AR4

Turn-Based Strategy Final Report

by Pedram Amirkhalili
AR4

Advised by
Prof. Sunil ARYA

Submitted in partial fullfillment
of the requirements for COMP 4982
in the
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
On exchange from
Department of Computer Science
University of Warwick
2014 - 2015

April 21, 2015

Contents

1 Abstract 3

2 Introduction 4
- 2.1 Overview 4
- 2.2 Objectives 5
- 2.3 Literature Survery 7

3 Design 10
- 3.1 Level Generation 11
  - 3.1.1 Map 11
  - 3.1.2 Units 12
- 3.2 Weapons 14
- 3.3 Gameplay 15
  - 3.3.1 AI 15
  - 3.3.2 Win Conditions 16
  - 3.3.3 Combat 17
- 3.4 Graphical User Interface 18
  - 3.4.1 General 18
  - 3.4.2 Player Interactions/Clarity of Information 20
- 3.5 Controls 26

4 Implementation 27
- 4.1 Graphics 27
- 4.2 Maps 27
  - 4.2.1 Terrain 27
  - 4.2.2 Generator 28
  - 4.2.3 Clean up 36
  - 4.2.4 Validity 36
- 4.3 Units 37
  - 4.3.1 Generator 37
- 4.4 Weapons 37
- 4.5 Gameplay 38
  - 4.5.1 Combat 39
  - 4.5.2 AI 40
- 4.6 Genetic Algorithm 41
  - 4.6.1 Fitness Function 41
  - 4.6.2 Selection 42
  - 4.6.3 Crossover 42
  - 4.6.4 Mutation 42
  - 4.6.5 Additional Information 43

5 Testing **44**
5.1 Unit Testing . . . 44
5.1.1 Map Generator . . . 44
5.1.2 Unit Generator . . . 49
5.1.3 Game Mechanics . . . 50
5.1.4 AI . . . 50
5.1.5 Genetic Algorithm . . . 51
5.2 System and Acceptance Testing . . . 52

6 Evaluation **53**

7 Discussion **54**

8 Conclusion **54**

9 Project Planning **55**

10 Required Hardware and Software **55**
10.1 Hardware . . . 55
10.2 Software . . . 55

11 References **56**

12 Appendix A **56**
12.1 Minutes from First Meeting . . . 56
12.2 Minutes From Second Meeting . . . 57
12.3 Minutes From Third Meeting . . . 57
12.4 Minutes From Communication Tutor Meeting 1 . . . 58
12.5 Minutes From Communication Tutor Meeting 2 . . . 59

13 Appendix B **60**
13.1 Questionnaire Response 1 . . . 60
13.2 Questionnaire Response 2 . . . 61
13.3 Questionnaire Response 3 . . . 62

1 Abstract

Turn-Base Tactics (TBT) games are often plagued with long level design times. This paper describes a system that randomly generates levels with the help of a genetic algorithm.

"Endless Tower" throws the player into an endless fight against waves of enemies across randomly generated maps. The player takes control via the keyboard as they try to outwit the AI by using their units and the structure of a map forged by a genetic algorithm. With stylised 2D graphics Endless Tower is designed to appeal to retro and modern gamers alike.

2 Introduction

2.1 Overview

Turn-Based Tactics (TBT) games take the format of two players, each taking a turn to perform actions on their forces. These actions are things such as moving units, attacking or healing, as well as a variety of other minor actions. A player's turn ends either when they decide that performing actions is no longer beneficial, or when all of the units within their force have performed an action.

The scope of the project is to address two issues in TBT games, the process of creating an individual level is too long and without the right testing created levels can be too difficult at certain points in the game.

The first issue, the length of time that level creation takes, can be dealt with by randomly generating levels that fit a set of criteria. Now instead of level creation taking upwards of a week, it can be completed in a fraction of the original time.

However by doing this second issue becomes worse. The difficulty of the levels will now vary greatly and won't follow a specific pattern causing frustration for the player.

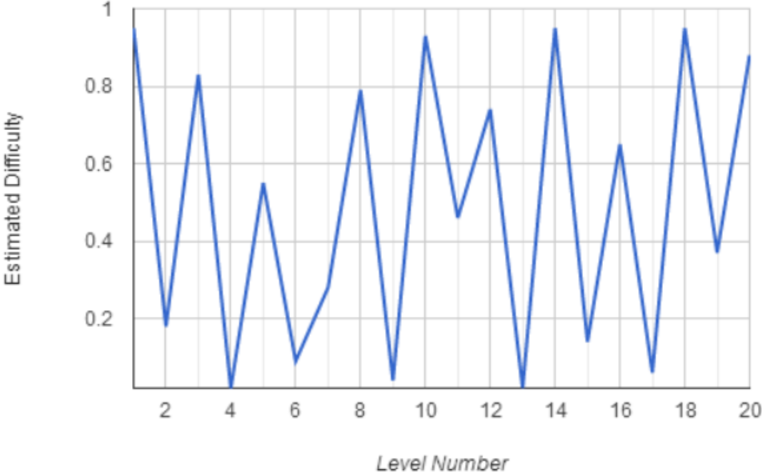

Figure 1: A difficulty curve that may occur from random generation, due to the fact that all control over the level is deferred to a random chance. 0 estimated difficulty is the easiest and 1 is the hardest.

This issue can be resolved through the introduction of a Genetic Algorithm, to evaluate and effectively control the process of randomly generated levels. It

does this by trying to generate a level at around a specified difficulty.

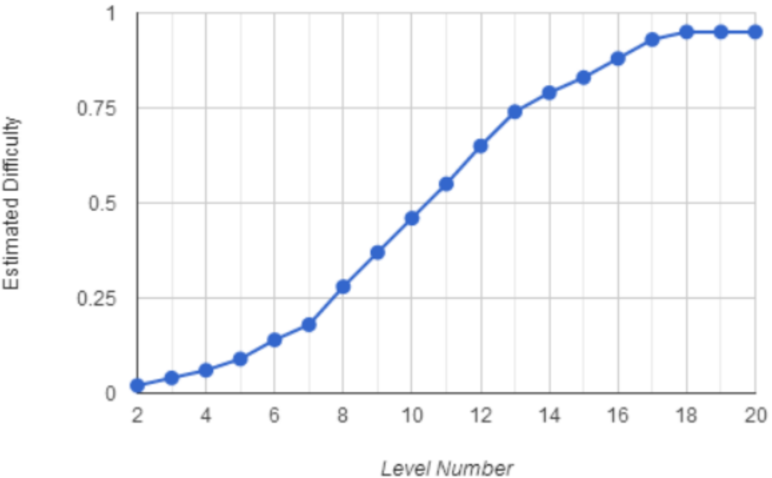

Figure 2: Another difficulty curve plot, however this is a pattern that is often aimed to be replicated in games

In the case of Fig.2, if the 8th level is wanted, then the genetic algorithm would aim to try to breed a level around the difficulty of 0.26.

Each level consists of a map, an AI, a set of units the player controls and a set of units the AI controls. All of these elements are randomly generated, or in the case of AI randomly selected from either a passive or aggressive AI.

After a set of levels has been generated, they are evalauted and bred by the genetic algorithm until a level is deemed to be acceptable, i.e. it fits roughly in place on the difficulty curve in Fig.2

2.2 Objectives

This project has a series of main objectives:

1. **Randomly Generated Levels**: All levels are created through random generation
   - (a) Map - This is made up of a series of tiles, which have been randomly selected from a set of specific types, should appear slightly realistic
   - (b) Units - These make up the player and AI forces, randomly generated by selecting a class from a pool and then adjusting the base stats, before levelling up if required

   (c) AI - Selected from two different AIs, passive and aggresive. As well as the order in which targets are selected

2. **Core Mechanics**: A set of actions that are essential to gameplay

   (a) Movement - Moving a unit from one location to another

   (b) Attacking - Attacking an enemy unit

   (c) Healing - Refilling an allied unit's health

   (d) Inventory Management - Selecting which weapon to have equipped or to use in combat

   Core Mechanics are gameplay aspects that are required for the game to function. In the case of a TBT game it is the ones stated above. These make up the bulk of the basic gameplay of the game and are enough to allow for progression into creating the genetic algorithm.

3. **Genetic Algorithm**: The algorithm that controls the creation of randomly generated levels

   (a) Difficulty Curve - The difficulty curve shown in Fig.2 is what the algorithm aims to replicate, x level should have roughly y difficulty as described.

   (b) Maps - From the randomly generated maps, the genetic algorithm takes into account choke points to help decide on difficulty.

   (c) Units - Each class has a value, from this and their level a score is calculated.

   Through the usage of a Fitness Function and then Selection, Crossover and Mutation[4] the genetic algorithm attempts to find a level that is of the correct difficulty for the specified point. This addresses any issues with difficulty not following a pattern

4. **Gameplay Features**: Non-essential aspects that improve the players experience

   (a) Graphics - The aesthetics of a game don't play a key role in whether people enjoy the game or not, due to this a more simplistic 2D, pixelated "sprite" style is used.

   (b) Clarity of Information - Insuring that information is relayed to the player in a clear and understandable manner through the Graphical User Interface, such as the use of a History Panel to show actions that have been taken.

   These features simply improve the user experience as oppose to helping towards the genetic algorithm. They do this by providing the user with feedback as to what they have done. They have be designated as goals as to help the game be as fun as possible.

2.3 Literature Survery

Genetic algorithms have distinct stages that work on solutions, or what is referred to as Chromosomes - "a candidate solution to a problem, often encoded as a bit string"[3]. A Chromosome is the entire level, it isn't represented through a bit string but by the components that make up a level.

The first stage is to randomly generate **n** chromosomes, this set is called a Generation, and then through a Fitness Function evaluate each one. A fitness function "assigns a score (fitness) to each chromosome in the current population"[3]. In the case of this project, it is the estimated difficulty of each level. The function can be represented like so, for a level x:

$$f(x) = MapDifficulty + \\ ((PlayerUnitsScore * PlayerSkillScore) - (AIUnitsScore * AISkillScore))$$

Where Map Difficulty, m(x), is defined as such:

$$m(x) = \sum_{cP=0}^{n} ChokePointScore \text{ (cP)}$$

$$ChokePointScore = \text{closer to player}\{^{\text{true = positive}}_{\text{false = negative}}$$

$$cP \text{ is a individual chokePoint}$$

NB: That for the m(x) the assumption is made that simply by being closer to an identified choke point the player or AI has an advantage. In reality this may not be the case since certain units are often required to take full advantage of them, as well a scenario arising in which allows for the choke to be used.

Where Player Units Score, pU(x), is defined as such:

$$pU(x) = \sum_{u=0}^{n} UnitScore \text{ (u)}$$

$$UnitScore = \text{rating based on the level and value of the unit (level * value)}$$

$$u \text{ is a individual unit}$$

AI Units use the same calculation but for the sake clarity it is named aU(x). Player skill is assumed to be following the curve that the algorithm is trying to create and the AI skill is a constant value dependent on the AI.

The results are then normalised to be between 0 and 1, where 0 is the easiest difficulty and 1 is the hardest.

The second stage is the beginning of what the bulk of a genetic algorithm is, this part is referred to as Selection. Where an "operator selects chromosomes in the population for reproduction"[3], for Selection there are two ways this can be done:

1. Cull any chromosomes with a fitness outside of a range around the value that you are looking for. Then select pairs of chromosomes with equal chance.

2. The alternative is that the "fitter the chromosome, the more times it is likely to be selected" [3], so what is described in nature as Survival of the Fittest.

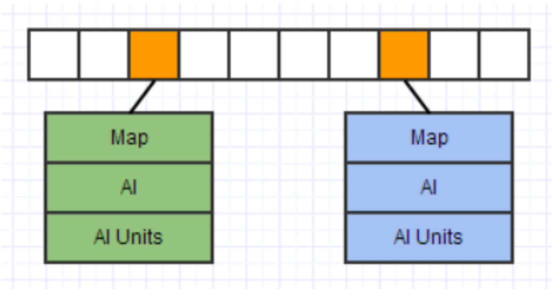

Figure 3: Two levels being selected for breeding

Then having then selected a pair of chromosomes using one of the above methods, Crossover is performed, where a locus is chosen and "subsequences before and after that locus between two chromosome" [3] are used to create two offspring. This is normally applied to the bit string, for this project the components of the evaluation equation are be used. For example the below AI Units and AI are taken from one chromosome and then the map from the other to create the first child, the second is the inverse. Note that the player units remain constant so they are the same across all chromosomes and generations.

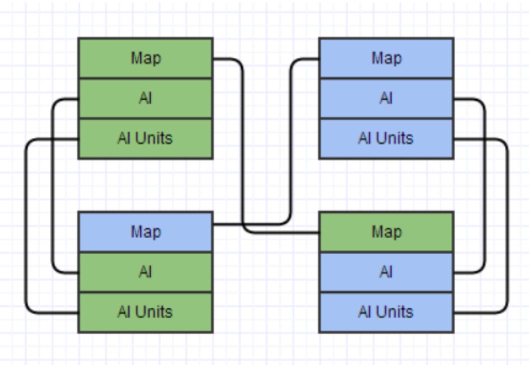

Figure 4: An example of crossover, where the locus is selected to be the Map

The final stage is Mutation where an operator "randomly flips some of the bits in a chromosome"[3], and this is applied to the newly generated children. Instead of just modifying the components of the level when mutation is chosen to be applied, the component is completely regenerated from scratch and any adjustements required are made.

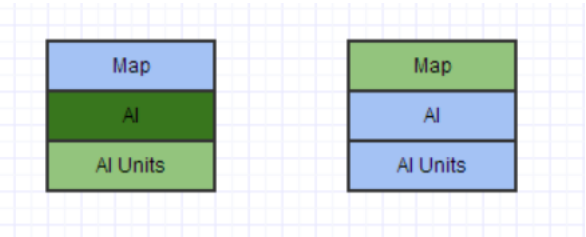

Figure 5: An example where AI from the first child has been completed regenerated

This new generation of chromosomes replaces the old one and can then be used for repetition from the second stage onwards, if required. Before the second stage there is a check to see if either from the initial random generation or the newly bred levels meet the current requirements. i.e. it is in the correct place on the difficulty curve and therefore can be used as the next level the game uses.

To avoid a scenario where the a appropriate level is never generated every n number of overall passes (Selection, Crossover and Mutation), the check is made more lax. What is meant by this is from searching for a single value, it becomes a range that progressively gets larger and larger.

3 Design

The project is split into two major stages, the first is the completion of the random level generation and core mechanics. The second is creating a genetic algorithm to control the random generation.

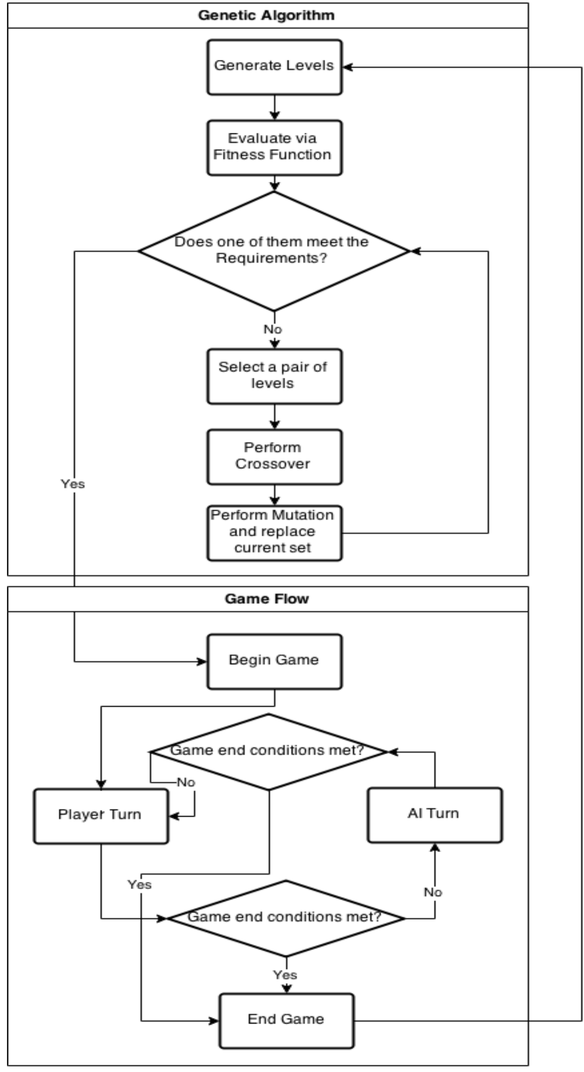

Figure 6: A diagram showing the overall flow of the program

3.1 Level Generation

3.1.1 Map

A major part of each level is the map that contains a series of tiles, of which themselves are of a terrain type from a pool of types. The map is the location where the battle takes place and should provide opportunities for an advantage to both sides if possible. Here is the set of tiles that are used in the game:

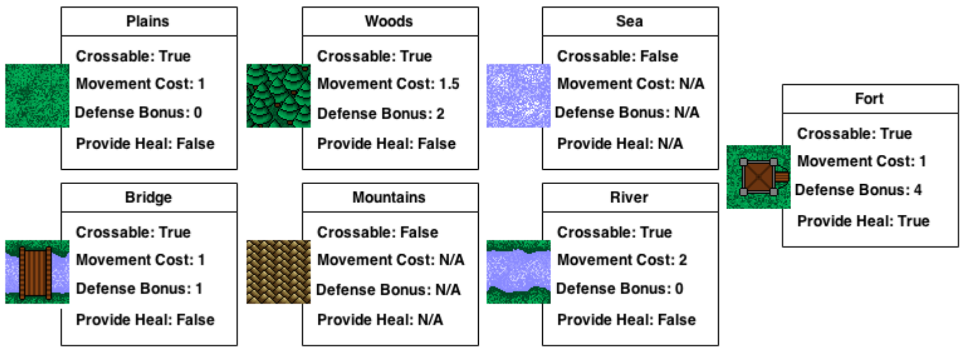

Figure 7: A set of definitions for the different terrain types with corresponding images

Fig.7 shows the 7 different Terrain types that are included in the game. The purpose of having so many is to provide a less bland visual design overall, as well as giving the game more depth as each Terrain is designed to have advantages and disadvantages:

Table 1

Terrain	**Advantage**	**Disadvantage**
Plains	Easy to move across	No defense bonus
Woods	Provides Defense Bonus	Harder to cross than Plains
River	Tactical Advantage*	Hardest to cross, no defense bonus
Bridge	Provides crossing on rivers	Rare, location may be inconvenient
Fort	High Defense Bonus, Healing	Rare, location may be inconvenient
Sea	N/A	N/A
Mountain	N/A	N/A

NB Tactical Advantage in Table 1 means that a player can use it to seperate their forces from the oppostion to create an advantageous situation, and due to Sea and Mountain both being uncrossable they provide no real advantages or disadvantages. The reason for having two different uncrossable Terrain types is

for variety.

When a map has been generated it needs to be checked to ensure it is valid, this entails the following conditions:

1. The map has no unaccessible crossable tiles (crossable terrain surrounded by uncrossable terrain)
2. Rivers “flow” in a correct manner

By ensuring the map is valid, random generation becomes a viable way to create maps as oppose to design them by hand.

3.1.2 Units

In TBT games players take turns to take actions upon their forces/armies. These are composed of several different units. These units have a class, and these classes change the statistics of that unit and therefore the role they play. As well as different base statistics a units class effects the weapons that they can make use of, as well as their Growth rates. A Growth rate is a percentage that defines that chance of a statistic of a unit increasing upon levelling up. Before going over the different classes, the different statistics that a unit has are explained below:

1. **Attack** - This affects how much damage a unit does when using a physical weapon (swords, lances, etc...), the higher the more damage can be dealt
2. **Magic** - This affects how much damage a unit does when using a magic weapon (tomes), the higher the more damage can be dealt. It is also used for healing (staves), the higher the better.
3. **Defense** - This reduces the amount of damage a unit takes when being attacked by a physical weapon, the higher the more the incoming physical damage is reduced
4. **Resistance** - This reduces the amount of damage a unit takes when being attacked by a magic weapon, the higher the more incoming magic damage is reduced
5. **Health** - This is the only “resource” of such that a unit has, upon reaching 0 the unit dies and is permanently removed from the game
6. **Skill** - This affects how likely an attack is to hit the defending unit, as well as how likely it is to Crit. A Crit deals 3x the normal amount of damage. The higher the skill the more likely to an attack is to hit and crit.
7. **Speed** - This affects the number of times units can strike each other in combat. If the one unit’s speed is significantly greater than the others, they can perform two attacks. It is also used to calculate the dodge/evasion

chance of the defending unit, the higher the speed, the better the evasion and likelyhood of performing two attacks.

8. **Movement** - This is how far the unit can move in a single turn, this value is set and doesn't change even upon levelling up
9. **Weapon list** - This is the list of weapons that a unit can equip and use. Each weapon also has a set of statistics.
10. **Exp** - Whenever a unit has participated in combat they receive exp, upon reaching 100 exp they level up. Exp is then reset to 0 and all other statistics, excluding movement, have a chance to improve.

The specific equations for damage, damage reduction, etc. are discussed later on.

Figure 8: A set of definitions for the different units with corresponding images

Fig.8 shows the base statistics of each class, these are the numbers assigned to the unit when it is first created. To add greater variation at lower levels, these statistics are randomly modified by -2 to +2 after being created. If a unit is required to made a higher level that 1, it simple goes through the level up the required number of times to achieve the required level.

Each unit is designed with a specfic role or purpose in mind and is listed below in Table 2:

Table 2

Unit	Role/Purpose	Notes
Acolyte	Provides magic damage	1 - 2 range
Archer	Ranged physical damage	1 - 2 range
Axeman	High damage, low defense	High risk, high reward
Cavalier	Mobile and Fast	Otherwise average statistics
Defender	Tank	Supposed to be able recieve lots of damage
Lord	"King"	Player only, on death game ends
Shaman	Healer	Can't deal any damage
Swordsman	Balanced physical damage	Less risky Axeman

3.2 Weapons

As well as statistics from units and terrain affecting gameplay, so do the statistics that weapons have. Weapons are what the units use to perform actions such as attacking or healing. There are 6 different weapon classes: Bow, Tome, Sword, Axe, Lance, Stave. NB: Staves have no offensive power and are simply used for healing. Here are the statistics that each weapon has:

1. **Damage** - This helps to determine how much damage the weapon can do, the higher the more damage is be dealt.
2. **Uses** - During combat after every strike a weapon's usage is reduced by one, upon hitting 0 the weapon breaks, better weapons often have less Uses than their weaker counterparts
3. **Hit** - Part of the calculation as to whether an attack against a targeted unit lands, the higher the more likely an attack is to hit
4. **Crit** - Increases the likelihood of an attack being a crit, the higher the more likely a crit is to occur.
5. **Range** - The distance the weapon can attack from, either 1 or 2
6. **Cost** - The price of the weapon in in-game currency

NB: The usage of the damage statistic in calculations are explained in the next segment.

Weapons have a tier list, a staple of TBTs, all classes except Tomes and Staves are to use the following structure: bronze, iron, steel, mithril. Bronze being the

worst in terms of damage and mithril being the best.

Tomes are not going to have a tier, simply 2/3 different types, each of which have advantages and disadvantages when compared to the others. To begin with there is only a single stave.

3.3 Gameplay

3.3.1 AI

As well as maps and units an AI is needed to control the opposition to the player. For this there are two simplistic AIs that have different natures:

1. **Passive** - This AI simply waits for the player's units to come into its own range before moving and attacking themm if there is no unit in range it does nothing.
2. **Aggressive** - This AI attempts to move towards the player if there are no attackable units in range. If there is a player unit in range, it performs an attack on said unit

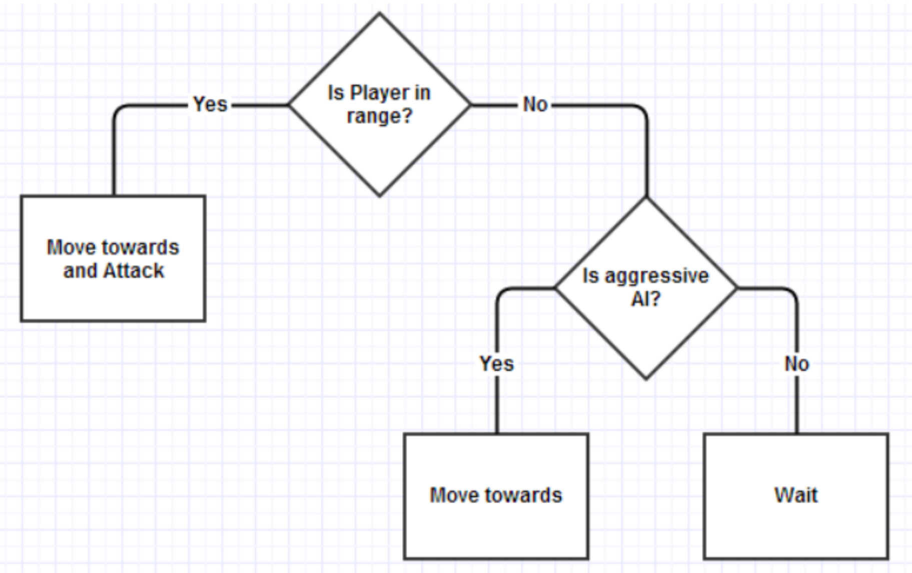

Figure 9: A flow chart showing the overall process the AI controller performs

Fig.9 shows the process that is performed on all of its units, except Shamans, by the AI controller. Shamans are the exception to this as they cannot perform attacks, and their primary function is to heal allied units. This also means that the chosen AI doesn't effect how they act.

If the unit is a Shaman it follows a seperate flow chart shown in the figure below:

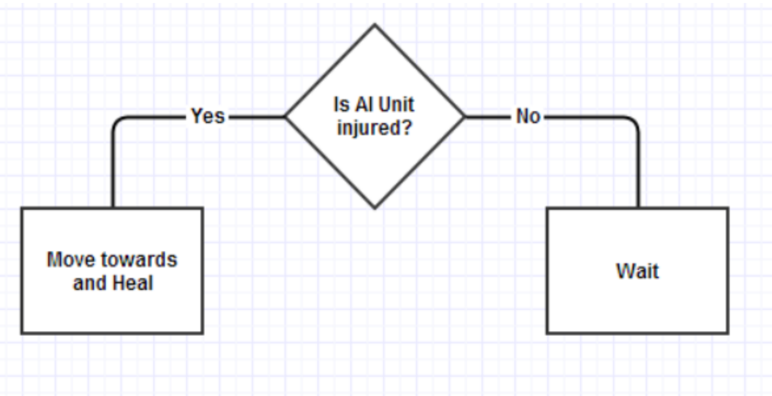

Figure 10: A flow chart specifically for the unique case of AI Shamans

For all units they may “Move towards” a target and perform their desired action if possible, to ensure some effeciency a shortest path algorithm is used to locate the fastest route between the unit and their target. So for example in Fig.9 the aggressive AI is stated to “Move towards” the player if nothing is in range, it does this by selecting the player’s Lord as each units target. Then the shortest path between the unit and Lord is calculated, the unit then moves as far along this path as it can.

Both AIs simply attack the first unit detected within range with no prior thinking. To counteract this, there is a chance that either of these AIs has the Ruthless trait. This trait adds a priority ordering as to which unit to attack that is within range, the order is as follows:

1. Lord
2. Shaman
3. Archer, Acolyte
4. Swordsman, Axeman, Cavalier
5. Defender

For each level one of these AI’s is selected to play against the player, along with whether it is Ruthless or not.

3.3.2 Win Conditions

In the flowchart at the beginning of this section, a check is made after either the player or AI’s turn to see if the game win conditions have been met. These

can be of a variety of different simple goals dependent on the game mode, in this project the objective of the player and AI is twofold:

1. Kill the enemy leader
2. Wipe out the enemy forces

If either of these conditions is met by the player or AI they will win the game.

3.3.3 Combat

All of the other core mechanics, mechanics that are required as a bare minimum for the game to function, have no real calculation. So the main focus will be looking at the equations that are used for combat. The equations can be split into two different categories:

1. **Defensive**
   (a) Evade Chance = Speed (Unit) + Defensive Bonus (Terrain) : The defense the terrain provides, can help increase the chance of dodging an attack.
   (b) Physical Defense = Defense (Unit) + Defensive Bonus (Terrain) : The defense of the unit and the bonus from the terrain they are on helps to negate incoming physical damage
   (c) Magic Defense = Resistance (Unit) + Defensive Bonus (Terrain) : The resistance of the unit and the bonus from the terrain they are on helps to negate incoming magic damage
2. **Offensive**
   (a) Attack Speed = Speed (Unit) : If the attacking unit's speed is 4 greater than the opposition's, then they are able to attack twice.
   (b) Hit Rate = Hit (Weapon) + Skill (Unit) : The hit chance of the weapon and the skill of the unit determine their hit rate, but not the overall accuracy.
   (c) Accuracy = Hit Rate (Offensive Unit) - Evade (Defensive Unit) : Takes into account the evade ability and hit chance of both units, values returned are forced to be between 0 - 100
   (d) Physical Attack = Attack (Unit) + Damage (Weapon) : Uses both the weapons damage and the units attack to get the amount of physical damage they can do
   (e) Magic Attack = Magic (Unit) + Damage (Weapon) : Uses both the weapons damage and the units magic to get the amount of magic damage they can do
   (f) Overall Damage = Physical/Magic Attack (Offensive Unit) - Physical/Magic Defense (Defensive Unit) : Takes the maximum damage the attacker can do and reduces it by the amount the defending unit can negate it for.

(g) Crit Chance = Crit (Weapon) + Skill / 2 (Unit) : The unit's skill and the weapon's crit chance effect the amount they can deal additional damage

(h) Crit Damage = Overall Damage * 3 : Critical hits do 3x the base damage that would have been dealt

3.4 Graphical User Interface

3.4.1 General

One of the objectives of this project is to have stylised graphics. This is achieved by using a pixelated tileset that is used to portray not only the levels but also seen the title screen below:

Figure 11: The title screen

As well as this static title screen there is one more static screen that was made, the loading screen. The purpose of this is so that while levels are generated the player knows that the game simply hasn't crashed.

Figure 12: This loading icon appears on a black screen

The game itself has a simple interface that consists of 2 segments that display all the required information:

Figure 13: A screenshot of the UI in game

1. **Map Area -** This is the large square segment that takes up the bulk of the screen from the lefthand side. It shows a portion of that battlefield and any units that reside there to the player. This is where the player performs all of their commands, by using a cursor to control to select units. The cursor is the small silver edges around a central tile. The player's units are the green shields with unit icons, and the AI are red shields with unit icons.

2. **History/Info Panel** - This is the rectangular panel on the righthand side, it contains all the information about actions the player and the AI have performed over the course of the battle. This was added to help achieve the "Clarity of Information" objective.

3.4.2 Player Interactions/Clarity of Information

On top of creating a UI additional graphics have been added for the sake of clarity of information, ensuring the user is given all the relevant information based on the actions they perform. This should be displayed in a meaningful and understandable manner. Below is a series of screenshots showcasing these additional graphics and a small paragragh explaining the purpose.

Figure 14: A screenshot of the menu that appears upon "examining" a unit

Fig.14 shows the screen when the player examines a unit, note this can be done to both player and AI units with the same result. What this does is it brings up a small menu box that displays the statistics of the unit, including weapon. The purpose of this is to allow the player in game to be able to analyse if the unit has the right statistics for the role they would like it to play.

Figure 15: A screenshot of what is displayed when the player selects an AI unit

Fig.15 shows the result of selecting an AI unit, the red overlay that appears is to show the attack range of the AI unit. This is the range in which the selected unit will be able to attack any player units. In the example above the player's Shaman is a potential target for the selected Cavalier. This allows the player to be able to see what areas are safe for them to move their units to.

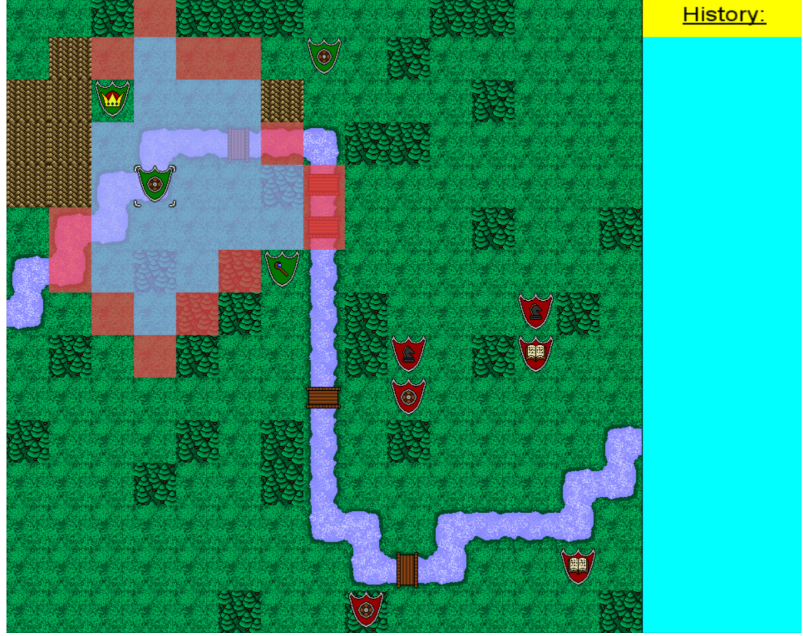

Figure 16: A screenshot of what is displayed when the player selects one of their units

Fig.16 shows what happens when the player selects one of their own units. This causes two different coloured squares to be overlayed, blue represents the movement range of the seleted unit across the map and the red represents the furthest away that unit can perform an attack at. The purpose of this is so that the player can identify which AI units they can attack and how far they can move each unit from that position.

Figure 17: A screenshot of what happens the player selects a blue square after selecting one of their own units

Fig.17 shows the menu that appears when the player wants to move their units within the range they can move. The blue box indicates which option on the menu the player is hovering over.

Figure 18: A screenshot of the menu that occurs when a player selects any AI unit in that is sitting in the red tiles

Fig.18 shows the menus and information that appears when the player selects an AI unit for a potential attack, along with the options for attacking or waiting there is 2 boxes that appear to show the information about the attack that is going to be performed. They contain infomation about the fight between the two units so the player can decide if it is worth it for them.

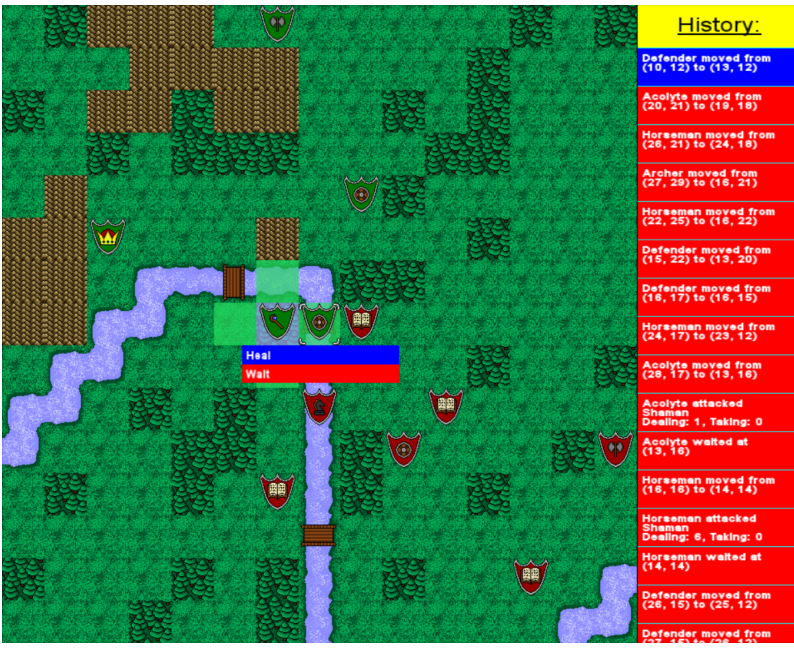

Figure 19: A screenshot of the menu that occurs when a player selects an injured ally unit to potentially heal

Fig.19 shows what happens when the player chooses to heal an injured ally, the green overlayed tiles represent the heal range of the Shaman. This was set to be a different color from attacking as not to confuse the player in what that unit can do.

Figure 20: An example of the information contained with in the History

Fig.20 shows a more detailed example of what information is displayed in the history panel, it contains information about where units have moved to, the damage they dealt/took in combat etc... The purpose of this is so that after the AI turn the player can see what has been done as well as seeing what they did themselves

3.5 Controls

TBT games do not have intense mechanics that require complex controls, to reflect this the player only needs a few buttons to do everything. Here is the

list of keys needed and the corresponding actions:

- **Arrow keys -** These move the cursor around the Map area of the screen
- **Z key -** This is the selection key, used to choose options from menus as well as selecting units
- **X key -** This is the deselect key, it goes back from menus and selections
- **A Key -** This is used to examine units on the battefield
- **S Key -** This is used to bring up the end turn menu, if the player wants to end the turn while some of thier units haven't performed an action
- **C Key -** This is used to scroll up on the history panel
- **V Key -** This is used to scroll down on the history panel

4 Implementation

The programming for this project has been done in C++, along with a 2D graphics library called sfml, version 2.0 [2]. This draws images to a window and allows for user input to control objects with in this space.

4.1 Graphics

As mentioned above graphics is done through the library sfml, it works by creating a window, and then drawing into it. On every button press that has an action associated with it the contents of the window is redrawn to reflect this.

For the actual textures and images used in the game, they were created using GIMP 2.0 to create tiles within a designated size. All of these images were created by myself.

4.2 Maps

4.2.1 Terrain

The map of a single level is stored in a single dimensional array of Terrain objects. In the design all the information regarded the types of terrain is specified, for the implementation an object-oriented approach was taken. A generic parent class called Terrain was created, each unique type of terrain was given its own child class.

However for rivers this differs, in that a base River class was created as a child of Terrain, and then several children for the different directions the river can flow were created as children of this River Class. This was needed for the way in which the map is drawn using the ID's of each Terrain type to identify its

location in a .png that was used to hold the textures.

To display the map a single dimensional array is to used to hold the information before drawing it through the use of a Vertex Array. This same method has been repurposed for drawing other aspects of the game to the screen, this will be mentioned where appropriate. [1]

4.2.2 Generator

For generating the map, different methods were used for the different types of terrain. To begin with the single dimension array was filled with Plains as this is the base of the level. After this the other types are added one by one in a specific ordering:

1. **River and Bridge** - This is added first as to make sure there are no obstacles to the its "flow"

2. **Sea** - Next is the Sea tiles, as an uncrossable tile it can't be overwritten and needs to hold a specific shape so it is given priority

3. **Mountains** - Again as an uncrossable tile it is given priority since it must take a specific shape

4. **Woods and Forts** - These hold the least importance and have no rules as to how they can be placed so they are placed on the map last

The river algorithm doesn't require any checks to make sure it doesn't overwrite anything on the map currently, this is due to the map currently only containing Plains tiles. This is why the river is placed first as to give the algorithm the freedom to not have to worry about where it chooses to make the river "flow". The river algorithm works in three distinct stages, the first is to pick two tiles on the edge of the map to be the "ends" of the river.

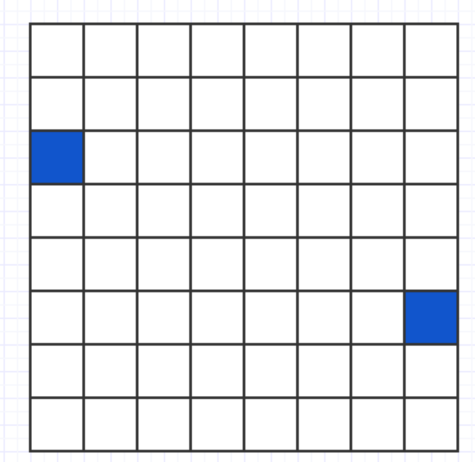

Figure 21: Two edge tiles have been selected

Having selected two edge tiles, like in Fig.21, the two tiles are connected through straight lines.

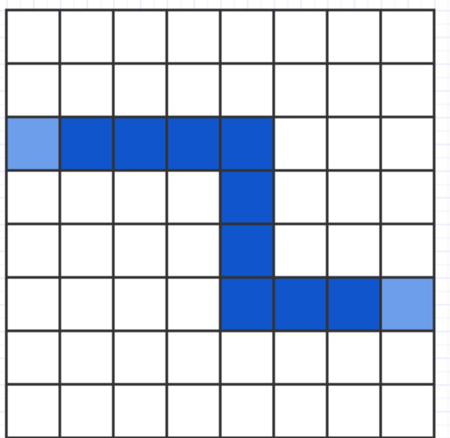

Figure 22: Two Edge river tiles being connected

The lighter blue tiles represent those that have been placed, the darker blue is for tiles that are currently active/being placed.

The final stage is a recursive algorithm that generates “bends” in the river, this is designed to give the rivers a more natural flow and less of a rigid look. It works like so:

1. Pick one of the connecting straight lines

2. Select a random point on that line
3. Apply the largest deviation possible (cap of 4)
4. Remove any unwanted tiles underneath the bend
5. Repeat with remaining segments of the line, till out of space
6. Repeat with other remain straight lines

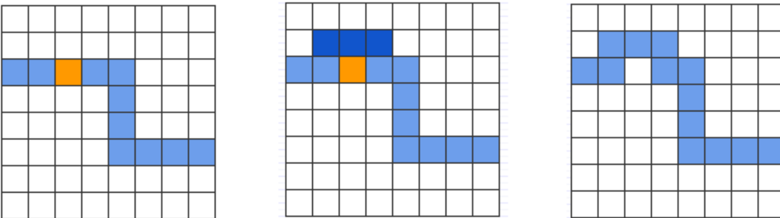

Figure 23: The process of adding a deviation to a straight connecting line

Fig.23 shows stages 2-4 of the recursive algorith, the orange tile represents the selected tile where the deviation should be applied. After completing that deviation the algorithm would see repear the process over the two remain straight connecting lines.

During the process of generating a river, each river tile has the chance to be a Bridge tile instead of a River tile, this is done to create zones of strategic importance as the river can be crossed at half the cost. The chance of a River tile turning into a Bridge tile is set to 15%.

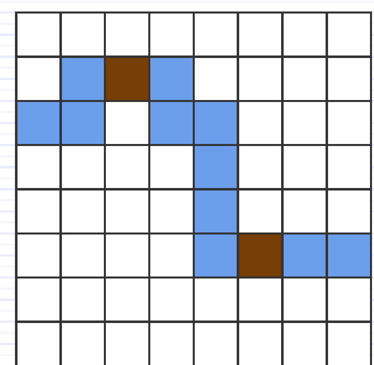

Figure 24: What a finished river may look like with Bridges (Brown tiles)

Next both of the uncrossable terrain types are placed on the map, to decide the amounts of each a random number generator is used. First to decide the total number of tiles on the map that are be uncrossable, this varies between 5 - 20% of the total number of tiles. This number is then split up into two numbers by a percentage between 0-90%, (divided by 100 to get the sea_mountain_ratio) like so:

$$mountain = (total_tiles * percent_uncrossable) * sea_mountain_ratio$$
$$sea = (total_tiles * percent_uncrossable) * (1 - sea_mountain_ratio)$$

Similar to the River tiles the Sea tiles have no restrictions on their placements. This decision was made as it means that Sea tiles can potentially overwrite the edges of the river, this provides a more natural effect of the river flowing in and out of the sea on some maps.

The Sea creation algorithm is split into 3 seperate parts, the first is to select a single edge tile.

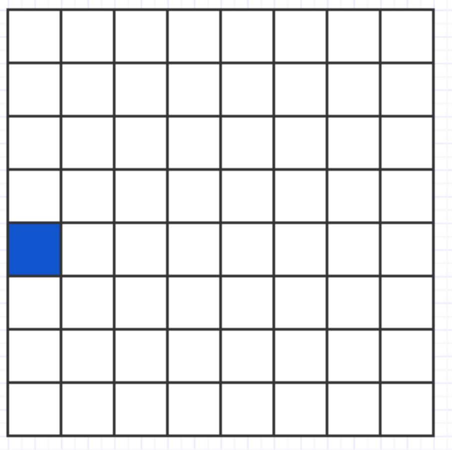

Figure 25: The starting location for the Sea being selected

Having placed this edge tile, a quater of the tiles designated for the sea are used to draw in a direction along the edge the tile resides. If the number of tiles drawn to the edge exceeds the amount of edge left then they are drawn on another edge like below:

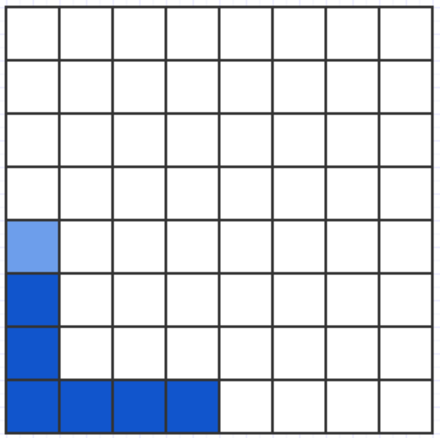

Figure 26: Two Edge river tiles being connected

After this base has been drawn to the map, starting at one end of the line, the number of tiles going inwards towards the center is changed. Either it is increased by one, decreased by one or remains the same. The chance of each happening is affected by how far inwards the previous tile's "row" went inwards, this is shown in the table below:

Table 3

Amount Inwards	**Increase by 1**	**Decrease by 1**	**Remain the same**
1	100%	0%	0%
2	60%	15%	25%
3	50%	20%	30%
4	30%	20%	50%
5	20%	30%	50%
6	20%	50%	30%
7	15%	75%	10%
8	0%	100%	0%

After deciding which of the 3 choices to take a new line is drawn inwards and the algorithm moves on to the next tile. This process is repeated until the end of the line(s) has been reached or when the number of available sea tiles reaches 0.

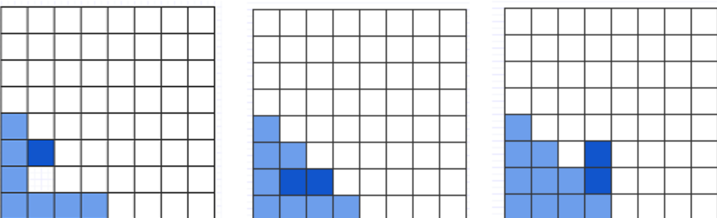

Figure 27: Starting from the left, a set of diagrams showing the 3rd stage of sea generation in action

Having created a river and a sea segment on the map, next is Mountains, unlike the previous two terrain types Mountains has some restrictions applied to its placement. A mountain tile can only be placed on a Plains tile, it doesn't have the ability to overwrite either the River or Sea tiles.

The algorithm to add Mountains try to add them in "clusters" to give a more logical look to their placement on the map. This is done ine 3 stages, the first is to select a valid tile i.e. a Plains tile. The dark brown tile represents the currently selected tile.

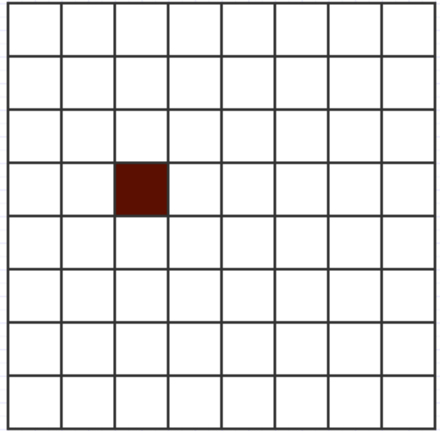

Figure 28: A starting location for a Mountain "cluster" being selected

The surrounding tiles are then added to a vector of tile adjacent to the current cluster of mountains. In the figure below the darker grey is a mountain and the lighter gray are the tiles residing in the vector of adjacent tiles.

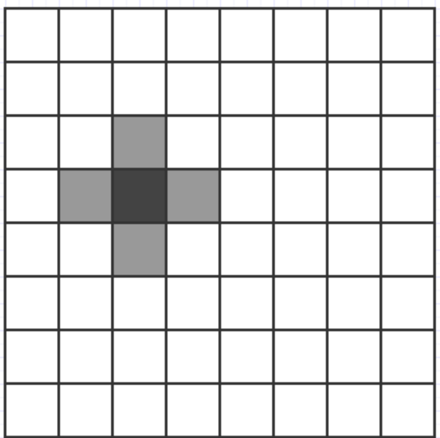

Figure 29: A diagram showing the current cluster and it's adjacent tiles

Before selecting a tile from the vector to be another mountain tile a check is first made to see if a new cluster should be formed. After every tile placed in the current cluster the chance of changing to a new cluster increases by 3%. This is to stop clusters that are too large forming as they are more likely to make areas of the map unaccessible.

If this check is failed, so we can continue on with this cluster, then a tile is selected from the vector of adjacent tiles and tested to see if it is a Plains tile. If so then it is replaced and its adjacent tiles are added to the vector

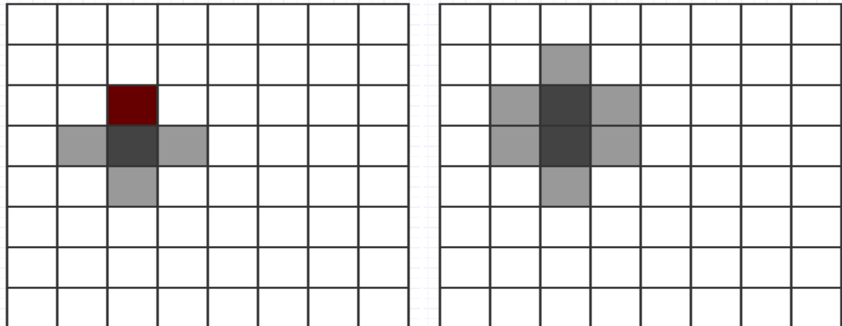

Figure 30: The dark brown tile is selected and added to the current cluster and its adjacent tiles added to the vector

Then if after adding this new tile to the current cluster the check is passed, so we need a new cluster. Then the vector is cleared, the chance of moving to a new cluster is back to it's base and a new start location is selected, like below.

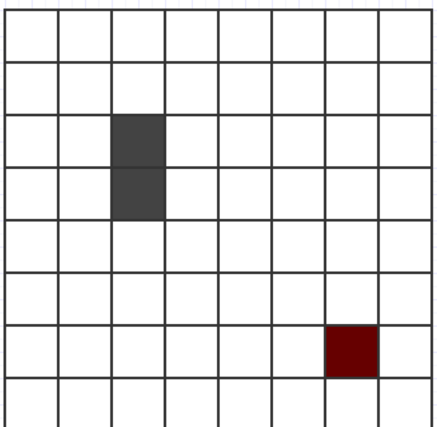

Figure 31: A new cluster being started, all adjacency information is cleared

After having placed all these tiles, Wood and Fort tiles are then placed, these are only allowed to placed on top of Plains tiles. This is done to reduce the chance of the map being invalid. Woods and Forts are placed by randomly selecting a

tile, checking if it is a Plains tile. Then if it is replacing it with whichever is wanted, if not another tile is selected. This process is repeated until the number of Forts and Woods has been met.

4.2.3 Clean up

On top of generating all of these different types of Terrain types there is also a “cleanup” method. It simply iterates through the whole map and checks some criteria for each terrain type. The purpose of doing so is to reduce the number of invalid maps by changing small things. These are:

- **Plains** - If a Plains tile is surrounded by uncrossable terrain then it is simply changes out for one of the uncrossable terrains around it
- **Sea** - Nothing needs to be adjusted for tile
- **Mountains** - Nothing needs to be adjusted for this tile
- **Forts** - If a Fort tile is surrounded by uncrossable terrain then it is simply changed out for one of the uncrossable terrains around it
- **Bridges** - Both sides that aren’t river of the bridge are checked to be accessible tiles (crossable and not out of bounds) if this is the case then the bridge is left alone. If the sides of the bridge that aren’t river don’t fulfill these conditions then the bridge is replaced with a River tile.
- **River** - Nothing needs to be adjusted for this tile

4.2.4 Validity

For a map to be considered valid it needs to fulfill two conditions, the river flows correctly and is only interrupted by Sea tiles and that from any crossable tile you can move to any other. The first condition is enforced by following the path of the river and making sure that for each tile in both directions the adjacent tiles are the correct ones from a list of correct connections.

The second condition is checked through the use of graph theory, firstly the map is converted in a to graph form where the crossable tiles are nodes and the edges are the tiles that are crossable and they are adjacent to. The graph is then checked for the property of connectivity, their is a path between every pair of nodes. If the graph has the property it means every crossable tile is accessible and the condition is met.

The test was done by an exhaustive method that classes the map as unexplored and slowly explores all possible routes adding any newly discovered nodes to a map called reachable. If this map is the same size as the original map then the level fulfills the second condition.

If the map meets both of these conditions then it is valid and can be used in the genetic algorithm and potentially in gameplay if it is chosen.

4.3 Units

Units are created through the usage of object-oriented programming by having a parent class Unit, from which all the unit types inherit from. All information about a unit is stored within in this object, mostly using Integers and Strings where required. The only exception to this is the equipped weapon which uses a object called Weapon and an array of Boolean values determining what weapon types a unit can equip.

There are four methods in the Unit class that need specific mention that are not simply get or set methods. These methods are setOffensive, setDefensive, adjustBase, and levelUp, the first two are used to set up all the information required for combat calculations and the third is used to adjust the base statistics on generation of a new unit.

leveUp uses information known as “Growth Rates”, the likelihood of that statistic to grow for a unit to see if a the statistic increase upon reaching the next level.

For the drawing of the set of units to the screen, an adaption of the method for drawing the map is used. Instead of using a single dimensional array, a map of Units to Integers corresponding to their locations on the map is used. [1]

4.3.1 Generator

Generating units is done by randomly selecting from the list of possible units, excluding Lord, then adjusting their base statistics. This is done by randomly generating a number between -2 and 2 and then changing the units base stats for that instance of the object.

Units are generated with a base weapon, a weapon that is the worst in its type hierarchy. If a unit of a higher level than 1 is wanted, it is simply put through the level up process to the requested level. This means that the same unit is never likely to returned twice creating a more variable experience.

4.4 Weapons

To hold data on weapons an Object called Weapons is used, a base class of Weapon is used to declare and set up all the data that a Weapon has to hold. Then for each subtype of weapon class was created that inherits from Weapon, so Sword, Tome, etc... Then for every different weapon with in its type a child is created and connected to the parent type.

4.5 Gameplay

Different actions can be performed at different and specific times, to ensure that the player can not perform actions out of turn or at a time when they can not do that specific action. A system of while loops has been employed to ensure this. The loops create a series of “States” within the game, by doing this user inputs can be responded in an appropriate way dependent on state.

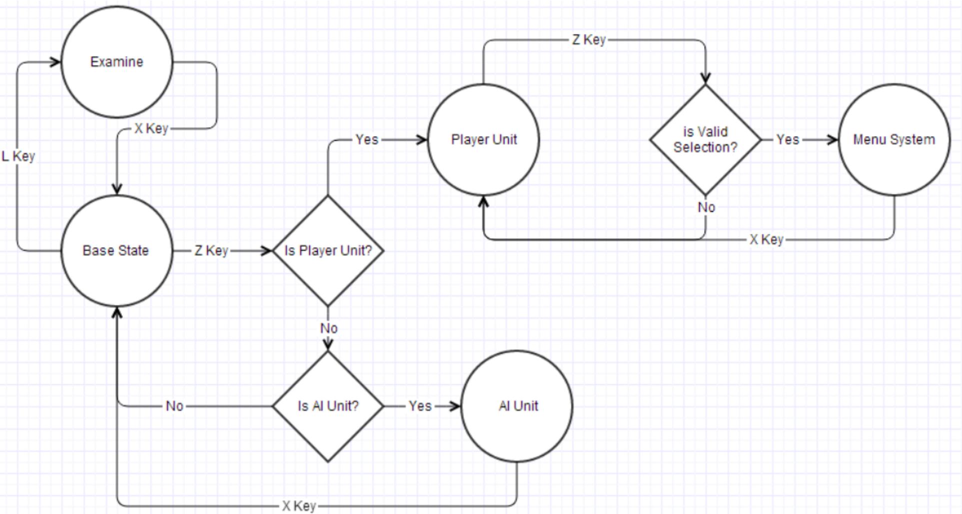

Figure 32: A simplified diagram showing some of the states that the player can be in during gameplay

To be able to facilitate gameplay functionality some specific purpose data structures were created, these mainly consist of maps to hold data about Units specifically or in general. For both the player and AI there is a map from Unit to Integer to state the location of the Unit within the single dimensional array used for the map.

As well as this to maintain what actions a Unit is still able to do in the player’s turn two other maps are used. These are moved and actionTaken both maps are from Units to Boolean values, if a Unit has performed an action, e.g. attacking then it will no longer be able to move. If the unit has only moved it will still be able to attack. When it is a player’s turn their unit list is loaded into moved and actionTaken. The actionTaken map also allows for checking if the turn of the AI or player has ended naturally, if every tuple in the map contains True for the Boolean value then whosoever turn it was, has ended

There are two other maps that are used for a single unit at a time and are to help display information back to the player about where a unit can move/attack,

part of the Clarity of Information objective. When a Unit is selected these two maps are populated. They are called moveSet and attkSet, which map Integers, representing a location, to Float, the cost of getting to that tile or the texture location respectively.

For calculating the moveSet, the set of tiles that a unit can move to from its location, a recursive algorithm is used. It looks at each tile around the current one and checks whether it can be moved to, if so and it is not in the set of movable tiles then the method is called upon using that tile as a start point. If the new tile wanted to move to is in the moveSet and it can be moved to, then it is checked if it is of lower cost that the current value, if so the method is called on that tile. Every other instance is culled, e.g. nowhere to move, run out of moves, can not move to that tile.

For generating the attkSet of a selected unit, the moveSet is simply iterated over and the surrounding tiles of each tile in the moveSet is checked. If the neighbour is not in the moveSet then it is added to the attkSet, note that if the range is 2 then a few additional tiles are added to the attkSet.

NB: Due to the usage of a single dimensional array user inputs must be converted from using an x, y coordinate system to fitting within a single dimensional array. This is done by using this formula:

$$new_coordinates = y * map_size + x$$

These maps also use a modified version of the Vertex Array method `[1]` to draw to a screen.

4.5.1 Combat

In terms of core gameplay mechanics all bar attacking are relatively uncomplicated, they simply require changing some data in a table and redrawing the screen. Attacking requires calculations to be made to decide the statistics of what can happen in the battle and this information needs to be relayed to the player. After selecting a valid unit to attack, all the base information is calculated both ways, what is meant by this is that setOffensive and setDefensive are used on both the aggressor and defending units. This is due to each unit being able to attack each other at least once.

From here two random number generators are used, the first is to see if the attack lands, if it number generated is less than the attacking units Accuracy the attack lands. After this another random number is given and checked against the Crit chance, if it is less than the Crit chance, the attack does 3x damage.

After this the damage is then dealt out to the units and the new health totals replace the old ones in the structures. If a units health becomes less than 0

then it is removed from the level. After this the exp is dealt out to the accordingly and any level up is applied if required. The exp dealt out depends on the actions performed:

- Being in combat grants 5 exp per attack
- Killing a unit in combat grants 20exp
- Killing a boss in combat grants 40exp
- Healing an allied unit grants 10exp

4.5.2 AI

The AIs that control the opposition forces to the player are very simplistic, they only have one decision to make. However these AI are not designed to be challenging at the moment simply be a element in the fitness function.

Before either of the AI was created a shortest path function had to be decided upon and implemented to allow the AI to effectively move it's units. Dijkstra's algorithm was choosen as the shortest path function for 2 reasons:

1. A* wouldn't work due to having uncrossable terrain, this means the more basic estimations techniques such as Manhattan distance wouldn't be monotonic. Therefore more complex estimations techniques would have to be made and with the time constraints this wasn't a viable option
2. Dijkstra's always finds the shortest path and is simple to program and adjust to the scenario of this game

After this the Shaman had it's on specific system coded as to make sure that it performs its role within the context of the game correctly instead of blindly moving into poor positions. The Shaman is also unaffected by whether the AI is to be passive or aggressive, as its focus is on finding and healing its allies.

Finally the two AI's where coded:

- Passive AI - it looks through all of its units and checks is a player unit is within one of these ranges, if not then the units remain where they are. However if there is a player unit with in range then it moves next to it and performs an attack. It simply selects the first unit it finds within the range of its own unit.
- Aggressive AI - it performs the same check as the passive AI, but when it does not find a player unit within its ranges it then moves towards the player's forces. If a player unit is within range it moves to attack and again like the passive AI selects the first unit it comes across.

For both the Shaman and Aggressive AI, they move towards their intended targets by calculating the shortest path to through Dijkstra's and moving as far along it as possible.

To add a small degree of complexity a modifier has been made called Ruthless, what this does it adds the priority ordering defined in the design. This means instead of moving to and attacking the first unit it finds, it completes searching the space and then selects the unit with the highest priority. If there are two or more of the highest priority unit in range then it randomly chooses one of the options.

4.6 Genetic Algorithm

The genetic algorithm uses a map from a defined data struture called potentialLevel to a Float, which is the evaluated score of that level. potentialLevel contains all the specific information about the level from the Terrain map, AI Units to AI properties. This map represents the population of the genetic algorithm, before anything is done it is filled and the levels are evaluated via the fitness function. The genetic algorithm works with a population size of 10.

4.6.1 Fitness Function

The fitness function was defined in the Literature Survey and for implementation currently the Unit evaluation is in place. The units are given a score by multiplying their level with an assigned value, listed below:

- Acolyte - 1.5
- Archer - 1.5
- Axeman - 1.0
- Cavalier - 1.0
- Defender - 0.5
- Lord - 2.5
- Shaman - 2.0
- Swordsman - 1.0

These values where arbitrarily assigned based on the AI priority order defined in the design, this was done as it is a simple way to give a logical numerical value to units. To compensate for the fact that the AI cannot have a Lord, the boss unit is given +2 to its score.

Having collected a value for the units in each force, the respective scores are then multiplied by a "Skill Level" for the player this was assumed to follow the

predefined pattern of the difficulty curve. Each of the AI's was given a value for skill, 3 for Passive, 6 for Aggressive. If the AI has the ruthless trait it is then given +4 to its score.

The AI's overall score is subtracted from the player's and this is normalised to be roughly between 0 and 1.

The planned map evaluation was designed to exhaustively search over the map searching for choke points, and then based on proximity to the player or AI give each a score before summing them up. If a choke point is closer to the player it recieves a positive score and if it is closer to the AI it recieves a negative score.

4.6.2 Selection

For selection two methods where specified [4] in the Literature Survey, and from these a modified version of the first method is used. Originally chromosomes outside an "ideal" range for the current searching value were culled to help produce better results.

However this isn't really necessary since when levels where being evaluated the majority are in the correct region. By also choosing not to cull any chromosomes in the population there will never by a case were an odd number remains causing a problem. So after evaluating levels they are selected by taking the first in the population and the last, these are removed before selection occurs again.

4.6.3 Crossover

Having selected two levels for breeding their children are created by generating a locus from 0 - 3. Then based on this number different aspects are taken from each parent and put into the child, as shown earlier in the Literature Survey. The combination from the parents in the child include the map, AI and AI units. The players units are not included in this as they should remain the same between every level.

4.6.4 Mutation

The two newly generated children are then put through mutation, this is done by generating a random number between 0 - 100, if the number is ¡ 15 then one of the components gets mutated. As specified below:

- **0-2** - The AI's nature is flipped, so passive become aggressive and aggressive becomes passive.
- **3-5** - The Ruthless modifier is flipped, if the AI was ruthless it is now not ruthless and not ruthless becomes ruthless
- **6-11** - The AI's units are regenerated

- **12 - 14** - The entire map is regenerated

The purpose of this mutation process is to stop the "gene" pool from stagnating meaning the algorithm generates a larger range of values as the population never becomes the same components being passed around.

4.6.5 Additional Information

On top of running through the standard procedures for a genetic algorithm it also performs some additional features to ensure a level is correctly generated. The first is to make sure for every child after going through mutation to generate new Unit locations for both the player and AI units. This is done to guarantee that no unit starts on Terrain that it shouldn't since their current locations may be in reference to another map.

To make sure that a level is actually output and the genetic algorithm doesn't infintely keep going, every 3 iterations the range of values being searched for is expanded. At first the system looks for a single designated value, held in an array where the ID corresponds to the level wanted. After 3 iterations this becomes a range of values from `wantedValue - 0.01` to `wantedValue + 0.01`, this range continues to expand by 0.01 in each direction every 3 iterations.

5 Testing

This game follows the standard testing procedures of software, by testing each module/part upon completion seperately and then after being integrated into the game simply playing it suffices to show if the components work together correctly. Most of the testing for this project was simply running the game to see the output in the game window and checking the console output if anything unsual occured.

5.1 Unit Testing

This is where each unit is tested individually as it is created.

5.1.1 Map Generator

```
Start Level Generation
up -> down
sPoint: 1
ePoint: 13
Difference: 12
devX: 25
devY: 8
diffBefore: 7
diffAfter: 5
maxDev: 4
deviation: 4
sPoint: 1
ePoint: 2
sPoint: 14
ePoint: 13
sPoint: 15
ePoint: 28
Difference: 13
devX: 12
devY: 21
diffBefore: 6
diffAfter: 7
maxDev: 4
deviation: -4
sPoint: 15
ePoint: 15
sPoint: 27
ePoint: 28
```

Figure 33: The console output when generating a river

What Fig.33 shows is the output to the console when a river is being generated, it displays information used in the creation of deviations being applied to the river. This was used to help see what values where being used when recursively applying the deviations. It helped with adjustment of parameters

for checks to say if there was any more space for another deviation.

```
Starting Ocean
Sea: 76
Chosen Start Point
Quarter Base: 19
Adding from 0 to 19
seaBase Size: 20
newHeight: 2
newHeight: 3
newHeight: 3
newHeight: 4
newHeight: 4
newHeight: 5
newHeight: 5
newHeight: 5
newHeight: 4
newHeight: 3
newHeight: 4
newHeight: 4
newHeight: 5
newHeight: 5
newHeight: 4
```

Figure 34: The console output when generating the sea

Fig.34 shows the console output when generating the sea. It lists the initial number of tiles set as well as the size of the base, in this case 19. Having then drawn the base, it moves onto recursively drawing inwards. As can be seen from the values it stays within the intended range of 4-5 as the amount inwards.

```
Mountains: 94
Placing start of new cluster at, x: 27 y: 22
Mountains: 92
Placing start of new cluster at, x: 25 y: 10
Mountains: 82
Placing start of new cluster at, x: 16 y: 2
Mountains: 76
Placing start of new cluster at, x: 12 y: 2
Mountains: 74
Placing start of new cluster at, x: 18 y: 8
Mountains: 69
Placing start of new cluster at, x: 20 y: 11
Mountains: 64
Placing start of new cluster at, x: 10 y: 12
Mountains: 56
Placing start of new cluster at, x: 18 y: 27
Mountains: 53
Placing start of new cluster at, x: 5 y: 28
Mountains: 47
Placing start of new cluster at, x: 14 y: 7
Mountains: 41
Placing start of new cluster at, x: 3 y: 13
Mountains: 38
Placing start of new cluster at, x: 11 y: 6
Mountains: 30
Placing start of new cluster at, x: 13 y: 25
Mountains: 28
Placing start of new cluster at, x: 12 y: 13
Mountains: 24
Placing start of new cluster at, x: 21 y: 21
Mountains: 20
Placing start of new cluster at, x: 1 y: 1
Mountains: 14
Placing start of new cluster at, x: 15 y: 15
Mountains: 10
Placing start of new cluster at, x: 18 y: 12
Mountains: 5
Placing start of new cluster at, x: 5 y: 15
```

Figure 35: The console output when generating a the mountain ranges

Fig.35 shows the output to console when generating the mountain ranges, as can be seen it slowly creates various different “clusters” across the map until it runs out of tiles to place.

```
Woods: 162
Wood x: 2, y: 17
Wood x: 18, y: 21
Wood x: 1, y: 22
Wood x: 5, y: 10
Wood x: 13, y: 7
Wood x: 18, y: 16
Wood x: 3, y: 8
Wood x: 7, y: 17
Wood x: 3, y: 10
Wood x: 22, y: 25
Wood x: 16, y: 11
Wood x: 2, y: 18
Wood x: 23, y: 28
Wood x: 27, y: 17
Wood x: 12, y: 24
Wood x: 9, y: 2
Wood x: 6, y: 7
Wood x: 17, y: 0
Wood x: 18, y: 10
Wood x: 6, y: 19
Wood x: 20, y: 17
Wood x: 10, y: 27
Wood x: 5, y: 11
Wood x: 8, y: 3
Wood x: 11, y: 23
Wood x: 1, y: 26
Wood x: 3, y: 19
Wood x: 6, y: 15
```

Figure 36: The console output when generating the woods

Fig.36 shows the output to console when generating woods, and as can be seen it places woods in random locations across the map. The output for Fort tiles is identical to Woods.

```
Changed Plains at (25, 9)
Changed Edge of VertBridge at (17, 14)
```

Figure 37: The output of the cleanup algorithm

Fig.37 shows the output of the cleanup method for the map, it only outputs the location and the type of tile that it had to change. This is so that the information can be checked when the map is examined in the game window.

```
Begun generating Graph representation
Completed Map
Beginning Validity Check
Passed the Connectivity Test!
```

Figure 38: The output to console for testing the connectivity of a graph representation of the level

```
Loc: 534 Terrain ID: 8
8: General
NID: 5
Pass 1
NID: 6
Pass Both

Loc: 535 Terrain ID: 6
6: General
NID: 9
Pass 1
NID: 8
Pass Both

Loc: 536 Terrain ID: 9
9: General
NID: 7
Pass 1
NID: 6
Pass Both

Loc: 538 Terrain ID: 8
8: General
NID: 10
Pass 1
NID: 6
Pass Both

Loc: 539 Terrain ID: 6
6: Right
NID: 8
Pass
```

Figure 39: The output to console when checking the flow of the river

Fig.38 and Fig.39 are both output to the console when checking the map is valid. These where used so that it was possible to identify that all the conditions where being checked and completed correctly, so that other operations that rely on having a valid map work without issue.

5.1.2 Unit Generator

```
Levelwanted: 1
Shaman Made
Base Attack: 0
New Attack: 0
Base Magic: 7
New Magic: 6
Base Defense: 3
New Defense: 1
Base Skill: 5
New Skill: 7
Base Speed: 7
New Speed: 9
Base Resistance: 7
New Resistance: 7
Levelwanted: 3
Archer Made
Base Attack: 5
New Attack: 3
Base Magic: 0
New Magic: 0
Base Defense: 3
New Defense: 1
Base Skill: 5
New Skill: 7
Base Speed: 7
New Speed: 7
Base Resistance: 5
New Resistance: 6
Attk: 3
Attk: 3
Magic: 0
Magic: 0
Max Health: 15
```

```
Lord Created
Base Attack: 5
New Attack: 6
Base Magic: 0
New Magic: 0
Base Defense: 5
New Defense: 6
Base Skill: 7
New Skill: 6
Base Speed: 7
New Speed: 6
Base Resistance: 3
New Resistance: 3
```

Figure 40: The output to console when units are being created

Fig.40 shows the output to console when any unit is wanted is created, it was used to test levelling up and adjusting the bases statistics at level 1. It was done by output the old and new values, as well as checking in game via the examine function.

5.1.3 Game Mechanics

Game mechanics such as combat, movement, examing units, were tested in game by simply playing the game to make sure they worked when intended.

5.1.4 AI

```
Aggressive AI
Use Ordering
AI TURN
Generating Attack Range/Move Range
Generation Complete
```

```
For every connection of Current
Connection: 0
Connection: 1
Node is unvisited
Connection: 2
Selecting new Candidate for current
Set List of Connections
For every connection of Current
Connection: 0
Connection: 1
Node is unvisited
Connection: 2
Selecting new Candidate for current
Should move to x: 14 y: 18
```

Figure 41: The output for the AI stating the properties it has, and then a unit applying Dijkstra's to find where to move

Fig.41 shows the debug statements for determining the properties of the AI as well as the process of working through the graph to find the shortest path, then returning the furthest for the unit to move along this path.

```
AI HEALER
UID: 20
Injured: 0
UID: 21
Injured: 0
UID: 22
Injured: 0
UID: 23
Injured: 0
UID: 24
```

```
UID: 38
Injured: 0
UID: 39
Injured: 0
Injured Enemy at x:-1 y: 0
```

Figure 42: The output for when the AI unit is a Shaman

Fig.42 shows what is output specifically on a AI's Shaman turn, due to it working

differently to other units. It outputs whether if finds an AI unit that is injured, the 0's represent false. The value -1 in x is used to alert it that there is no injured units. These statements helped test the ability of the AI to find and heal it's own units.

5.1.5 Genetic Algorithm

```
pSkill: 0.5
aU: 24.5
aiSkill: 10
pU - aU: -232.5
normal: 0.0199405
pU: 27
pSkill: 0.5
aU: 26.5
aiSkill: 6
pU - aU: -145.5
normal: 0.0302976
pU: 31
pSkill: 0.5
aU: 23
aiSkill: 7
pU - aU: -145.5
normal: 0.0302976
pU: 26.5
pSkill: 0.5
aU: 29
aiSkill: 3
pU - aU: -73.75
normal: 0.0388393
Finished Evaluation
```

Figure 43: The output to the console during the level evaluation process

Fig.43 shows the output to the console during the level evaluation process, this was used to compare to hand calculated results to see if they matched.

```
Selected first and deleted it
Selected last and deleted it
Set up both Children
Beginning Mutation
Regenerated Unit Locations
Beginning Player Unit Regeneration
Beginning AI Unit Regeneration
Beginning Player Unit Regeneration
Beginning AI Unit Regeneration
Set up for re-eval
Children becoming new Population
```

Figure 44: The output to console during the 3 stages of a genetic algorithm

Fig.44 shows the output during the 3 stages of a genetic algorithm, Selection, Crossover, Mutation, at then of each stage completion statements are outputted due to actually information being more difficult to print out succintly. Even these statements helped with debugging and testing allowing for bugs to be found by seeing what was a wasn't printed on crashing runs.

```
Adjusting Bounds
Looking for: 0.02
Checking if level is within bounds
pLevel: 0 Score: 0.0388393
pLevel: 1 Score: 0.0239881
pLevel: 2 Score: 0.0302976
pLevel: 3 Score: 0.0199405
Level Found
```

Figure 45: The output to console upon when checking if any level meets the criteria set

Fig.45 shows the output upon finding a valid level, not that in this case since the bounds of the search have been adjusted 0.019 as a score is accepted.

5.2 System and Acceptance Testing

The system as a whole was tested by playing through a couple of levels of the game by myself as well as 3 other people that I requested to play it, they were then asked to fill in a questionnaire. The topics on this questionnaire were based mainly on user experience to see if the project met the expectations of those that played it. The results of these questionnaires are contained in Appendix B, from these results the player experience is positive, with players feeling the concept of randomly generated levels has a lot of potential in the genre. They also felt that the levels were reasonable well generated.

Desipite this there was some dissatisfaction with the graphics of the game as

most modern players felt they were outdated, and "not stylised enough". As well as this players still found that sometimes the difficulty spikes on occasion.

6 Evaluation

This games success is measured based on a few factors:

- **Meeting Objectives -** The project met 75% of the stated objectives, based on what was completed and results of testing. The only objectives completely unmet:

  1. Inventory Management - This was left uncompleted due to the effort required for very little pay off, as the game functions well without this feature
  2. Evaluation of Maps - This is a large part of the genetic algorithms evaluation, however with the time constraints there wasn't time for it's completion. How the map effects difficulty in this project is under heavy assumptions, such as proximity to choke points mean an advantage, so it's incompletion doesn't effect the works of the evaluation to a large extent.

  The graphics created were within the objective bounds, however from the user experience the graphics was one of the issues players had with it, saying it "left a bit to be desired".

- **Enjoyability -** This is how enjoyable users find the experience of playing the game and the mechanics that are involved. This is a slight loaded gun of an evaluation measure since it does depend on if the user likes this type of game or not. Most players stated they enjoyed the games mechanics and feel however the AI was too easily exploitable.

- **Understandability -** Is the information displayed on screen enough, does it allow the user to fully grasp the exact situation? The project met this goal by displaying information on screen when required and matches several other games in the amount of information displayed.

- **Graphics -** This is not in terms of understandability but more aesthetic pleasure of the graphics, this is more of a personal measure for my own design ability. As stated this objective was 50/50 since the created sprites fitted the defined style but players felt it to simple.

- **No Difficulty Spikes -** It may be impossible to completely remove them, so it is important to look at who experiences them and in what scenarios. This allows for in future the Fitness Function to be improved.

7 Discussion

The main purpose of this project was to create a TBT game that randomly generated levels via the usage of a genetic algorithm, by doing so removing some issues with in the genre.

I feel that from the Testing and playing the game, the main focus of the project was beginning to come together. The game mechanics worked fine and information regarded these mechanics is displayed simply and cleanly to the players. On top of this the random generation works effectively creating a wide variety of interesting levels for a player to enjoy without them feeling to similar each time. I think the genetic algorithm in it's current state is a weakness of the project but it currently works well to help generate levels.

Given more time there are a few aspects I would like to improve:

- **More Effeciency** - Several of the algorithms used in this project are very exhaustive and take a longer time, given more time I would like to improve the algorithms
- **Map Evaluation** - This was not implemented and I would like to in the future have a working evaluation fully working.

8 Conclusion

In conclusion I believe the project was a moderate success, it proved that the concept of random level generation can be brought a genre typically considered very rigid and unappealing to random generation.

There is still more that could have been done on the project given the time:

- Fully completing the genetic algorithm and reducing the number of assumptions made for the Fitness function. This would make ithe genetic algorithm stronger and make random generation more attractive to this genre of games
- Addition of a Machine Learing component for analysing player skill, this would provide the player with a more personal experience making sure not just the average player but the majority of players can avoid frustration of difficulty spikes.
- Building a Game World, this would be adding to the user experience by giving background and lore to the game world.

I learned alot about random generation and that it isn't as easy as it seems to generate a level that makes sense, one that doesn't break the users focus due to it being unrealistic. I feel that it has given me a the ability to look at problems from a new perspective having come to understand genetic algorithms to greater degree and where they can be applied effectively.

9 Project Planning

Here is the Gantt Chart for the project:

	September	October	November	December	January	February	March	April	May
Proposal									
Map Generation									
Unit Generation									
Weapon Generation									
Core Mechanics									
Progress Report									
Fitness Function									
Genetic Algorithm									
Graphics									
System Testing									
Evaluation									
Final Report									
Poster Design									
Presentation + Demo									

Note that Unit/Integration testing has been factored in at the end of each stage. Graphics covers the entire development time as they was only made when required.

The Map Generator took longer than expected but due to this, more was thought about what the fitness function should actually contain. This lead to the current version in this report, time has been allowed for improvements and programming in the modules as discussed in Implementation.

10 Required Hardware and Software

10.1 Hardware

Development PC:

PC with Windows 8,
i7-4700MQ 2.40Ghz,
16Gb RAM,
64-bit Processor

Please note these hardware specification is for the laptop that the game will be mainly developed on it should be able to run on a computer/laptop that doesn't have the same power.

10.2 Software

Codeblocks - IDE being used for the project
C++ - Programming Languages (sfml graphics library)
Paint/GIMP - For graphical purposes

11 References

[1] Gomila, L. Designing your own entities with vertex arrays. http://www.sfml-dev.org/tutorials/2.0/graphics-vertex-array.php (accessed 23 November 2014).

[2] Gomila, L. Documentation of SFML 2.0. http://www.sfml-dev.org/documentation/2.0/ (accessed 23 November 2014).

[3] Mitchell, M. An introduction to genetic algorithms. Cambridge, England: MIT Press; 1999. http://www.boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf (accessed 05 January 2015).

[4] Wolfgang Banzhaf ... [et al.], *Genetic programming : an introduction on the automatic evolution of computer programs and its applications*, San Francisco: Morgan Kaufmann,1997

12 Appendix A

12.1 Minutes from First Meeting

Date: 02/09/2014
Time: 11:00am
Place: 3514 Prof. Sunil Arya's Office
Present: Pedram Amirkhalili, Prof. Sunil Arya
Recorder: Pedram

1. Approval of minutes:

First meeting therefore no previous minutes to approve

2. Report on Progress:

Nothing to report for Progress as first meeting to set up project

3. Discussion Items:

3.1 Overall Project idea, as well as if suitable idea
3.2 How to show the work that is being behind the scene

4. Goals:

4.1 Setup project on system, Pedram needs to write a description for the project
4.2 Figure out how to show the work off

5. Next Meeting

Will be decided within the next few days upon the completion of goals.

12.2 Minutes From Second Meeting

Date: 05/09/2014
Time: 11:30am
Place: 3514 Prof. Sunil Arya's Office
Present: Pedram Amirkhalili, Prof. Sunil Arya
Recorder: Pedram

1. Approval of minutes:

Minutes from first meeting have been approved, no amendments made

2. Report on Progress:

2.1 Project now on system, and assigned to Pedram
2.2 Work will be shown by showing the randomly generated levels and the process behind it

3. Discussion Items:

3.1 Usage of graphics and copyright laws
3.2 Structure of how the project is going to be accomplished
- Being changed build game first then add the algorithm after

4. Goals:

4.1 Research copyright law, begin prelim design for own graphics

5. Next Meeting

Not arranged as of yet, will be arranged if any problems arise

12.3 Minutes From Third Meeting

Date: 16/04/2014
Time: 2:00pm
Place: 3514 Prof. Sunil Arya's Office
Present: Pedram Amirkhalili, Prof. Sunil Arya
Recorder: Pedram

1. Approval of minutes:

Minutes from second meeting have been approved, no amendments made

2. Report on Progress:

2.1 Application was shown in a minor Demo
2.2 Everything up to parts of the Genetic Algorithm shown

3. Discussion Items:

3.1 Focus on polishing up the game or improving the genetic algorithm
3.2 Contents of Progress report and improvements to be made for final report

4. Goals:

4.1 Improve the Genetic Alogorithm
4.2 Polish some additional features

5. Next Meeting

This was the final meeting, no other meeting to be arranged

12.4 Minutes From Communication Tutor Meeting 1

Date: 13/02/2015
Time: 3pm
Place: Communication Tutor's Office
Present: Pedram Amirkhalili, Noorliza
Recorder: Pedram

1. Approval of minutes:

No minutes to approve since first meeting

2. Report on Progress:

Editting and adding to Proposal for the Progress Report has started

3. Discussion of Items:

3.1 Areas of Improvement from proposal
3.2 What the Literature Survey Should be
3.3 What is required for Implementation and adjustments to design

4. Goals:

4.1 Rewrite Literature Survey
4.2 Work through suggested improvements

4.3 Complete Progress Report

5. Next Meeting

After marks have been distributed for progress report

12.5 Minutes From Communication Tutor Meeting 2

Date: 13/04/2015
Time: 11:00am
Place: Communication Tutor's Office
Present: Pedram Amirkhalili, Noorliza
Recorder: Pedram

1. Approval of minutes:

Minutes from first meeting have been approved, no amendments made

2. Report on Progress:

Re-written large segments of Final Report as well as additions to some sections

3. Discussion of Items:

3.1 Areas of Improvement, such as the addition of figures and images to explain points
3.2 Explain everything more thoroughly
3.3 Discussion on placement of certain parts of the report

4. Goals:

4.1 Create Diagrams and Figures for the report
4.2 Work through suggested improvements
4.3 Complete Final Report

5. Next Meeting

Final required meeting, no more meetings scheduled

13 Appendix B

13.1 Questionnaire Response 1

Name: Dominic Brown

Age: 21

How do you rate your as a gamer from 1 - 5? Where 1 equates "don't game at all" and 5 is "play games daily"

5

How do you rate your experience in playing other Turn-Based Tactics (TBT) games? Where 1 equates to "never played a TBT Game" and 5 is "regular play TBT games"

2

How do you feel the concept of using random generation in TBTs stands up?

I thnk that conceptually it has a lot of merit, since it can provide a good way to make games like this extend longer as well as provide an additional game mode outside of the standard campaign. However it is the sort of thing that would be difficult to control and manage effectively I feel, purely due to how random generation works.

How did you find the gameplay?

The gameplay was okay, had all the necessary features to function, though the it does need more. Also for an experienced gamer the AI's didnt provide any real challenge at any point in the game

Did you feel like the difficulty ever spiked causing you frustration?

No, but the difficulty never really climbed to greatly to begin with

How did you find the graphics? Where they pleasant? How was the style?

The graphics felt a little blocky and left a lot to be desired due to their simplistic nature

Did you feel you were given enough information on screen when trying to perform actions?

The amount of information given was on par with other TBT games I've played

so no complaints there.

Any other comments?

Overall nice, concept good progress and the random generation worked better than expected generating quite a few good levels.

13.2 Questionnaire Response 2

Name:Ali Amirkhalili

Age: 51

How do you rate your as a gamer from 1 - 5? Where 1 equates “don’t game at all” and 5 is “play games daily”

1, I don’t play any games at all!

How do you rate your experience in playing other Turn-Based Tactics (TBT) games? Where 1 equates to “never played a TBT Game” and 5 is “regular play TBT games”

1

How do you feel the concept of using random generation in TBTs stands up?

From what I’ve been told it sounds like the idea has some potential to provide something extra to a genre of games, but I would guess the execution would have to be very good in order to pull it off.

How did you find the gameplay?

Nicely paced, and everything seemed to as intended

Did you feel like the difficulty ever spiked causing you frustration?

After the first level the AI changed to be a lot more aggressive, it caught me rather off-guard. Beat it on my second attempt other than the difficulty didn’t cause any problems.

How did you find the graphics? Where they pleasant? How was the style?

I assume it is going for a stylised look and it looks okay, though I think it could have been made to look a bit nicer!

Did you feel you were given enough information on screen when trying to per-

form actions?

The information displayed on screen was good and concise, really helped to see the information laid out in front of me as I got the hang og the game

Any other comments?

Hopefully one day I can play a full version

13.3 Questionnaire Response 3

Name:Danny Foode

Age: 21

How do you rate your as a gamer from 1 - 5? Where 1 equates "don't game at all" and 5 is "play games daily"

5

How do you rate your experience in playing other Turn-Based Tactics (TBT) games? Where 1 equates to "never played a TBT Game" and 5 is "regular play TBT games"

1

How do you feel the concept of using random generation in TBTs stands up?

I think it's a cool concept, I've never played a TBT game but I think that random generation would make me more interested in playing as it add something new to the game

How did you find the gameplay?

Pretty smooth, no issues everything worked the way it was laid out on screen.

Did you feel like the difficulty ever spiked causing you frustration?

Not really.

How did you find the graphics? Where they pleasant? How was the style?

They looked okay, but I think that they aren't stylised enough for the type of graphics that the game was aiming for

Did you feel you were given enough information on screen when trying to perform actions?

Yeah all the information is displayed nicely in the right place, however I think that the boxes could have less bright colours and textures themselves.

Any other comments?

The random generated levels were really cool, it was interesting to look around the level and see a river weaving round the map.