NOTTINGHAM TRENT UNIVERSITY

SCHOOL OF SCIENCE AND TECHNOLOGY

Turn based strategy game developed in Java, with an AI opponent

By

JAMES JONATHAN DUNN

In

2018

Project report in part fulfilment

of the requirements of the degree of

Bachelor of Science with Honours

In

Software Engineering

I hereby declare that I am the sole author of this report. I authorize the Nottingham Trent University to lend this report to other institutions or individuals for the purpose of scholarly research.

I also authorize the Nottingham Trent University to reproduce this report by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

James Jonathan Dunn

ABSTRACT

The goal of this dissertation is to explore the development process of building a computer game environment from scratch. The project utilises LWJGL(Lightweight Java Graphics Library) a Java library built on the OpenGL framework.

It explores the way a game engine should be designed and built so that is can be used more many more projects in the future. The implemented game utilises the environment built to show how a turn based strategy game could be designed.

A computer AI has been designed to try and beat the game best it can, the AI is modelled on the 3 main principles of behaviours trees.

Behaviour trees determine how and when an AI unit will move according to the 3 principles of survival.
Physiological routines are routines that need to be executed every time otherwise the game or unit will not survive.
Subsistence routines are routines that need to be executed once the living conditions have been met so the long term existence is secured.
Aspirational routine are executed after the subsistence has been secured and before subsistence needs to be executed again.

The project is successful in developing a modular game environment which could be used for variety different game types, and the implemented game provides a clear environment to test the AI built.

ACKNOWLEDGEMENTS

I would like to thank Dr. Michalis Mavrovouniotis, who provided invaluable support throughout the duration of this dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Introduction

This thesis describes the learning process that led to the development of a game engine from which the implemented game was built on top of. The environment was built in such a way that it can be adapted for future game projects that are widely different from the one demonstrated in this project.

Over the next 6 chapters this document will explore what is necessary to create a video game from scratch with no prior knowledge of the graphics library being used. It will determine the best way to create a game level and how a user can interact with that world to play a game.

Research into other similar games has been carried out so gain an idea of what the current market look likes and so that the product produced is not an exact copy of an existing one.

The computer AI has been built specifically for the game demonstrated but at the same time it is not too overly reliant on the design of the game environment and interactions are based more on how the opposing player moves and acts as to allow for expansion in multiple game scenarios.

The use of behaviour trees determine how and when an AI unit will move according to the 3 principles of survival. Physiological routines are routines that need to be executed every time otherwise the game or unit will not survive. Subsistence routines are routines that need to be executed once the living conditions have been met so the long term existence is secured. Aspirational routine are executed after the subsistence has been secured and before subsistence needs to be executed again.

CHAPTER 2

CONTEXT

2.1 Background research

2.1.1 Game development

The games development industry and has been going strong for over 30 years now, starting with arcade machines and adapting through the years until you can game on your mobile now. It is important to evaluate existing projects to gain knowledge of what the market competitors look like. For a large scale commercial development of a game, requirement analysis should take into consideration all possible stakeholders, however, this could be should a large group of consumers it would be feasibly too difficult to get a fair representation for this project.

2.1.2 Similar games on the market

Warhammer 40,000: Regicide is a turn based strategy game developed for mobile and adapted for PC. In it you control a group of units much like chess and have to destroy the enemy opponent. The structure of the game is very similar to aims of this project in that one unit at a time can be moved around a grid based environment. When a unit is selected all their possible moves are highlighted on the ground with red tiles representing enemies. However the game only seems to have one set map that although visually changes, it does not change in size of definition.

2.1.3 2D vs. 3D development

Both 2D and 3D development have their initial learning curves to overcome before development can properly begin. The original concept was presented as a 2D product but 3D still needed to be considered. The main problem with 3D development is how much longer it would take to learn any 3D API and produce a product in the given time frame. It could have been completed using a game engine but the purpose of this thesis is to explore the development process from scratch so 2D seemed a lot more feasible.

2.2 Which API should be used?

When people think of the openGL library they think of the standard C++ version of the library. This is a very popular library for rendering all types graphical products not just games development. Having never used openGL in any capacity before but being fairly fluent in C++ it seemed like an obvious decision to use them hand in hand to build the game. However, are some preliminary research it became apparent how popular this field was and how many similar projects were already out there. Therefore it was necessary to come up with a way to make this project different from the rest. To make the project more interesting other languages were considered as openGL is a cross platform API so it is not constrained to just using C++. Despite not having much experience in it Java seemed like the most suitable language to use for the project, so before starting it was required to become familiar with Java and openGL as to be able to produce the product in the given time frame. Most of the other projects have been built in C++ so any of them I use to help will need to be converted into Java before they can be used.

Before finalizing the use of LWJGL other java graphics libraries were considered, firstly what platform the project was going to be designed for as there are many suitable libraries for building games in Android. There are also many game engines that could have been used to develop this project such as Unity, Havok and Godot, however, that aim of the project was to design the game from scratch without any high level tools as the focus of the project is to investigate the development process which an engine would do all the heavy lifting for.

2.3 Limitations

2.3.1 Limitations of platforms

When considering what platform to build the product for a wide range of possibilities were examined. For the last 10 years the mobile games market and grown dramatically and become a major competitor in the games industry. Mobile phones and tablets are a common way for people to play casual games on limited hardware. As this is an investigation into the process of developing a game rather than making a 'good' game to be marketed the focus way more on building a PC game as there is already a lot of information on designing and building for PC. In addition there is already a steep learning curve to this project adding another element such as mobile development would make the project too overly complex and not create a good enough project in the time frame given.

2.3.2 Limitations of current AI

One of the first AIs to take on a human at a game and win was Deep Blue, created by IBM in 1996 it could calculate 50 billion position in 3 minutes and by 1997 it could calculate 200 million moves a seconds and beat the world champion at the time. The biggest problem with trying to program an AI is that you may know all the possible moves on the table and it can work on probability and work out where the human player will go but it can't predict randomness, and unless built correctly will struggle with adapting to a situation it hasn't been explicitly told what to do in.

2.3.3 Limitations of game engines

Although the vast majority of games are developed in an already established engine this can limit what can be created because you must adhere to the rules of that engine, there are many game engines such as unity or unreal engine which simplify the game building process by giving the creator access to high level object that come with pre built properties. Although you can go into and edit what these do for yourself, for this project it would be most suitable to create any objects needed from scratch.

2.4 Creating an artificial intelligence

2.4.1 Existing products on the market

AI was introduced as a computer opponent many years ago and since then has become prevalent in the majority of games. Nearly every game on the market has some form of AI player in it, but the one I have chosen to look at here is a simple flash game called Age of War. This is a real time strategy game in which you create one of 3 units which have to travel across the map and destroy the opponents base. For each unit killed you get gold which you can then use to upgrade to the next 'age' which starts with cavemen, then moves to medieval and so. Here the AI constantly has to decide which unit to create next based on how much gold it has, how much HP its tower has left and where the opponent is. This must be constantly computed and the most important decided moved to the top of the queue.

2.4.2 How an AI thinks

A behaviour tree is something that can be used to map out how an AI reacts to its current situation. The 3 principles of survival state which action should be performed next. Physiological routines are routines that need to be executed every time otherwise the game or unit will not survive. Subsistence routines are routines that need to be executed once the living conditions have been met so the long term existence is secured. Aspirational routine are executed after the subsistence has been secured and before subsistence needs to be executed again.

So in terms of a game such as Age of War the physiological routine would be to protect the base at all cost because if that gets destroyed the game is over. Once it knows it's base is safe, such as the enemy is not close to attacking it then it will look at the subsistence routine, these are to ensure it will survive in the future so this will mean creating new units that are stronger or equal to the ones created by the other player so to make sure the other player isn’t getting closer to their base. If these are both find the aspirational routine is creating a new unit in order to attack the player, if the AI is already winning or has excess gold it may consider doing this.

CHAPTER 3

NEW IDEAS

3.1 Specification

After researching existing products on the market, and looking at various concepts that other people have put forth for games the specification for the game has been narrowed down to the following.

The game will be a turn based strategy game, meaning each player takes their turn to move a unit at a time then it is that other players turn again, like a game of chess. However, to make it more interesting the users will move all of their units each turn rather than just one. This will create a deeper strategy because each unit will need to be considered individually each turn and many more possible moves open up. If players fail to keep check of all the opponents units they could quick find themself in danger of losing the game. Each opponent will have a tower where their units spawn from, to win this must destroy their opponents tower.

The game will be built in Java using the LWJGL api as previous discussed last chapter to make the project more challenging and stand out from all the games developed in C++ using openGL. The world for the game will be created using a tile engine, which will read a .png file and take the RBG colour for each pixel and based on that create the correct tile, this is explained in further detail later in the report.
There will be 3 units in the game, one weak but fast unit, one fast but strong unit and one which is a balance or fast and strong. This means various combinations of units can be created to try and defeat the opponent. More units may be added later depending on time constraints.

The AI shall aim to defeat the opponent as quick as possible, focusing on the objective of destroying the users tower rather than just trying to kill their units.

3.2 Requirements Specification

3.2.1 Requirements structure

A standard for requirement specifications is outlined by Sommerville (1). The two types of requirements explained are functional and non-functional requirements. Under the umbrella of non-functional are usability and user requirements which go into greater depth about how the user interactions with the system.

Below are the functional requirements for the project, given the time frame of the project there are two types of requirements:
Important ones that are styled as "The system must…".
Ones that will be added in or improved on once all the others are implemented fully styled as "The system should…"

3.2.2 Functional requirements

1. The system must allow the user to easily navigate around the game environment

1.1 Since the release of Quake® in 1995 most state of the art games use the WASD keys in order to navigate around the world. It was therefore chosen to move the camera around the screen using these keys.

2. The system must allow the user to control their units simply

2.1 The game must make it easy for users to do exactly what they want to, hiding controls of abilities behind menus or releasing on unconventional key binding can be confusing for new users.
2.2 The game must use only mouse input to select and move units around the screen.

3. The system must correctly render the type of tile in the correct location

3.1 The system must read in the png file and from it be able to create the corresponding map

3.2 The system must allow for the tiles to be changed on the fly while the game is running.

3.2.1 This is necessary to change tiles to selected.

4. The system must allow for 2 human players, 2 AI players or 1 of each

4.1 The system must allow for a selection screen between the 3 options

4.2 The system should allow for 2 AI players for demo purposes

5. The system must ensure the AI conforms to the rules of the world

5.1 The system must make sure the AI does not move onto a invalid tile

5.2 The system must ensure the AI can't move onto an already occupied space

6. The system must critically decide how to move the AI

6.1 The system must look at the board relative to the currently selected unit.

6.2 The system must check the location the AI is moving too is safe from enemy attack.

6.3 The system should look multiple moves ahead to plan its strategy.

6.4 The system should plan all of its units move in conjunction with each other before moving.

7. The system must ensure the AI has completed its turn before moving on

7.1 The system must check each of the AI's units have moved.

7.2 The system must ensure the AI have created and moved any new units if it need to.

3.2.3 Non-Functional Requirements

1. The system must not fail at any point

1.1 The system must be rigorously tested to ensure it doesn't fail at any point

1.2 Multiple scenarios must be played out to try and crash the system to see where any weak points are.

2. The gold tracker must be accurate

2.1 The game must correctly update the gold tracker when a unit is killed and at the start of each turn

3. The system must accurately keep track or units positions on the screen

3.1 When a unit is killed its previous position must be set empty in the world matrix

3.2 If a unit kills another and moves into its position then both units must be updated to reflect that.

4. The system must be able to run on a range of PC's

4.1 The game should aim to run on the vast majority of PC's as it will not be a very graphically intense game.

5. The system must have a clear and concise user interface

5.1 When looking at the system it must be clear how to interact with everything on screen and manipulate object in the world.

5.2 The control layout implemented must be accessible, logical and uncomplicated.

3.3 Requirements conclusions

On a project without such tight time constraints a prototype could be created and tested with users to gauge their response, however, there was only time to quickly sketch an idea of what the game will roughly look like to ask people their opinions on the design. Below is the image produced.

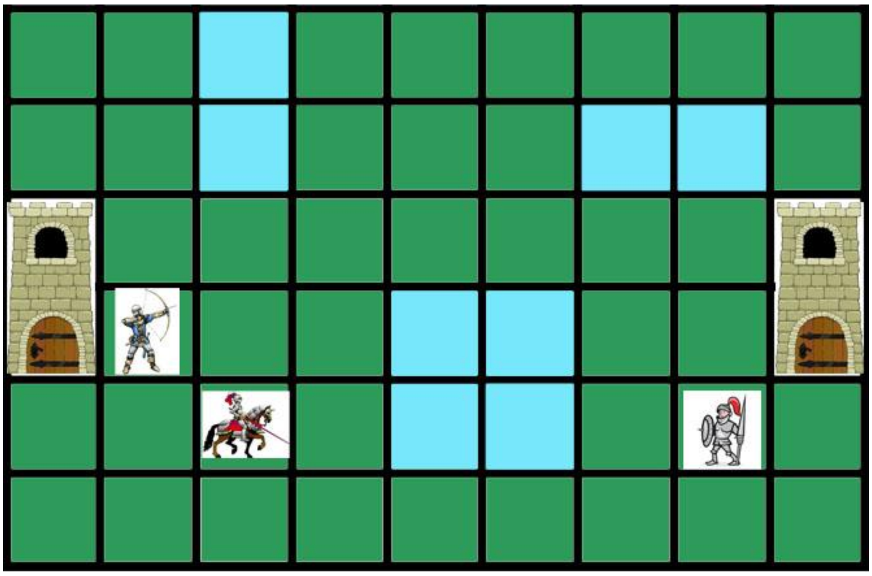


FIGURE 1 : Rough UI Design

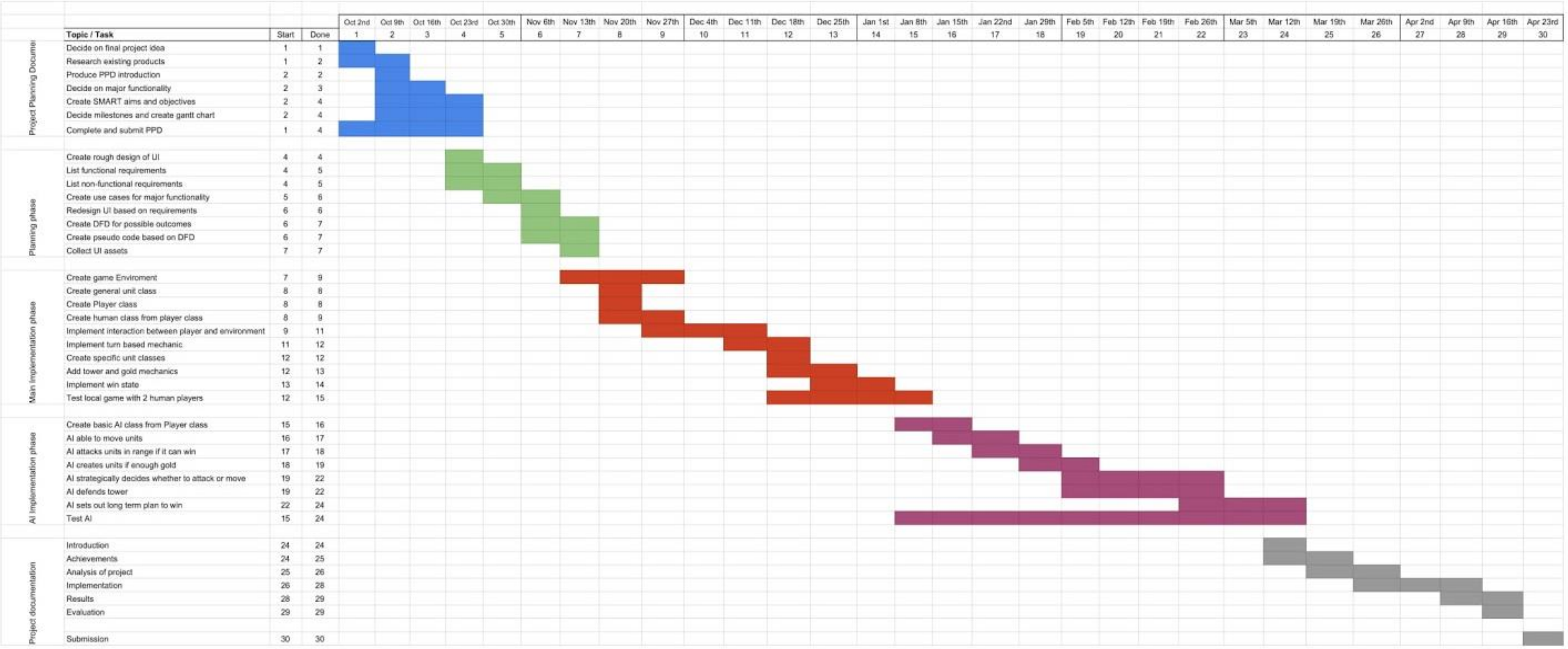
From the small group asked their opinion on the design the majority was positive. However, people said the board look a bit small and that the final game needs to look more visually appealing. This is only a mock to give an idea of what the product may look like.

The requirement specification has concluded what features need to be included in the game; the next chapter goes into detail as how the design and implementation process goes.

3.4 Project planning

Below is the gantt chart created to track the progress of the project. It is broken down into sections so each needs to be completed before moving onto the next. The AI section has been given significantly more time than it will actually need so if the project gets behind track the slack can be picked up here.

3.4.1 Gantt Chart



	Topic / Task	Start	Done
Project Planning Documei	Decide on final project idea	1	1
	Research existing products	1	2
	Produce PPD introduction	2	2
	Decide on major functionality	2	3
	Create SMART aims and objectives	2	4
	Decide milestones and create gantt chart	2	4
	Complete and submit PPD	1	4
Planning phase	Create rough design of UI	4	4
	List functional requirements	4	5
	List non-functional requirements	4	5
	Create use cases for major functionality	5	6
	Redesign UI based on requirements	6	6
	Create DFD for possible outcomes	6	7
	Create pseudo code based on DFD	6	7
	Collect UI assets	7	7
Main implementation phase	Create game Enviroment	7	9
	Create general unit class	8	8
	Create Player class	8	8
	Create human class from player class	8	9
	Implement interaction between player and environment	9	11
	Implement turn based mechanic	11	12
	Create specific unit classes	12	12
	Add tower and gold mechanics	12	13
	Implement win state	13	14
	Test local game with 2 human players	12	15
AI Implementation phase	Create basic AI class from Player class	15	16
	AI able to move units	16	17
	AI attacks units in range if it can win	17	18
	AI creates units if enough gold	18	19
	AI strategically decides whether to attack or move	19	22
	AI defends tower	19	22
	AI sets out long term plan to win	22	24
	Test AI	15	24
Project documentation	Introduction	24	24
	Achievements	24	25
	Analysis of project	25	26
	Implementation	26	28
	Results	28	29
	Evaluation	29	29
	Submission	30	30

FIGURE 2 :Gantt chart

CHAPTER 4

IMPLEMENTATION

4.1 How the game plays

The object of the game is for one opponent to defeat the other, the first to do so will win the game. You do this by destroying their tower first using your units. To create a new unit the player presses the keyboard keys 1,2 and 3. A unit is selected by placing the mouse over it and click it, then once selected the available moves will be displayed and the player will click on one of these to move there, if it is free they simply move, if it is occupied both units attack each other, it is a tower that unit attacks the tower for their attack damage. Once a unit has moved it cannot move again till next turn once the user is done moving all they want too they hit the Enter key to end their turn and the game moves onto the next player.

4.2 Building the world

4.2.1 The tile engine

The play area for the game is made up of a grid of tiles which for this game are either displayed as water, grass or a tower image. A tile renderer takes the information about each tile to produce the grid seen, using individual tiles rather than a predesigned game level means the map could be easily changed and adapted to multiple scenarios or even used in a different game all together. When the buffered image is loaded into the program each pixels colour profile is taken and using the RGB 255 colour system a number is returned. Each tile type has a value between 1-255 which loads into correct texture, in the game created only 1-4 have been necessary. The tile renderer is then given a matrix2f containing what type each tile is and from this is able to produce the map seen.

4.2.2 The different units

There are 3 different units in the game, a light unit, a medium and a heavy power unit. The 3 units are derived from a base unit which implements how they all move and act in the world. When a new unit is created its corresponding texture is loaded and scaled to the world. Many more units can be easily added with their own special functions because of the way the base class takes care of all the relation to the world.

4.2.3 The camera

The camera is what the user can currently see of the world, it will depend on what the current size of the window is, as to how much of the world needs to be rendered at any one time. The game has a default window size of 640x480 so it will be usable on the majority of computer without any problems however, the window can be expanded and the world render will compensate. When the user makes the window larger or small a function is called to adjust the camera projection matrix so we only render the size of the current window and not larger or else everything will be squashed.

The camera can be moved around the to view the rest of the world using the WASD keyboard keys as further explain in the user input section. The the camera is moved from the edge of the map we then have to calculate the offset from the edge of the map to how many pixels the camera has been moved so when can render the correct portion of the world. The camera also provides basis for the user input because we need to know if the user is clicking on 0,0 if that is actually 0,0 or if the camera has been moved and they are actually clicking on a different tile.

4.3 Player interaction

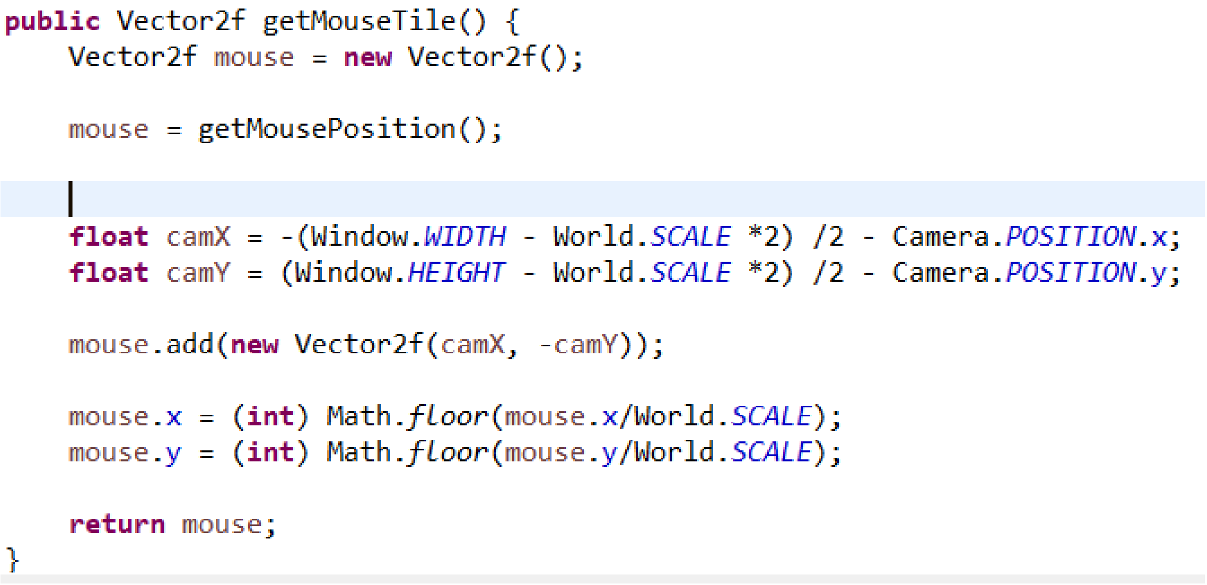
The user can interact with the game using both keyboard and mouse input. Most of the complex interactions are driven by the mouse such as selecting and moving a unit.

4.3.1 Getting user input

While the game is on the users turn it is constantly checking for input from the mouse and keyboard, firstly to see if the camera has been moved. The camera needs to be checked first so if the user does click on a tile we get the correct tile due to the offset. Next it is checked if the number keys 1,2 or 3 have been pressed. These are what create the units in the game, explain in further detail on the next section.

Now we wait for a mouse input, we are using GLFW part of the openGL library to poll for input and return true when the user selects a tile. There was an issue with this section of the code were by it was being triggered when the mouse was pressed down and therefore being run multiple times before the mouse was released. This was changed so a click is registered as the mouse was just pressed down and is now released.

When a user clicks the mouse the position of the mouse is recorded and we need to convert this into the tile which they have clicked. To do so the following algorithm is used to convert the window cords into a tile in the world.



```
public Vector2f getMouseTile() {
    Vector2f mouse = new Vector2f();

    mouse = getMousePosition();

    float camX = -(Window.WIDTH - World.SCALE *2) /2 - Camera.POSITION.x;
    float camY = (Window.HEIGHT - World.SCALE *2) /2 - Camera.POSITION.y;

    mouse.add(new Vector2f(camX, -camY));

    mouse.x = (int) Math.floor(mouse.x/World.SCALE);
    mouse.y = (int) Math.floor(mouse.y/World.SCALE);

    return mouse;
}
```

FIGURE 3 : getMouseTile extract

Once we know which tile in the world the user is clicking on we can run the current.selectUnit function which searches the user's myUnits array and checks if they have any units on that tile. If they do have a unit there and it has not yet moved that turn then we 'select' it. Selecting a unit means we change all the tiles in that units MoveRange to selected tiles so show the user what there possible moves are. This will not select tiles in which the user's other units are positioned. Clicking anywhere but on these selected tiles will cancel the unit selection.

4.3.2 Collision detection

When the user selects a tile to move too that already has an enemy unit on it, they are declaring an attack on it, units cannot share tiles. Once the world class has returned the baseUnit at that location the attack damage of the players current unit is applied to the enemy's health and vice versa. We then have to check if either of the units in the exchange are now dead and we need to update the location of them in the world array.

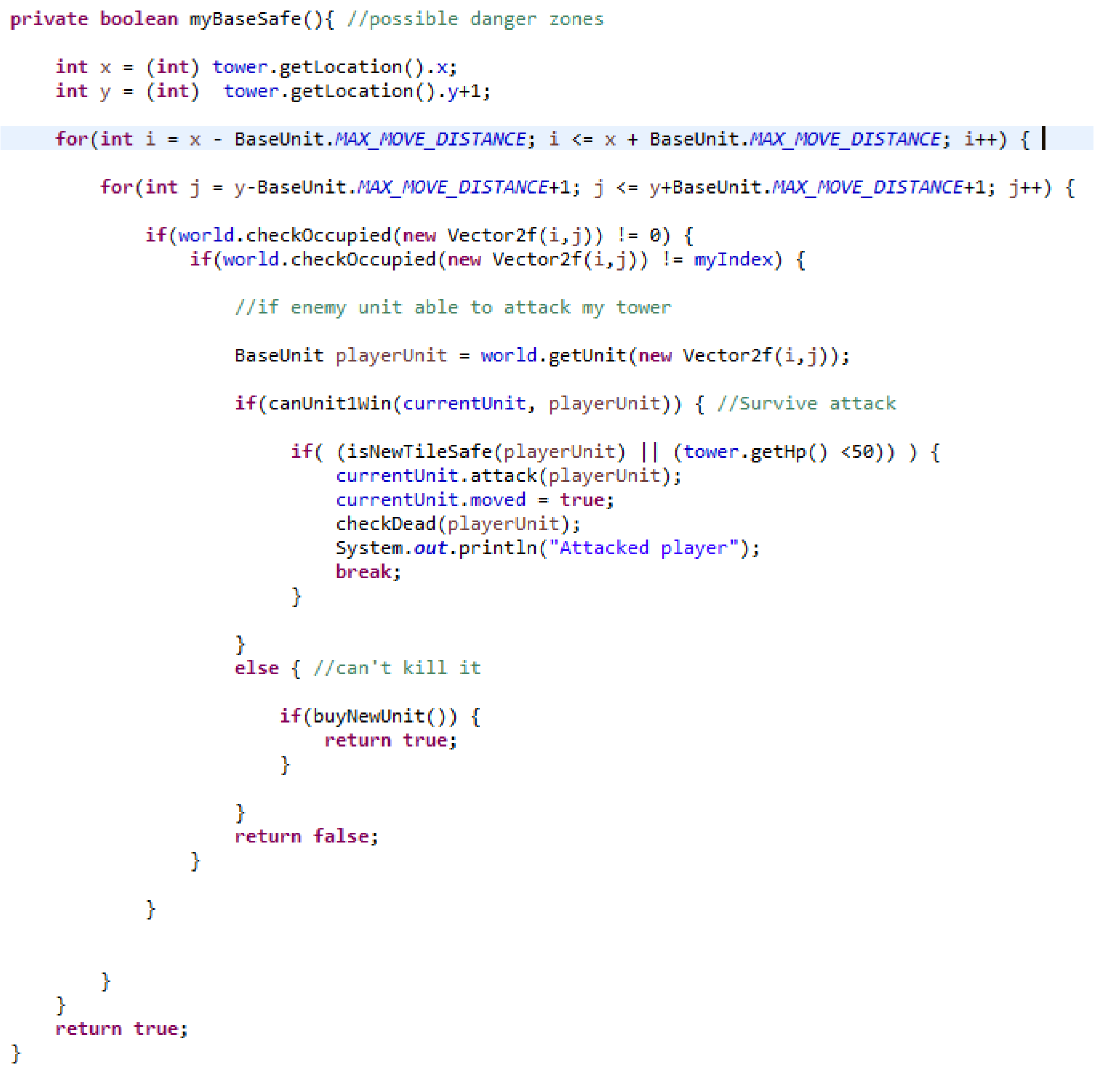
4.3.3 Unit creation

Units are created when the player presses the 1,2 or 3 number keys each creates a different of the 3 units available. When one of them is pressed a check is ran to make sure they have enough gold to create that unit before it is placed in their tower. If there is already a unit sitting on the tower it must be moved into play before one can be built, a unit could be built and kept sitting on the tower where it can't be attack as a tactical move.

4.4 Creating the Artificial Intelligence

4.4.1 AI survival

For the AI to successfully move around the board and complete its task of winning the game a few core functions needed to be designed. First to complete the physiological needs of the AI it needs to make sure it’s base is safe from enemy attack. It does this by running the myBaseSafe function before it decides to move each unit.



```
private boolean myBaseSafe(){ //possible danger zones

    int x = (int) tower.getLocation().x;
    int y = (int)  tower.getLocation().y+1;

    for(int i = x - BaseUnit.MAX_MOVE_DISTANCE; i <= x + BaseUnit.MAX_MOVE_DISTANCE; i++) {

        for(int j = y-BaseUnit.MAX_MOVE_DISTANCE+1; j <= y+BaseUnit.MAX_MOVE_DISTANCE+1; j++) {

            if(world.checkOccupied(new Vector2f(i,j)) != 0) {
                if(world.checkOccupied(new Vector2f(i,j)) != myIndex) {

                    //if enemy unit able to attack my tower

                    BaseUnit playerUnit = world.getUnit(new Vector2f(i,j));

                    if(canUnit1Win(currentUnit, playerUnit)) { //Survive attack

                         if( (isNewTileSafe(playerUnit) || (tower.getHp() <50)) ) {
                             currentUnit.attack(playerUnit);
                             currentUnit.moved = true;
                             checkDead(playerUnit);
                             System.out.println("Attacked player");
                             break;
                         }

                    }
                    else { //can't kill it

                        if(buyNewUnit()) {
                            return true;
                        }

                    }
                    return false;
                }

            }


        }
    }
    return true;
}
```

FIGURE 4 : myBaseSafe extract

This function gets the tower location and takes the move distance of the fastest unit and loops through each tile is range. This means it gets the potensial furthest away an enemy unit could be right now for each of these coordinates we run the checkOccupied function from the world class, the world class contains a 2D array where each element in the array contains the details for that time such as if there is a unit on that tile. So it there is an enemy unit on one of the tiles in range then we get that unit so we can test if we can kill it.

The function canUnit1Win takes the current unit the AI is moving an compares the attack damage of the players unit against it's units health and vise versa to determine if it's possible to kill it. It also has an OR game were by if the AI's tower's health has dropped below 50HP (half the starting value) the it decides it's worth killing the unit even if that unit then kills it's unit.

checkDead simply checks if either of those units were killed in that exchange and if so the winner moves onto the place where the other was killed. It also needs to update the world class so it can place the units in the new correct position.

4.5.2 When to attack

The AI attacks under 3 circumstances; It's tower is under threat, it can kill a unit and range and safely move there, or it is attacking the enemy base. It has been designed in such a way to try and complete the objective above all else so will not unnecessarily attack an enemy when it could get closer to their base. It a player unit is within attack range of the AI's tower then the AI will decide whether to attack it or not. Firstly if it can be killed and the AI unit survive it will do so, if it cannot survive the attack but kills the other unit at the same time it considers this a win because it saves the tower from attack next turn.

If there are no AI units in range of the player near the tower or the ones that are aren't strong enough then the AI will check if they have enough gold to create a new unit and using that one or a combination of them kill the player unit and do so.

If in range of the enemy tower and no other more powerful enemies are in range then it will attack it, if one that could kill it is near then it will see how much damage it can do to the tower and if it can significantly lower the tower hp it will continue to attack if it even if it might get that unit killed.

4.5.3 When to move

Before the AI decides to move, either because it plans on killing a unit or because it can't do anything else, it will look at the tile it is planning on moving to and check the surrounding area for potential threats that could kill it if it moves to that position. If it can kill the unit on that tile then it will have to move there once it is dead. So it checks if it was at that location with it's new HP from just attacking that unit if it would survive another attack. If it can then it will move onto the tile, if not it will move safely instead. Below is the behaviour tree created to roughly show how to AI decides what to do each turn. You have seen above it how each turn for the AI is run and compare it to the tree to see how it makes its decision.

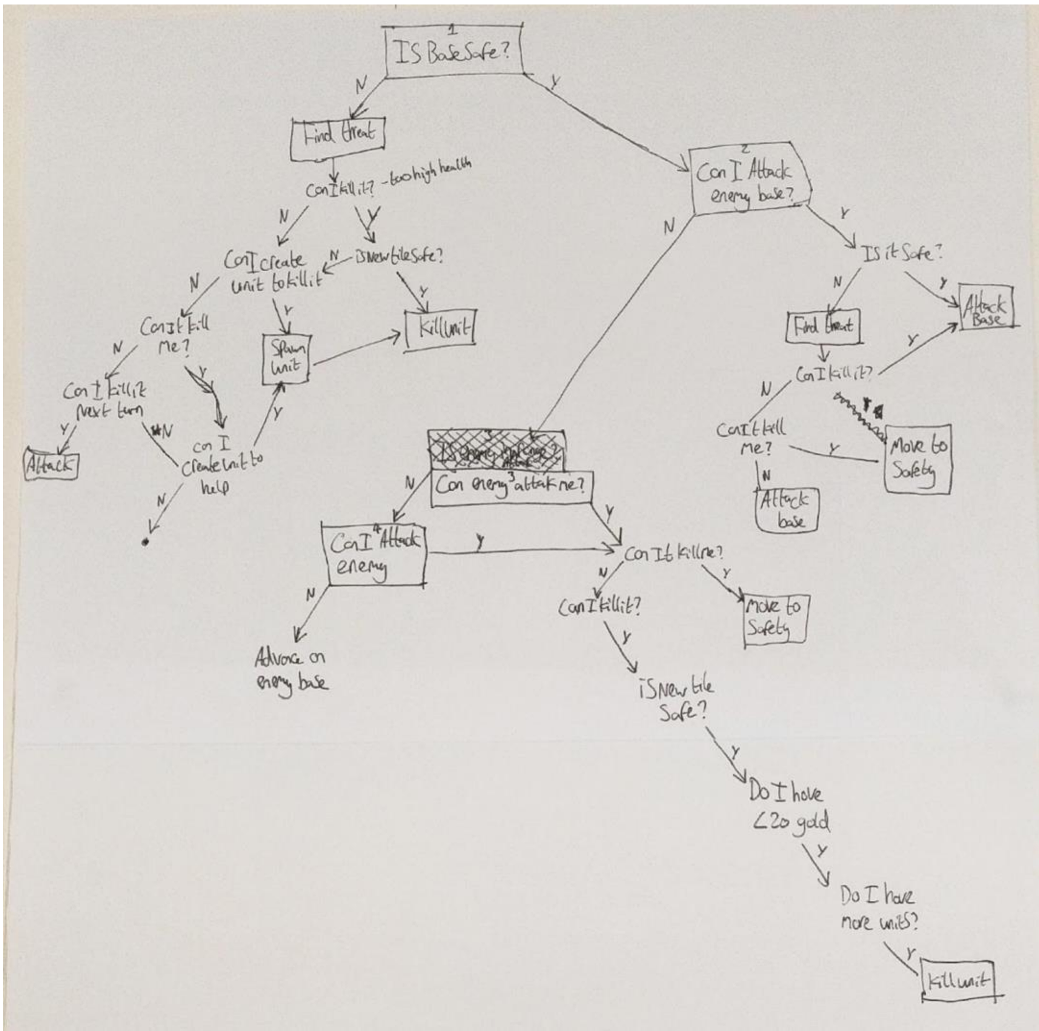
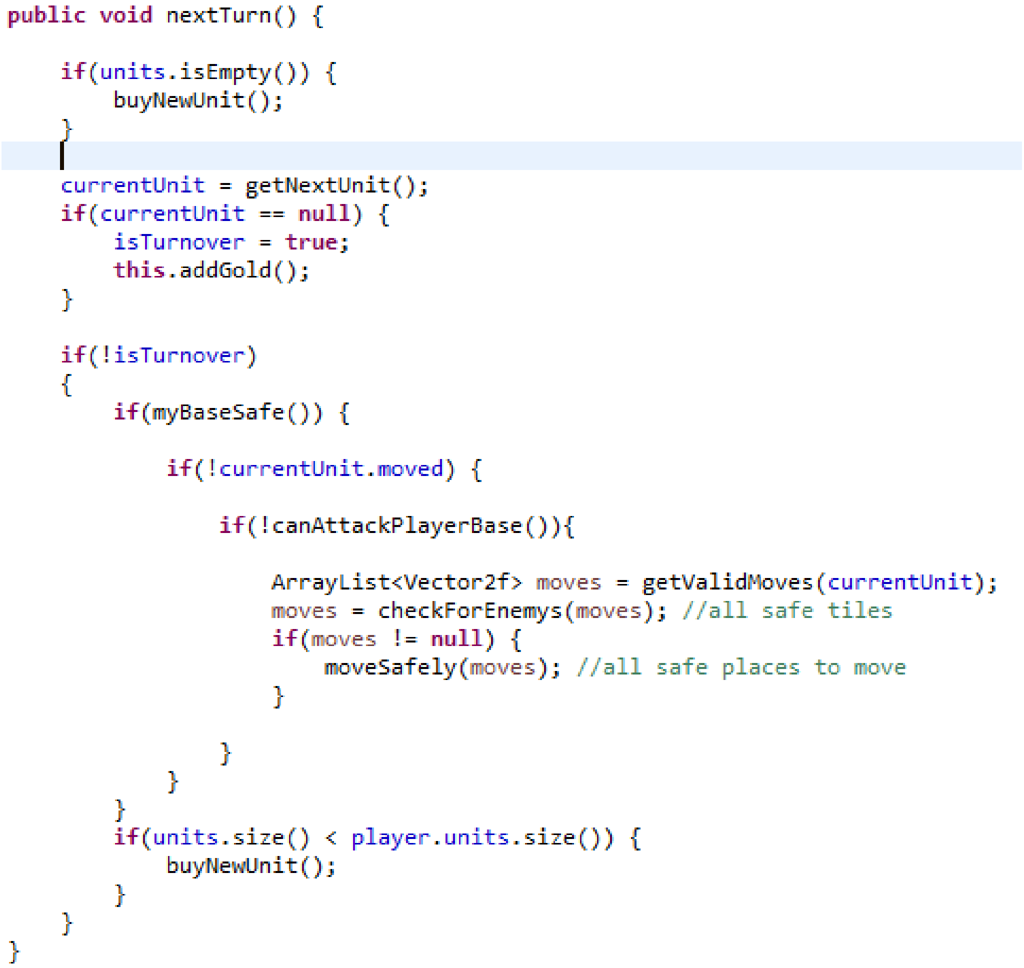


FIGURE 5 : AI behaviour tree



```
public void nextTurn() {

    if(units.isEmpty()) {
        buyNewUnit();
    }

    currentUnit = getNextUnit();
    if(currentUnit == null) {
        isTurnover = true;
        this.addGold();
    }

    if(!isTurnover)
    {
        if(myBaseSafe()) {

            if(!currentUnit.moved) {

                if(!canAttackPlayerBase()){

                    ArrayList<Vector2f> moves = getValidMoves(currentUnit);
                    moves = checkForEnemys(moves); //all safe tiles
                    if(moves != null) {
                        moveSafely(moves); //all safe places to move
                    }

                }
            }
        }
        if(units.size() < player.units.size()) {
            buyNewUnit();
        }
    }
}
```

FIGURE 6 : nextTurn extract

CHAPTER 5

RESULTS

5.1 Testing

This chapter details the testing to the system which has been run to check the functionality of the system has been implemented correctly, and see how well the system matches up to original requirements specification.

5.1.1 Testing the engine

Testing the engine happened continuously while building the product until the final version was produced. Multiple different layouts were tested using different png files to see if they map would build correctly. It was tested that if a number was returned that hasn’t been assigned to a tile already then it will just default to the lowest value which is the tile that the players can move freely on as to not disrupt the game if anything happened to the image. In addition the tile sizes were rigorously tested by changing their size to way bigger than they should every by and too small to make sure it would still render correctly which it did.

5.1.2 Testing the AI

A series of tests have been detailed below of what was tested on the AI to make sure it correctly reacts to the circumstance as predicted. The majority of testing on the AI was conducted as it was being built to make sure each component was working as intended.

Scenario	Predicted outcome	Actual outcome
AI has no units near its tower but a weak enemy is nearby	The AI should produce a new unit, depending on how much gold it currently has depends on the level it produces	The AI produced a level 3 unit and killed the level 2 unit next to base
Player unit that can kill AI unit is in range	AI should move to a safe location away from the unit as best possible	The AI moved to a safe location but trapped itself in a corner by doing so
One enemy unit in range that matches AI unit in strength	When getting list of possible moves remove this tile from the array	Tile was removed from possible moves however next to it was not which would not be a safe location
AI just started game no threat nearby	Pick a tile to move to	Always picks the top left tile because it is the first tile it deems safe
AI’s tower is below 50hp and enemy in range with no AI units nearby	Create the highest power unit with the amount of gold AI has	As predicted, but unit moved to top left again

Table 1: Testing the AI

5.1.3 Black box testing

User testing was conducted on a group of people who had not played the game before, they were told how to play and asked to complete a set of objectives.

Objective	Comments on process	Suggested improvements
Kill an enemy unit	“It’s difficult to nail down an enemy because it keeps moving away” “Had to create a 2nd unit to be able to kill one”	Only allow units to move in X and Y axis rather than all
Let an enemy kill one of your units, then kill the weakened enemy unit	“AI is easy to trap once you’ve been told this” “Need to sacrifice 2 units to kill one or else you just lose a unit”	Units need balancing
Win the game	“Fairly hard game, AI destroys unit every time you get close to attacking the tower” “When you’re trying to attack it’s tower, it’s attacking yours”	Make the game board larger so it is easier to get around the enemy’s units Units need balancing
Test using a combination of camera controls at once to test for deadlock	“Works well, but could make diagonally if holding 2 at the same time”	Camera is working suitably if the project timeline was longer this could be added

Table 2: Black box testing

5.1.4 HCI testing

Users found that the game controls were straightforward and easy to grasp, most liked that the game used only left mouse clicks to move units around because they said usually it can take some time to figure out exactly what does what. However, they were overall disappointed with the design of the product saying it was fairly visually un-appealing to look at which wouldn’t make them want to play much more.

Some commented on how they liked the lack of UI because it can make games messy, but as expected they wanted some UI to show elements such as the health of a unit and how much gold they have. This can be accessed view keyboard shortcuts but it is only printed to the output in the debug problem not the final product yet.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Reflection on process

The aim of this thesis was to explore the games development process and implement a simple game utilising the information gathered through background research, and create an AI for the user to play against.

6.2 PSEL issues

The project is low in all areas of PSEL because it is a self-contained look at game development and apart from the few people testing the product it will never be released to the public. Their information was not gathered other than their remarks about the product so their privacy is protected.

All art work used was gathered from opengameart.org which is an open source site providing people with assets to use for non-commercial products the misuse of these products would be a breach of intellectual property as the project can claim no ownership of the asserts used and should not profit off of them.

The game is designed to be an enjoyable experience and not malicious in anyway, there is no graphic imagery displayed so people of any age could potentially play the game.

6.3 Conclusions of process

6.3.1 Game design

Looking back to Chapter 3 (3.2) at the function requirements the vast majority have been correctly implemented with a few minor exceptions. Although it was only secondary objective 6.3 and 6.4: " The system should look multiple moves ahead to plan it's strategy." and "The system should plan all of its units move in conjunction with each other before moving." have not been implemented. This is due to time constraints as getting the system to the standard it is currently where it simply looks at the next move ahead took longer to implement than expected.

The system does allow for either 2 human players, or 2 AI players or one of each however, a start screen with buttons was not considered in the initial planning document and therefore was not factored into the timeline of production. When this mistake was discovered a decision had to be made as to how important it was versus other features and it the decision was made to not implement the menu. It is possible to quickly switch between the 3 modes in the code.

6.3.2 AI design

By the time the AI was being produced the game had already been created as a 2 human player game so it was tested working, this meant the AI could simply be derived from the player class and added too. Once the behaviour tree was draw out it provided obvious functions that needed to be created which made the process fairly straight forward. However the AI doesn't follow such a straightforward tree structure in the code because not every branch is a binary yes or no as several checks need to be made which would result in repeating code. Given longer the AI would be stripped down and rebuilt better using clear modular functions. As it stand the AI provides the core functionality that was set out to be created so it proves a success for the project.

6.3.3 LWJGL

Overall LWJGL turned out to be a robust and useful library to develop the game in. There was no need for any extra libraries to complete any of the project aims. The main issue was that with no experience using it or even OpenGL before there is a lack of resources helping teach anything past the basic of LWJGL so standard OpenGL was needed to be used but then figuring out the differences between the C++ version and Java did make it complex at times. To complete the aim of this project it was a very good learning experience in both Java and OpenGL however, to built a large scale product C++ would be a much more beneficial due to the large number of examples already out there to help development. Another issue with LWJGL is that a new version was recently released so several functions were now deprecated so tutorials were hard to come by.

6.3 Future ideas

The game created was a good basis for a much bigger project to be built on top of, with more time a much more refined engine could be created. Additions could include allowing the map to be created on the fly by the user adding tiles or changing existing ones to create new levels.

The AI created was fairly basic compared to what was initially set out to be created, although it does what to should most of the time it does not critically analysis the board more than 1 move ahead and plan a strategy which is necessary to properly beat someone who knows what they are doing. The AI could have been designed with less cohesive with this particular game as the aim was to create a modular game which could be used over and over. However, it would work for any game based on a grid with a bit of changing.

In any game like this there can always be improvements adding new content and making the core game a better experience for the user. Specifically if there was more time in the project online play could have been implemented, java is a very easy language to quickly add network facilities to a project. There would also be time to make the game visually appealing, adding animations and slick menus.

Given just a bit more time the final few features that didn’t get time to be implemented could have been added. This is a better GUI that displays the current users gold at all times and when they click on a unit it was show them it’s health and attack. Due to time constraints these features are printed out to the IDE but not in the finished game, the AI took longer than anticipated to implement and so the least essential features were passed over.

REFERENCES

1. SOMMERVILLE, I., 2001. Software Engineering. 6th Edition. Essex: Addison-Wesley Publishers Limited.
2. http://www.maxgames.com/play/age-of-war.html
3. https://play.google.com/store/apps/details?id=com.hammerfall.regicide&hl=en_GB

BIBLIOGRAPHY

1. SOMMERVILLE, I., 2001. Software Engineering. 6th Edition. Essex: Addison-Wesley Publishers Limited.
2. http://www.maxgames.com/play/age-of-war.html
3. https://play.google.com/store/apps/details?id=com.hammerfall.regicide&hl=en_GB
4. http://www.33rdsquare.com/2016/07/the-limitations-of-artificial.html
5. AI Game Programming Wisdom 3 - Steve Rabin 2006
6. https://www.youtube.com/channel/UCVebYXGDlnFPTIB4CT2dcGA
7. http://www.glfw.org/docs/latest/intro_guide.html
8. https://github.com/lwjglgamedev/lwjglbook
9. http://wiki.lwjgl.org/wiki/Learning_LWJGL.html
10. https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

Appendix

A.

User manual:

Action	Input
Select Unit	Left mouse click unit
Attack Unit	Left mouse click unit with unit already selected
Create Light unit	1 on keyboard
Create Medium unit	2 on keyboard
Create Heavy unit	3 on keyboard
View gold	G on keyboard
Display unit details	H on keyboard

A light unit:

Movement speed: 3

Health: 10

Attack damage: 5

Cost: 8

A medium unit:

Movement speed: 2

Health: 15

Attack damage: 10

Cost: 12

A heavy unit:

Movement speed: 2

Health: 19

Attack damage: 14

Cost: 18