developments I could implement in the future. The first could be to implement a much wider range of unit types to allow players different ways to interact with the game's environment. Airborne units could have the unique ability to travel over both walls and water tiles. Similarly, I could implement water based units that can only travel on water. Certain ground and water based units could be very effective against air units, but weak against other types of units.

An expanded range of units types could work well with a unit health system. Currently, when a unit is attacked it is immediately removed from the game, but a more interesting mechanic might be to have each unit type deal a certain amount of damage to a unit's health (say each unit has 10 hit points): a soldier might typically inflict 7 damage on another soldier, and only 2 to a water based unit. Units with less health would become less effective, which would open more possibilities for unit types: medics could help units regain health, and engineers could repair vehicles. Of course, I would have to be careful not to over complicate the game, as having too many units could frustrate new players.

Another possible future mechanic could be a 'fog of war'. I already have a means of calculating whether one unit can 'see' another, and I could potentially adapt this so that at the start of any turn, I calculate all the cells that the units belonging to the current player can see into based on a vision range variable. Cells that none of a player's units can see would be greyed out so that the player cannot see what is in those cells. This could enable far more tactical options, however it would be a major challenge for the AI implementation, as it would switch the game from one with perfect knowledge to one with imperfect instead.

Currently players can only move one unit per turn, but an idea for an alternative mechanic I had revolved around 'move tokens'. Each player would start the game with a certain number of move tokens based on the map selected, and then be awarded a flat rate of tokens at the beginning of each turn, with bonus tokens awarded for doing certain things in the game. Moving a unit would consume a move token, as would attacking. A player could choose to use all their move tokens in one go, or save their move tokens to use in one large attack later on. This is definitely a mechanic that I would implement in the future.

Multiplayer can currently only be played by two people at the same computer. In the future, it would be good if two players could compete over a network instead: an easy way to do this could be derived from the map saving mechanics: since it is possible to save a map's state to a machine, in theory a state could be saved and sent to another player and then used for the opponent to make a turn, before being saved again and sent back. This would also make a fog of war mechanic more feasible, since if two players are sharing a screen, one would have to look away whilst the other is making their move or they would see where all of the opponent's units are located anyway.

There are some slight graphical issues in the game I would like to fix: if a player uses their full move allocation, then attacks, the game board will not redraw until the AI has finished thinking. Similarly, when the AI is thinking, the "AI is thinking" message that appears does not show the "Please Wait" label like it is supposed to. These problems could be fixed by running different computationally expensive tasks in different threads.

I would like to experiment with alternative state evaluation methods. One such method could be to use changeable material calculations: for example, a grenadier could have a flat unit worth of 500, but increased by 100 for each friendly unit still alive. A commando's score could be increased by the number of cells he can move to, and a sniper's score could be based on the number of cells he can attack from his location. Additional score could be added based on the number of potential moves that can be made from the state. We would calculate each team's score seperately, then subtract one from the other to get a final score. This would give a material based score, as opposed to the advantage-based system used currently. A system such as this would be a lot more computationally expensive however, especially as we increase the numbers of units involved in a match, and would require further optimisation of the AI. This would be a worthwhile effort as this would likely improve the AI's behaviour at the opening of games, and in situations where it cannot reach a high search depth quickly.

Finally, I would like to build in more customisation options: for example, the AI has a few variables that can be changed to adjust its behaviour, such as the maximum search depth, or the size of the random fluctuation. It would be interesting if human players had the opportunity to adjust these values using an interface to create customised opponents. Similarly, it would be interesting if players could change the amount a tree reduces attack range, or play with the maximum move ranges of units to create a different style of gameplay.

6 Conclusions

The aim of this project was to create a multiplayer strategy game playable against other humans or an AI controlled opponent. As a result of this project I have learned a huge amount about a number of topics. I feel that my programming knowledge and practice has been massively improved: I now have a much wider knowledge of the features Java offers, and have found practical use for a number of design patterns and features of object oriented programming with which I previously only had a theoretical knowledge of. With this project finished, I am already thinking of ideas for future coding projects, such as a hugely expanded android port of the game with all the future possibilities described in section 5.

I have created an AI opponent that achieves everything I wanted it to: it "thinks ahead" in the game, allowing it to show a deep understanding of the game and its mechanics, as well as show tactics and strategies players might not normally consider. The AI is customisable thanks to being able to change scoring functions and alter move depths, allowing several different styles of opponent to be created. The AI also displays human-like behaviour thanks to the random fluctuation of move scores and the situational behaviour triggers.

I wanted to have a game that featured a range of unit types and terrain types in order to produce varied and interesting gameplay. I achieved this and ended with four different types of unit and four different types of terrain. I even demonstrated during the implementation the expandability of the unit systems, meaning that I could add many more in the future if I wished.

I implemented a pathfinder that worked exactly as I hoped: it always find a legal route between two points if one exists, and it does so extremely quickly with no noticeable impact on game performance. The same can be said of my collision detection: it is a fast and efficient algorithm that produces accurate results time and time again. The algorithm's speed can be see in its repeated use in the AI algorithms to produce thousands of potential states in a very short space of time. A less efficient algorithm would almost certainly have slowed the AI player to a crawl.

In summary I feel that my final year project was a great success; I achieved everything I set out do from the beginning, learned a huge deal in the process, and also left with a range of potential future developments, showing that the finished project has near endless potential for growth and adaptation.

7 Reflection on Learning

I believe I learned a lot during this project. During my second year of university it is fair to say that I struggled a great deal during Java programming assignments. As a result of the knowledge gained during this project, I feel that I would enjoy much more success during programming assignments, such as the perfect hash function coursework in Algorithms and Data structures, as well as the majority of the Object Oriented Applications module. I feel as though I was able to approach programming assignments with far more confidence this year than in any previous year of university, and began to enjoy them rather than dread them, putting them off until as late as possible. By looking at the code it may be possible to see which modules I coded first: for example, the AI Player object factory was created late into this development, and as a result it is a fairly simple piece of code, compared to the unit creation method which is more convoluted (using unitInitialisers) due to early parts of the implementation becoming a hindrance later on.

I learned a lot about approaching and solving problems. Previously I had been skeptical of the importance of thorough design, as I felt it might restrict creativity later on in a project. However, with the thorough design I settled on during the project, I am certain I would have encountered far more problems if I had jumped straight into implementation without fully planning the system first.

I mentioned that I had a few assumptions on which my work was based. With implementation finished, I think these were good assumptions. I assumed a human player would have limited patience and would get bored if the AI took too long to choose moves. This resulted in a greater focus on optimising the AI so it could find strong moves as quickly as possible, and if I had not made this assumption, I may not have spent as much time working on this area. A second assumption was that a perfect AI opponent would not be much fun to play against, as it would always beat the player. A tough opponent that a player knows is beatable is much more fun than a perfect, unbeatable one, so I implemented systems such as the random move score fluctuations and different AI opponent presets.

References

[1] Amy S. Biermann. Minimax and alpha-beta template. Website. http://www.pressibus.org/ataxx/autre/minimax/node2.html.

[2] Paul Brna. Iterative deepening. Website, January 1996.

[3] Bruce Eckel. Controlling cloneability. Website, March 2001. http://www.codeguru.com/java/tij/tij0128.shtml.

[4] Kevin Glass. Path finding on tile based maps. Website. http://www.cokeandcode.com/index.html?page=tutorials/tilemap2.

[5] Neil Burch et al Jonathan Schaeffer. Checkers is solved. *Science Magazine*, 317:1518–1522, 2007.

[6] Max Kanat-Alexander. What is software design. Website, February 2008. http://www.codesimplicity.com/post/what-is-software-design/.

[7] Mitchell Kapor. Bringing design to software. Book, 1996. http://hci.stanford.edu/publications/bds/1-kapor.html.

[8] Liam. Slick2d. Website. http://www.slick2d.org/.

[9] Relpha Morelli. Cpsc 352 artificial intelligence notes: Minimax and alpha beta pruning. Website. http://www.cs.trin-coll.edu/ram/cpsc352/notes/minimax.html.

[10] Oracle. Interface serializable. Website. http://docs.oracle.com/-javase/7/docs/api/java/io/Serializable.html.

[11] Paulo Pinto. Minimax explained. Website, July 2002. http://ai-depot.com/articles/minimax-explained/.

[12] James Roper. The importance of good software design. Website, February 2004. http://cs.anu.edu.au/ Alistair.Rendell/Teaching/mdtutorial/node11.html.

[13] Hugh McCabe Ross Graham and Stephen Sheridan. Pathfinding in computer games. Article. http://gamesitb.com/pathgraham.pdf.

Appendices

A Initial Requirements:

- A two dimensional grid on which units can move around
- There will be several different types of grid cells, some which will allow movement or shooting through, and some that will not
  - Users will be able to generate their own grid designs or play on a pre designed or saved one
  - Grid squares will be visually distinctive for ease of use
  - Players should be able to shoot at any angle and the game will calculate if the shot is blocked
- Several types of unit
  - Each type of unit needs to be balanced against the next - a unit that can cover a large distance in a move should not also be able to attack over long distances, and a unit that can only move a short distance should be able attack over long distances in order to compensate
- Multiplayer functionality
  - Two players should be able to take turns in order to attempt to defeat the other team's units
  - A score system should allow players to track scores and statistics: total wins, total moves, total units defeated, and so on
- Robust pathfinding
  - A pathfinding algorithm that always produces a valid move between two points. Moves will be restricted because some tiles will not allow units to pass through them, and each unit will have a maximum move distance, making some locations impossible to reach.

  - The pathfinder should be efficient and have a low computing cost

- Artificial intelligence
  - The AI should have multiple difficulty options that can be customised before matches, with a few presets such as an "easy" or "hard" setting
  - The AI should be able to think several moves ahead
  - The AI should have human traits such as a chance to make mistakes, or different levels of aggression depending on the current game situation

- Further features
  - Visually distinctive units and tiles
  - A user friendly interface

B Additional Design: System Requirements

B.0.1 Functional Requirements

- **Gameplay Requirements**
  - Players should be able to select any unit that is on their team, and move them by clicking another tile
  - Different types of grid tile should not just be cosmetic but actually have a tangible effect on gameplay
  - Multiple unit types, with their own move ranges and attack ranges to make them distinct
  - Units should be balanced: unit A should be good at defeating unit B, and unit B should be good at defeating unit C.
  - Players should be able to preview what squares each unit can attack at any time, rather than being forced to rely upon trial and error to find a legal move
  - Players should be able to end a turn by clicking a button, or have a turn automatically end once a player has both moved and attacked during their turn
  - Players should be able to play matches "to the death" or to a set limit of turns, chosen by the player
  - The game should proceed in a turn based manner

- **Non Gameplay Requirements**
  - The game should track statistics as games are played, such as moves made, units defeated, games won, and so on

  - The game should have a reasonable number of pre designed maps for players to play on to ensure a variety of different situations/environments for games
  - Players should be able to design their own maps on which to compete against other players
  - The AI should be able to play as well on custom maps as it does on premade ones
  - Customised maps should be able to be saved between gameplay sessions and not lost on shutdown

- **Background Functionality**
  - Pathfinding should be calculted "on the fly", once a unit is selected the pathfinder should be able to display a legal path to wherever the mouse pointer is on the board without a noticeable period of computation
  - Collision detection for shots should be consistent: if a unit is placed at a central position on a completely symmetrical map, the same squares should be blocked in all directions
  - Collision detection for shots should be logical: situations where a player feels they should be able to attack a tile need to be kept to a minimum
  - A two dimensional grid on which players can move units
  - Grid size should be customisable to give gameplay variety
  - The grid should have several different types of tile

- **AI Functionality**
  - The AI should have several difficulty settings so that human players can find an AI opponent that is satisfying to play against
  - The AI should show considerable depth in its thinking, and be able to consider several moves ahead of where it is currently
  - The AI should not be perfect, instead it should approximate human behaviour by making mistakes
  - The AI should react to the game and change its tactics based on certain situations

B.0.2 Non Functional Requirements

- Different tiles should be visually distinctive so it is clear to the user exactly what they are

- Units should be visually distinctive and match their abilities (sniper units should carry weapons with a visible scope, whilst a commando should appear to carry a weapon that looks like a shotgun to reflect his short range)
- File size should be kept to a minimum to ensure portability
- The game should run on any system that runs the correct version of Java and not just my own
- Code should be of a high standard: readable and logical in design
- There should be consistent user interface that is easily followed and does not overload the user with information

B.0.3 Performance Requirements

- The AI should return a move within 5-10 seconds on average. Any longer could cause boredom for a human player
- The pathfinding should not cause a noticeable slowdown in gameplay
- Response times when selecting and moving units should be instant
- The collision detection should respond without a noticeable delay: users should not have to wait after selecting a shot to see whether the shot was allowed, instead there should be instant feedback

C Additional Design: Expected Classes

Prior to implementation I made listings of the classes I would need in implementation. As such this is not a complete class list, and was subject to change during implementation. The variations are explained in the implementation section.

C.1 Small Custom Data Structures

In order for the game to operate I will need a number of data structures to represent objects. These objects are listed below.

- **Turn**
  - This structure holds a turn object. These objects are used to compile lists of potential moves for the AI, and new gamestates are also generated from these objects
  - Contains 4 pairs of integers: two pairs are related to an attack (the unit attacking's co-ordinates, and the unit being attacked), and two to a move (the moving unit's co-ordinates, and the location it is being moved to)

  - Contains 3 booleans: one to see if the player moved first or fired first, and two to tell the game if the object contains a move or an attack seperately
  - The object has a single constructor, which simply takes 8 integers and 3 booleans to build a turn

- **UnitNode**
  - This is a simple object that represents a potential unit location
  - Each object has two integers which represent an x and y co-ordinate respectively, and a reference to a unit.
  - There is a single constructor, which takes an x and y co-ordinate as an argument
  - The unit in cell reference is initially null, and if a unit moves into the cell, it is updated with a reference to that unit
  - Implements serializable

- **UnitInitialiser**
  - This simple object holds an x co-ordinate, a y co-ordinate and an integer
  - The integer refers to the team and type of the unit to initialise
  - The x and y co-ordinates are used so that the unit the object is used to initialise is placed at the correct location

C.2 Game Classes

These are the classes I expected to need in the Game package.

- **AttemptShot**
  - This class handles the calculations for checking if a wall or tree blocks a shot.
  - Returns true if a shot is blocked, and false if it is not

- **Game**
  - This class 'sets up' the game: it creates player instances, handles graphics, sets up the game window and tracks game statistics
  - Also contains methods for generating potential AI moves, generating new states from turn objects, performing moves returned from the AI module, and checking if game states are final

- **Game Interface**

  - This class contains the main clause
  - Creates a window containing the main menu from which all the system's functionality can be reached
  - Based on drawings in design section
  - Creates a GameMap object to use in the game

- **GameMap**
  - Represents a game state
  - Has variables representing map width and height, unit counts, as well as arrays containing the grid of tiles, and a grid of UnitNodes
  - Can have a reference to a turn object. This turn would be the turn used to generate this state so that the evaluation heuristic can evaluate the state and the turn used to create it
  - The constructor takes no arguments
  - Uses MapLayouts class to fill the unit grid with Unit Instances and tile grid once they have been created
  - Contains the UnitNode definition
  - Implements serializable

- **Map Editor**
  - Similar to Game class, however it uses a different mouse listener
  - Has a second GameMap object which is static, which holds the map once the user has clicked the save button
  - Has a button to show a window, from which units can be added or removed from the map
  - Has a dropdown menu to choose a save slot to save a map to. Clicking save to file serializes the static gamemap

- **Map Layouts**
  - Has a function called createMap which takes a GameMap and an integer. The GameMap's tile map is filled based on the integer fed to the function
  - A second function createUnits returns a list of UnitInitialiser objects
  - Defines the UnitInitialiser object

- **Map Serialiser**
  - Handles serialising gameMap objects

  - One function serialises a GameMap object taken as an argument. The function also takes an integer as an argument: this is the save slot to save the map to
  - Another function de serialises a GameMap object, and takes an integer as an argument so it knows which file to grab

- **Turn**
  - Implements the turn object described above

- **UnitMover**
  - Implements the Mover interface from the pathfinder, so that UnitInterfaces can be tagged as the object to be passed around in the pathfinder

C.3 Pathfinder Classes

- **AStar**
  - Used to perform an A* pathfinder search
  - Constructor builds an AStar object based on an input map, a maximum search distance, a heuristic, and a toggle for whether or not to allow diagonal movement
  - If no heuristic is chosen, a standard closest first heuristic is used instead
  - The findPath method performs the search, and is given a unit, a pair of input co-ordinates and a pair of output co-ordinates as arguments

- **Closest Heuristic**
  - Implements heuristicInterface and assigns a node a cost such that the tile closest to the target is the best

- **Heuristic Interface**
  - Has a single method, getCost that any implementation of the interface must use
  - Is used to implement new heuristics for the search

- **Mover**
  - Blank interface used to tag objects that are passed around in the pathfinder. Implemented in UnitMover so that it can be used to tag units

- **Node**

  - Data structure used to describe a single node in a path. Has a pair of co-ordinates that describe the node's co ordinates in the grid, a cost, a pointer to a parent node (the node from which we reached this node), a heuristic cost, and the node's search depth
  - Constructor uses a pair of integers to set the node's co ordinates
  - setParent method takes a Node as an argument to allow us to assign the node's parent
  - Implements comparable and has a compareTo method so that nodes can be compared in the pathfinder

- **Path**
  - Data structure to hold a path built in the pathfinder
  - Constructor simply initialises an empty path
  - Contains an array list called steps to which steps are added using appendStep and prependStep
  - A step is an inner class which has a pair of integers x and y to store the co-ordinate's steps
  - The equals method is used by the pathfinder to check if a step already exists in a path

- **Pathfinder Interface**
  - Used when creating new types of pathfinders. AStar implements this interface

C.4 Player Classes

- **AI Player**
  - Implements **Algorithm 1**
  - Evauate State implements the scoring function described in **2.5.1.**
  - swapPlayer is implemented from PlayerInterface. Each game will have a pair of PlayerInterface objects to represent the players. Whether human or AI, both will have a chooseMove method to allow that player to pick their move. swapPlayer calls the chooseMove method in the other Player object.
  - The AIPlayer implementation of swapPlayer also executes the move returned from chooseMove

- **Human Player**
  - Rather than chooseMove allowing an AI to compute a move like in AI Player, this version simply enables the mouse listener so that the player can move and attack at will

  - swapPlayer acts similarly to AI Player, but does not execute the move, as this is handled in the mouse listeners. It does disable the mouse listener whilst the AI takes its turn (if two human players are playing, the other player's object will re-enable the mouse listener)

- **Player Interface**
  - Interface used to create player objects. The implementation of this object and its methods can be used to create different kinds of AI players.

C.5 Unit Classes

- **UnitInterface**
  - All units must implement this interface, which defines a set of methods that can be carried out on unit instances. These methods check whether units are alive or dead, their properties such as maximum move distance and attack distance, their team, and so on

- **Blue/Red Commando/Sniper/Soldier**
  - These are the blue team implementations of the Unit Interface.
  - Red and Blue variants are exactly the same, save for their TeamID
  - Methods are the same across all implementations, the only changes are the initialisations of shoot/move distance
  - All units also implement serializable, since they are initialised by the GameMap class, which itself will be serialized