DOKUZ EYLUL UNIVERSITY

ENGINEERING FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

CME 4201

Computer Engineer Design / 2020-2021

ART OF WAR

(Tactical Real Time Strategy Game with Dynamic AI)

Advisor

Assoc. Lec. Tanzer ONURGİL

By

Atakan Arda ÇELİK

Mehmet DEMİRTAŞ

Yıldırım Çağlar DAŞDEMİR

Dec 2020
İZMİR

ART OF WAR
(Tactical Real Time Strategy game with Dynamic AI)

A Thesis Submitted to the
Dokuz Eylül University, Department of Computer Engineering
In Partial Fulfillment of the Requirements for the Degree of B.Sc.

by
Atakan Arda ÇELİK
Mehmet DEMİRTAŞ
Yıldırım Çağlar DAŞDEMİR

Advisor
Assoc. Lec. Tanzer Onurgil

January, 2021
İZMİR

Chapter One

Introduction

1.1 Background Information:

For many years, humanity played games to relax or have fun. Games are a form of entertainment and a recreational activity that is played purely for fun, for competition or for education.

A new form of games emerged as electronical devices are popularized in the last century. These games are called “video games”. Starting from 1947 with “Cathode Ray Tube Amusement Device”, one of the first examples of electronic games, later popularized by the likes of “Pong, Space War etc.”, video games gained popularity fast. Today, video game industry is one of the biggest entertainment industries of the world.

Many forms of games have been developed over the years. Today, there are a selection of categories that a user can pick, one of them being “Real Time Strategy Games”. Real time strategy games appeared in 1981 with “Infoworld” and has been a popular category since. In RTS’, players generally tro to control large amount of units, battle with opponents and manage resources. With arise of RTS genre, pursuits of an intelligent opponent played by a computer began.

These researches led to the use of AI (Artificial Intelligence). Developers came up with clever AI designs that would challenge players with clever decisions and tactical capabilities. Today many RTS games include computer AI’s that players can play with, as a matter of fact this became a standard property for this genre.

Our aim is to develop a tactical RTS game and an AI from scratch that can make logical decisions and can use resources efficiently.

1.2 Problem Definition

In many RTS games, there are several difficulties that players can choose. Although these difficulty levels also affect AI, most of the time rather than implementing more complex AI mechanics, they give advantages and handicaps to computer players. For example in Age of Empires 2, when you choose a harder difficulty setting, AI players get several advantages over you such as, building faster, training units faster, collecting resources faster etc. We think that by improving AI implementations, more clever solutions can be developed.

We will try to solve this player – AI conflict by designing an AI that can analyze it's surroundings, decide what to do and calculate it's advantages.

1.3 Motivation / Related Works

As avid players of RTS genre, we like to play games like Total War, Age of Empires, Stronghold and we encounter the problems we mentioned about regularly. We believe that seeing an army you just wiped out coming back regenerated 5 minutes later is not good game design. Our biggest motivation for this project is creating a smart and fun to play with AI.

As the foundation, we took the concept of real time battles from Total War series. Total War has one of the most advanced real time AI's in todays gaming industry. It can evaluate different conditions and make complex decisions in a matter of seconds, such as; where to locate their army according to height advantage, cover advantage etc., how to position their units among each other and so on. We will try to replicate this game in our own terms with our own custom created AI.

Even the U.S Army took interest in AI, It is a known fact that various armies used RTS AI based applications to train their troops, or test their weapon efficiencies and tried to optimize their tactics to reach perfection. Therefore, we believe that this project is more likely to get investments and marketable compared to other genres.

1.4 Goal/Contribution

First of all, Art of War game will consist of wars. Thus, our thesis is that there are a number of factors that in the Art of War game is to constitute Cavalry, Infantry and Archer units. It’s clear to obvious that, these issues will be given exclusive attention as the struggle mechanisms and war tactics of the Cavalry, Infantry and Archer units are very important. It is aimed for these Cavalry, Infantry and Archer units to progress battle in the form of regiments. After that, there will be additions such as these units moving and battling as a regiment.

As attachment, factors such as different maps and weather conditions in the Art of War game will affect the war situation. In the Artificial Intelligence(AI) section, which we are going to focus more on in the 2nd term of the school, coding will be made about events such as the superiority of military units in war and war conditions. To sum up, as an Art of War game, it is our primary goal to make an effective and stunning game in the Real-time Strategy(RTS) world.

1.5 Project Scope

The implementation of out project consist of several mechanics,

- Unit movements,
- Several unit formations,
- Combat mechanics of different classes such as infantry, archer etc.
- Environmental effects on unit movements, combat and tactics altogether.

In light of these desired mechanics listed above, there are several milestones to hit before successfully completing the project,

- Preparing the preferable game engine and libraries,
- Creating movement scripts, formation and combat scripts with placeholder models.
- Implementing pathfinding algorithms.
- Building first prototype without AI implementations.

- Scripting a basic AI component to track units or flee according to their power advantage or disadvantage against them.
- Crafting models and playable terrains.
- Building a playable version with models and several demo maps.
- Implementing core AI elements within playable version.

1.6 Methodology/Tools/Libraries

The methodology we follow mostly resembles the characteristics of Extreme Programming, the requirements and releases changes frequently and swiftly. We use Unity3D Engine as the base of our game. Unity3D is a partially free, advanced game engine widely used in game development community. It has GameObject tools which works around components like Rigidbody, colliders, Renderers. Which means it provides all the primary tools for developers to tweak and build their games around. It supports C# scripts as its main language, there are versions where developers can use C++, but we decided to use C# since it provides better documentation and better implementation with Unity3D engine.

CHAPTER 2

2. Literature Review

In Real-Time Simulation games, in particular Pathfinding usage is important and necessary, for instance Age of Empires. A* is known as one of the most popular pathfinding algorithms, but on the other hand, its big problem is CPU time and memory consumption, hard to satisfy in a complex, dynamic, realtime environment with large numbers of units. Even if specialized algorithms, such as D*-Lite exist, it is most common to use A* combined with a map simplification technique that generates a simpler navigation graph to be used for pathfinding. An example of such technique is Triangulation Reduction A*, that computes polygonal triangulations on a grid-based map. Considering movement for groups of units, rather then individual units, techniques such as steering of flocking behaviors can be used on top of a path-finding algorithm in order to make whole groups of units follow a given path. In recent commercial RTS games like StarCraft 2 or Supreme Commander 2, flocking-like behaviors are inspired of continuum crowds ("flow field"). A comprehensive review about (grid-based) pathfinding was recently done by Sturtevant. *(Ontanon, S., Uriarte, A., Synnaeve, G., & Richoux, F., 2015).*

One of the best papers about AI and RTS fields is "RTS Games and Real–Time AI Research" by Michael Buro and Timothy M. Furtak *(Ontanon, S., Uriarte, A., Synnaeve, G., & Richoux, F., 2015).* This research is a general outlook of RTS title and its AI implementation process. Commercial computer games are a growing part of the entertainment industry and simulations are a critical aspect of modern military training. These two fields have much in common, cross–fertilize, and are driving real–time AI research.

RTS games offer a large variety of fundamental AI research problems, unlike other game genres studied by the AI community so far:

- **Adversarial real–time planning.** In fine–grained realistic simulations, agents cannot afford to think in terms of micro actions such as “move one step North”. Instead, abstractions of the world state have to be found that allow AI programs to conduct forward searches in a manageable abstract space and to translate found solutions back into action sequences in the original state space. Because the environment is also dynamic, hostile, and smart — adver-sarial real–time planning approaches need to be investigated.

- **Collaboration.** In RTS games groups of players can join forces and intelligence. How to coordinate actions effectively by communication among the parties is a challenging research problem. For instance, in case of mixed human/AI teams, the AI player often behaves awkwardly because it does not monitor the human’s actions, cannot infer the human’s intentions, and fails to synchronize attacks.

- **Pathfinding**. Finding high–quality paths quickly in 2D terrains is of great importance in RTS games. In the past, only a small fraction of the CPU time could be devoted to AI tasks, of which finding shortest paths was the most time consuming. Hardware graphics accelerators are now allowing programs to spend more time on AI tasks. Still, the presence of hundreds of moving objects and the urge for more realistic simulations in RTS games make it necessary to improve and generalize pathfinding algorithms. Keeping unit formations and taking terrain properties, minimal turn radii, inertia, enemy influence, and fuel consumption into account greatly complicates the once simple problem of finding shortest paths.

- **Decision making under uncertainty.** Initially, players are not aware of the enemies’ base locations and intentions. It is necessary to gather intelligence by sending out scouts and to draw conclusions to adapt. If no data about opponent locations and actions is available yet, plausible hypotheses have to be formed and acted upon.

• Opponent modeling, learning. One of the biggest shortcomings of current (RTS) game AI systems is their inability to learn quickly. Human players only need a couple of games to spot opponents’ weaknesses and to exploit them in future games. New

efficient machine learning techniques have to be developed to tackle these important problems.

• Spatial and temporal reasoning. Static and dynamic terrain analysis as well as understanding temporal relations of actions is of utmost importance in RTS games — and yet, current game AI programs largely ignore these issues and fall victim to simple common–sense reasoning.

• Resource management. Players start the game by gathering local resources to build up defenses and attack forces to upgrade weaponry, and to climb up the technology tree.

At any given time the players have to balance the resources they spend in each category. For instance, a player who chooses to invest too many resources into upgrades, will become prone to attacks because of an insufficient number of units. Proper resource management is therefore a vital part of any successful strategy. *(Buro, M., & Furtak, T. M., 2004).*

Artificial Intelligence techniques have been successfully applied to several computer games. However in some kinds of computer games, like real-time strategy (RTS) games, traditional artificial intelligence techniques fail to play at a human level because of the vast search spaces that they entail. In this paper we present a real-time case based planning and execution approach designed to deal with RTS games. We propose to extract behavioral knowledge from expert demonstrations inform of individual cases. This knowledge can be reused via a case based behavior generator that proposes behaviors to achieve the specific open goals in the current plan. Specifically, we applied our technique to the WARGUS domain with promising results.

Developing the AI behavior for an automated agent that plays a RTS is not an easy task, and requires a large coding and debugging effort. Using the architecture presented

in this paper the game developers will be able to specify the AI behavior just by demonstration; i.e. instead of having to code the behavior using a programming language, the behavior can be specified simply by demonstrating it to the system. If the system shows an incorrect behavior in any particular situation, instead of having to find the bug in the program and fix it, the game developers can simply demonstrate the correct action in the particular situation.

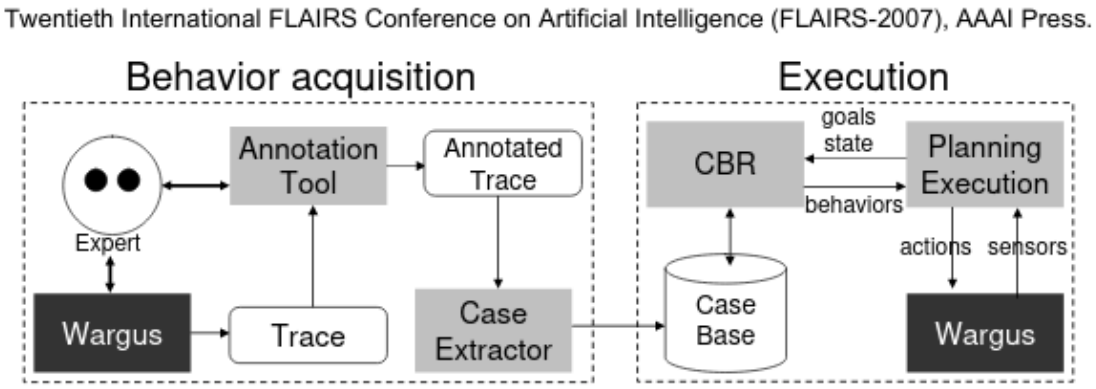

Figure 2.1: Case Based behavior acquisition and planning architecture.

Case-Based Behavior Learning in Wargus

Figure 2.1 shows an overview of the process used to learn behaviors from expert demonstrations. The process is divided into two stages:

• **Behavior acquisition**: Where a set of cases are extracted from an expert trace.

• **Execution:** Where the cases extracted are reused to play the game.

The first step in the process involves an expert providing a demonstration to the system of how to play the game. As a result of that demonstration, the system obtains a game trace consisting of the set of actions executed during the game. The next step is to

annotate the trace. For this process, the expert uses a simple annotation tool that allows him to specify which goals was he pursuing with each particular action. Next, as Figure 1 shows, the annotated trace is processed by the case extractor module, that encodes the strategy of the expert in this particular trace in a series of cases. A case stores a sequence of actions that an expert used in a particular situation to achieve a particular goal. Notice that from a single trace many cases can be extracted. For instance, if the expert destroyed multiple towers of the enemy, we can collect multiple cases on how to destroy a tower.

Once the cases have been extracted from the expert demonstration, the system is ready to use them in actual game play. During performance, the interleaved planning and execution (PE) module keeps track of the open goals. For each open goal, the PE module sends the goal and the current game state to the CBR module. In response, the CBR module selects a case from the case base that suits the goal and game state. That case is adapted to match the current game state (adjusting which units make which actions, and on what coordinates, since in the new map the stored coordinates in the case might not make sense). Once adaptation has taken place, the adapted behavior is sent to the PE module. The PE module keeps track of the behaviors that are being executed for each goal, and ensures that each sub-goal of each behavior is also satisfied by maintaining an execution tree of goals and sub-goals.

In addition, by using the alive conditions of the behaviors, it monitors whether a particular behavior is still alive, and whether it is worthwhile to continue executing it. Each time a behavior is finished (or canceled), the PE module checks whether the goal that behavior was pursuing has been achieved. If it has not been achieved, then another behavior must be created by the CBR module to satisfy the goal. Future Work We plan to incorporate learning during performance by retaining those adapted behaviors that succeeded when applied. This will enable the system to learn with experience. The goal is to use the behaviors extracted from the expert as the starting point, and slowly improve the behavior library with experience. At any time, if the game developers see that the system is unable to succeed in a particular situation, an expert demonstration for that situation can be provided. *(Ram, A., Ontanon, S., & Mehta, M., 2007).*

“Artificial Intelligence for Adaptive Computer Games” presents its ideas with a solid understanding of AI and its implementation to RTS titles. It emphasizes on “Case Based Learning” and solidifies its ideas with a well-built figure. The idea is built around AI’s learning capabilities, cases and phases included into process.

We developed knowledge-rich agents to play real-time strategy games by interfacing the ORTS game engine to the Soar cognitive architecture.The middleware we developed supports grouping, attention, coordinated path finding, and FSM control of low-level unit behaviors. The middleware attempts to provide information humans use to reason about RTS games, andfacilitates creating agent behaviors in Soar. Agents implemented with this system won two out of three categories in the AIIDE 2006 ORTS competition.

Another useful work on RTS game development is, “Implementing Coordinated Movement” by Dave Pottinger.

Group Movement

Looking at the definition of a group (see sidebar at right), we can immediately see that we need to store several pieces of data. We need a list of the units that make up our group, and we need the maximum speed at which the group can move while still keeping together. Additionally, we probably want to store the centroid of the group, which will give us a handy reference point for the group. We also want to store a commander for the group. For most games, it doesn't matter how the commander is selected; it's just important to have one.

One basic question needs to be answered before we proceed, though. Do we need to keep the units together as they move across the board? If not, then the group is just a user interface convenience. Each unit will path and move as if the player had issued individual commands to each group member. As we look at how to improve on the organization of our groups, we can see that there are varying degrees of group movement cohesion.

Units in a group just move at the same speed. Usually, this sort of organization moves the group at the maximum speed of its slowest unit, but sometimes it's better to let a slow unit move a little faster when it's in a group (see Figure 2.2 below). Designers generally give units a slow movement speed for a reason, though; altering that speed can often create unbalanced game play by allowing a powerful unit to move around the map too quickly.

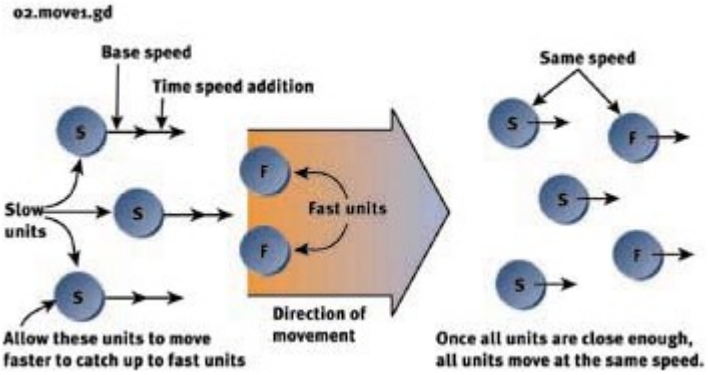

Figure 2.2. Unit movement

Units in a group move at the same speed and take the same path. "This sort of organization prevents half of the group's units from walking one way around the forest while the other half takes a completely different route (see Figure 2.3 below). Later, we'll look at an easy way to implement this sort of organization.

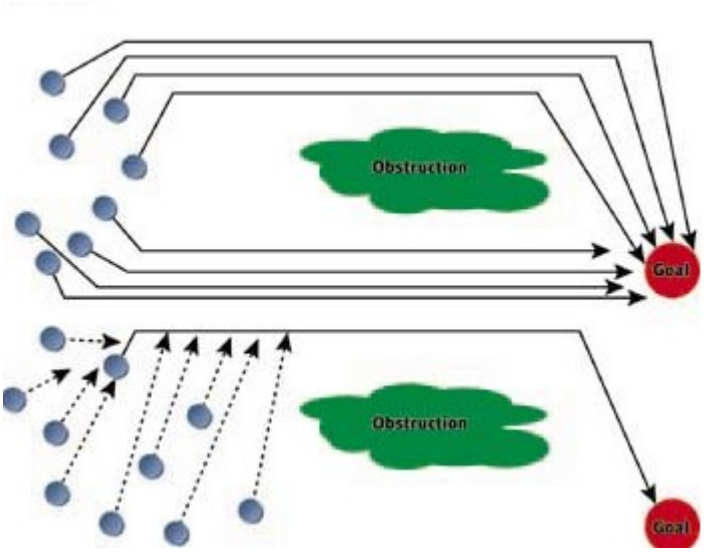

Figure 2.3. Unit movement with obstruction

Units in a group move at the same speed, take the same path, and arrive at the same time. This organization exhibits the most complex behavior that we'll apply to our group definition. In addition to combining the previous two options, it also requires that units farther ahead wait for other units to catch up and possibly allows slower units to get a temporary speed boost in order to catch up. (*Pottinger, D., 1999, January 20)*

This research is a great fundamental source for implementing unit movement and pathfinding.

Finally, implementing combat mechanics to an RTS title is discussed in "*Fast Heuristic Search for RTS Game Combat Scenarios*". This research focuses on some of the common scenarios and defines solution and new implementations for these scenarios.

Battle unit management (also called micromanagement) is a core skill of successful human RTS game players and is vital to playing at a professional level. One of the top STAR CRAFT players of all time, Jaedong, who is well known for his excellent unit control, said in a recent interview: "That micro made me different from everyone else in Brood War, and I won a lot of games on that micro alone". It has also been proved to be decisive in the previous STARCRAFT AI competitions, with many battles between the top three AI agents being won or lost due to the quality of unit control. In this paper we focus on small-scale battle we call combat, in which a small number of units interact in a small map region without obstructions.

In order to perform search for combat scenarios in STARCRAFT, we must construct a system which allows us to efficiently simulate the game itself. The BWAPI programming interface allows for interaction with the STARCRAFT interface, but unfortunately, it can only run the engine at 32 times"normal speed" and does not allow us to create and manipulate local state instances efficiently. As one search may simulate millions of moves, with each move having a duration of at least one simulation frame, it remains for us to construct an abstract model of STARCRAFT combat which is able to efficiently implement moves in a way that does not rely on simulating each in-game frame. *(Churchill, D., Saffidine, A., & Buro, M., 2012)*

CHAPTER 3

3.1 Functional Requirements

- Player is able to display menu and choose the desired option.
- System allows player to set configurations through settings window.
- System allows player to read instructions.
- System allows player to exit game.
- System allows player to create the game session.
- Player can move camera accross the map.
- Player can create units, move units.
- Player can change formation of his/her units.
- Player can attack enemy units.
- Player can inflict/take damage.
- AI can attack/retreat or change formation of its units.
- AI can inflict/take damage.

3.2 Non-Functional Requirements

- **Performance:** Our game design has to be well optimized in terms of performance, a wide range of hardware and operating systems should be able to run “Art of War”.
- **Reliability:** It has to be technically solid, user should be able to execute functions without any vulnerabilities.
- **Scalability:** It should be able to support scale changes so “Art of War” can be a game where expensions or design changes would be no problem for development team.
- **Usability:** “Art of War” has to offer solid and alternative ways of usage and.
- **Maintainability:** Maintain phase is a critical part of development flow, so it has to support high maintainability.

CHAPTER 4

4.1 ER DIAGRAM:

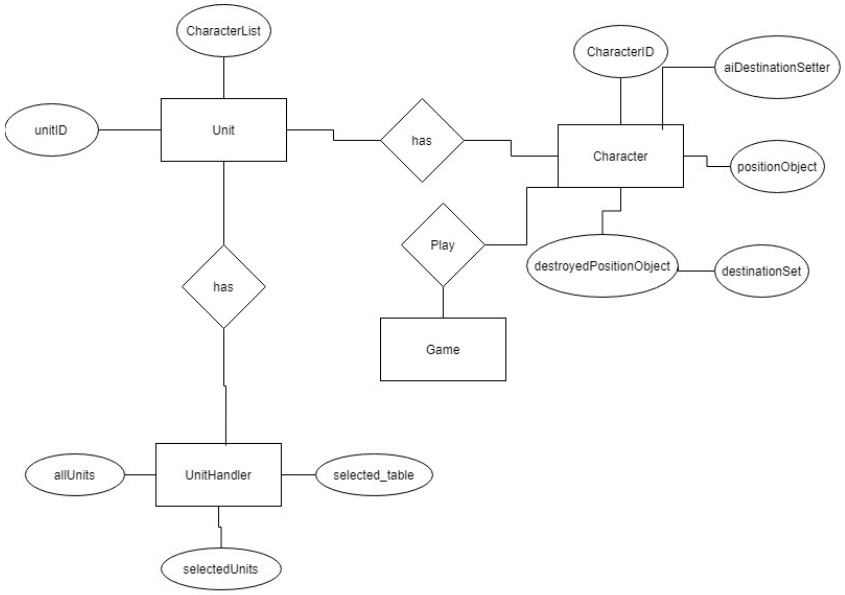

Figure 4.1. ER Diagram for units

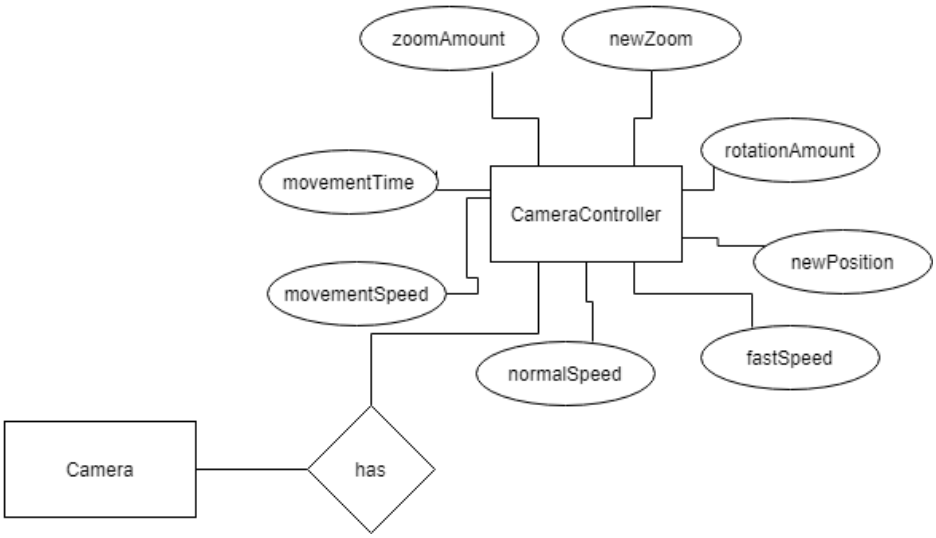

Figure 4.2. ER Diagram for camera asset

In these figures we can see entity relations for the components “CameraController” and “Unit”. All these relations are classified by specified functions defined in scripts of these entities.

4.2 Class Diagram:

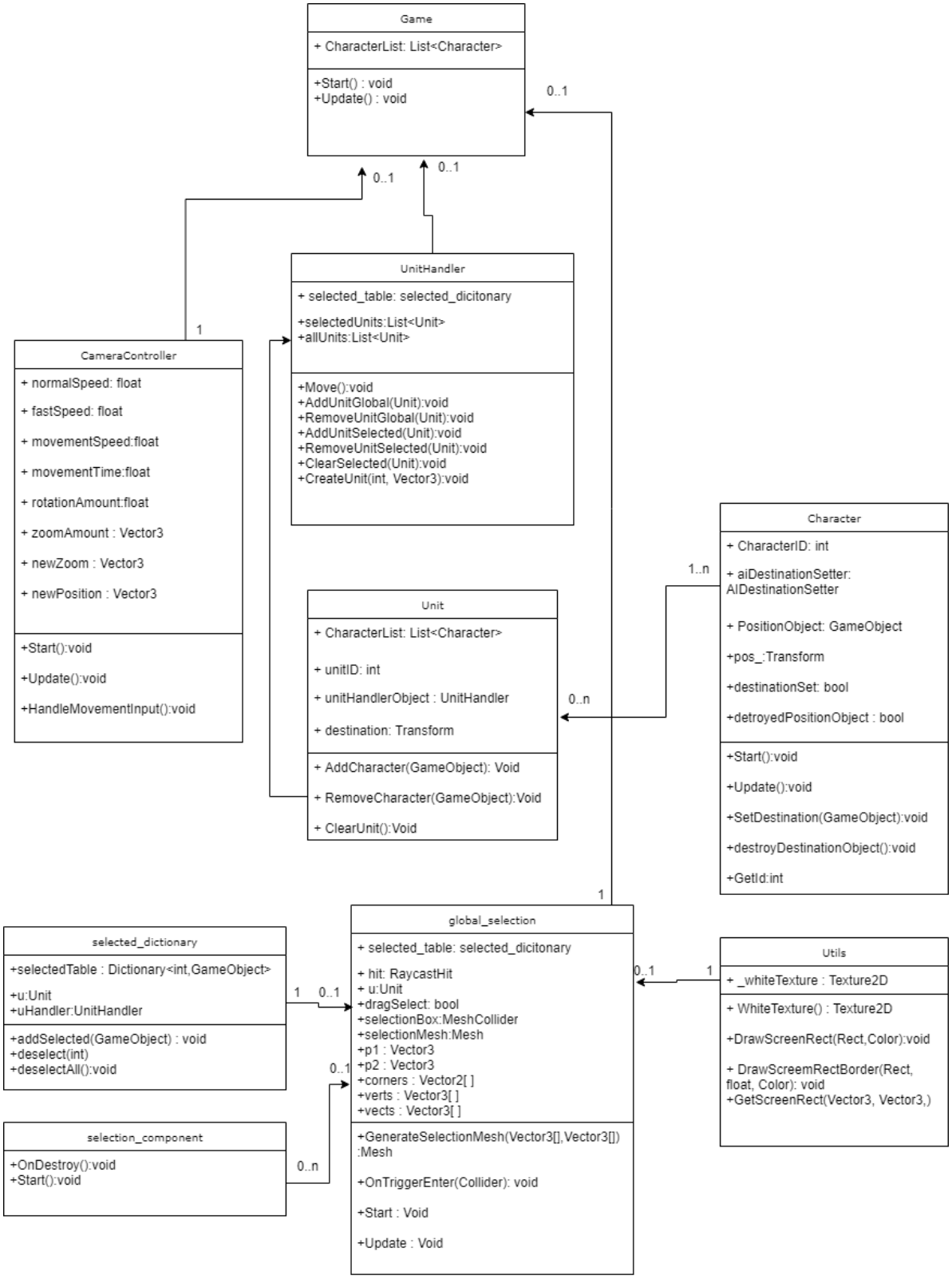

Figure 4.3. Class Diagram for the project

4.3 UI Design

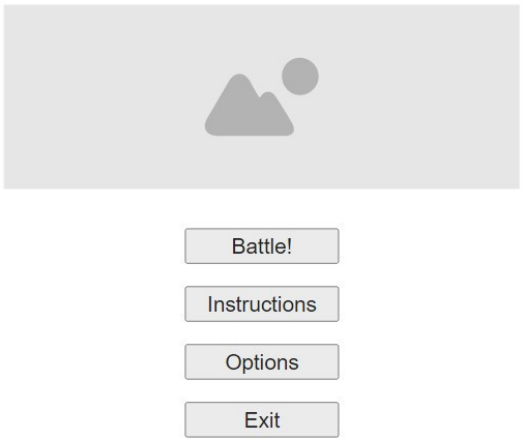

Figure 4.4. UI mockup for main menu

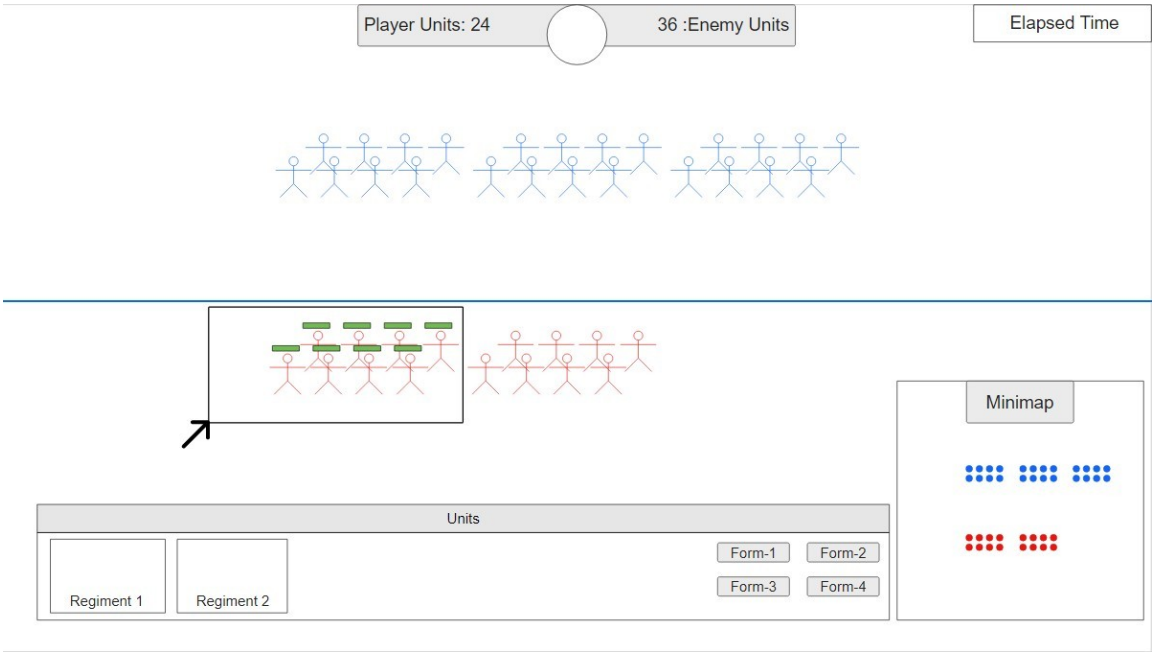

Figure 4.5. UI mockup for in-game screen

Here we can see two mockup designs for “Main menu screen” and “In-game screen”. Final UI of the project will look like this, minding readability and easy-access functions.

4.4 Use Cases

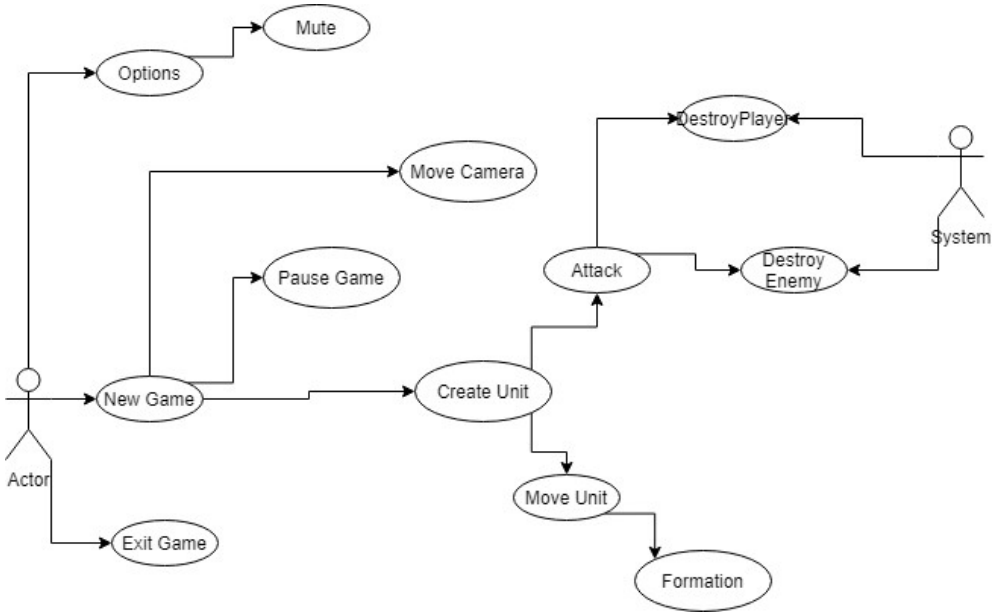

Figure 4.6. Use case diagram for both player and system

In this figure we can see all use cases for player, both in-game and menu included. These use cases are fundemental for the game and may include different use cases in the future.

4.5 Sequence Diagram

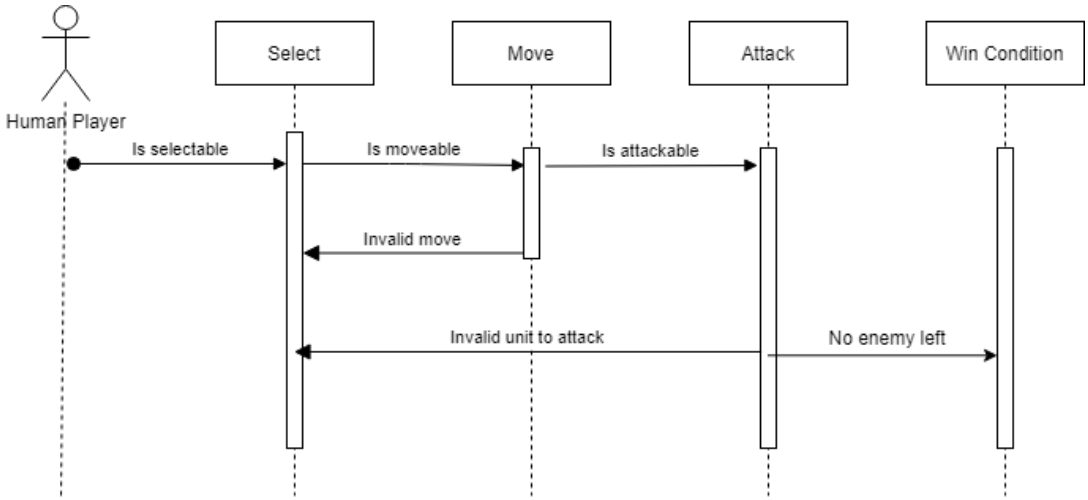

Figure 4.7. Sequence diagram for the gameloop of player

This sequence diagram summarizes the main game loop for the player.

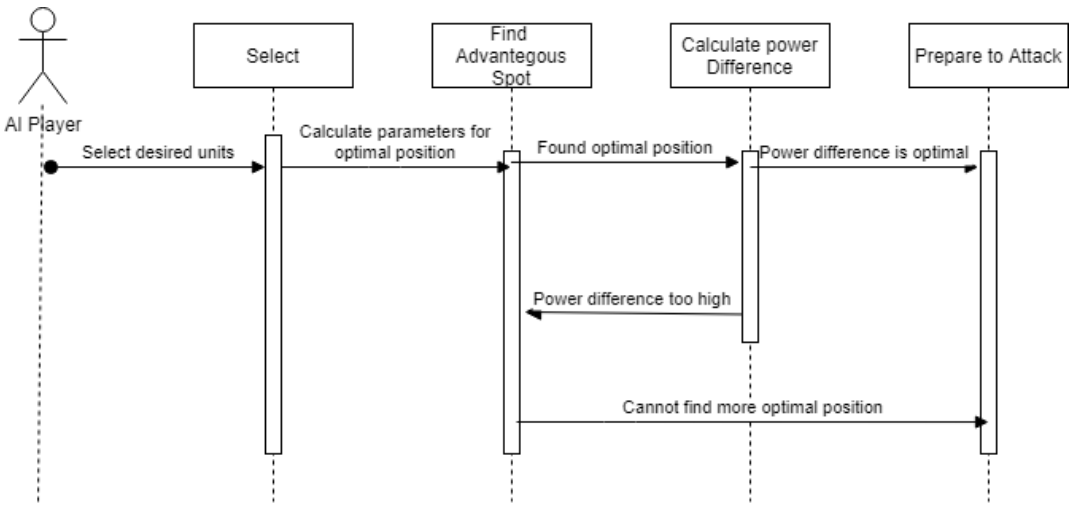

Figure 4.8. Sequence diagram for the gameloop of AI

This sequence diagram summarizes the main game loop of AI.

4.6 Activity Diagram

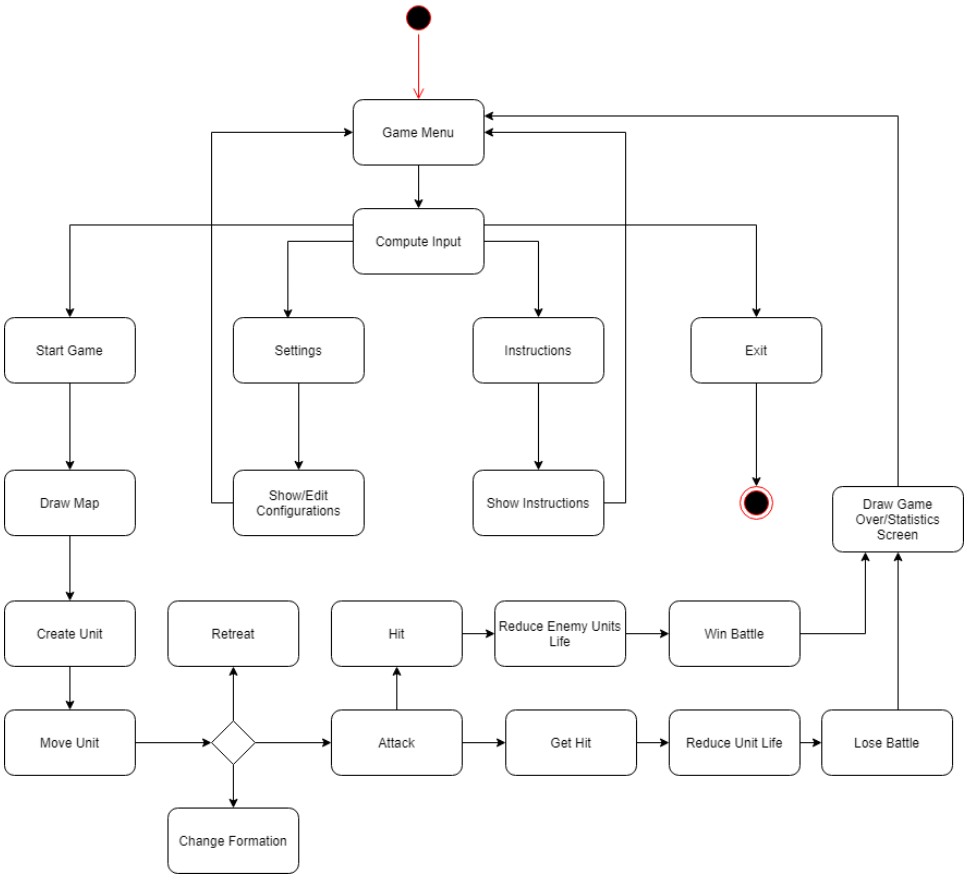

Figure 4.9. Activity diagram of the player

CHAPTER 5

5.0 Camera Controller

Camera renders the scene and objects. A camera controller has been implemented by our team.

```
public Transform cameraTransform;

public float normalSpeed;
public float fastSpeed;
public float movementSpeed;
public float movementTime;
public float rotationAmount;

public Vector3 zoomAmount;
public Vector3 newZoom;
public Vector3 newPosition;
public Quaternion newRotation;
// Start is called before the first frame update
void Start()
{
    newPosition = transform.position;
    newRotation = transform.rotation;
    newZoom = cameraTransform.localPosition;
}
```

Image 5.1: CameraController variables and instantiations

newPosition, newRotation and newZoom variables are instantiated as the camera object is created by engine.

```
void HandleMovementInput()
{
    //Camera speed while holding "SHIFT"
    if(Input.GetKey(KeyCode.LeftShift)){
        movementSpeed = fastSpeed;
    }
    else{
        movementSpeed = normalSpeed;
    }

    //Camera movement
    if(Input.GetKey(KeyCode.W) || (Input.GetKey(KeyCode.UpArrow))){
        newPosition += (transform.forward * movementSpeed);
    }
    if(Input.GetKey(KeyCode.S) || (Input.GetKey(KeyCode.DownArrow))){
        newPosition += (transform.forward * -movementSpeed);
    }
    if(Input.GetKey(KeyCode.D) || (Input.GetKey(KeyCode.RightArrow))){
        newPosition += (transform.right * movementSpeed);
    }
    if(Input.GetKey(KeyCode.A) || (Input.GetKey(KeyCode.LeftArrow))){
        newPosition += (transform.right * -movementSpeed);
    }
```

Image 5.2: Handle Movement Function

HandleMovementInput() is the function where the script takes the inputs such as w,a,s,d or arrow buttons and applies the movement vectors to camera object. The script also have conditions for “Q,E” for rotation and “SHIFT” for accelerating the camera.

5.1 Unit Selection with Mouse

To create a fully dynamic units, first there has to be a system where user can select units and materials by clicking or dragging accross the screen to select multiple. Several scripts including a dictionary script called “selection dictionary” to map the selected objects, a “selection component” script to change the material of selected objects and a “global selection script” to implement main functions like selecting the units with clicking or dragging, has been implemented by our team.

```
//3. when mouse button comes up
if (Input.GetMouseButtonUp(0))
{
    if(dragSelect == false) //single select
    {
        Ray ray = Camera.main.ScreenPointToRay(p1);

        if(Physics.Raycast(ray,out hit, 50000.0f))
        {
            if (Input.GetKey(KeyCode.LeftShift)&&hit.transform.gameObject.CompareTag("Unit")) //inclusive
            {

                selected_table.addSelected(hit.transform.gameObject);
            }
            else if(hit.transform.gameObject.CompareTag("Unit")) //exclusive selected
            {
                selected_table.deselectAll();
                selected_table.addSelected(hit.transform.gameObject);
            }
        }
        else //if we didnt hit something
        {
```

Image 5.3: Global Selection Script

Ray object shoots a ray by referencing the camera and if this ray hits an object that means user selected a unit. And if the object is selected using SHIFT key, script adds that object into the list, if not script deselects them all and only selects the last clicked object.

5.2 Movement and Pathfinding

The movement is one of the key elements when it comes to creating an RTS, and there are some rules that comes with it. First, units has to find the closest path to destination, and it has to find it while it avoids the obstacles. So, an industry standard pathfinding solution “A* Pathfinding Project” by Aron Granberg has been implemented using some of the primitive functions like move, avoid and scan. First, the algorithm scans the terrain an excludes the obstacles from “walkable area”.

```
public void SetDestination(GameObject go){
    positionObject = go;
    Transform pos = positionObject.transform;
    pos_=pos;
    aiDestinationSetter.target=pos_;
    destinationSet = true;
    destroyedPositionObject = false;
}
```

Image 5.4: Set destination

Set destination function takes a GameObject object as parameter and assigns this object to aiDestinationSetter objects target field. Then the target field applies the movement vector to the attached unit.

5.3 Unit Formations

With implementing the movement of multiple units, there comes a problem of giving them a proper formation. Several formations such as horizontal, vertical and cross formations have been implemented. First, a leader unit is assigned and given the original position, remaining units are located accordingly.

```
public int compareVectorsDiagonality(Vector3 playerP, Vector3 clickedP)
{
    float tolerance = 8f;
    float dx = playerP.x - clickedP.x;
    float dz = playerP.z - clickedP.z;
    if (Mathf.Abs(dx) > tolerance && Mathf.Abs(dz) > tolerance)//çapraz
    {
        if ((dx<0&&dz<0)||(dx>0&&dz>0)) { //sol alt sağ üst
            return 1;
        }
        else if((dx > 0 && dz < 0) || (dx < 0 && dz > 0)) {//sağ alt sol üst
            return 2;
        }
    }
    else if (Mathf.Abs(dx) > tolerance && Mathf.Abs(dz) < tolerance) {//aşağı yukarı
        return 3;
    }
    else if (Mathf.Abs(dx) < tolerance && Mathf.Abs(dz) > tolerance) {//
        return 4;
    }
    return 0;
}
```

Image 5.5: Vector diagonality function

Compares the given destination vector to the position of leader unit, and returns an integer representing the positions.

```
for (int i=1; i<selectedUnits.Count;i++) {
    radius = selectedUnits[i].GetComponent<Unit>().getWidth()*1.50f;
    if (compareVectorsDiagonality(destination,selectedUnits[0].GetComponent<Unit>().transform.posit
        if (i % 2 == 0)
        {
            x = leaderX - (i * radius);
            y = leaderY + (i* radius);
        }
        else
        {
            x = leaderX + (i * radius);
            y = leaderY - (i * radius);
        }
    }
    else if(compareVectorsDiagonality(destination, selectedUnits[0].GetComponent<Unit>().transform.
        if (i%2==0) {
            x = leaderX + (i*radius);
            y = leaderY + (i * radius);
        }
        else {
            x = leaderX - (i * radius);
            y = leaderY - (i * radius);
        }
    }
```

Image 5.6: Assign x and y positions of unit

5.4 Melee Combat

Melee units has the shortest range for attacking, and they inflict the most damage to ranged units, and takes the most damage from cavalries. They're the most basic attacking unit of our RTS title.

5.5 Ranged Combat

Ranged units have the widest range for attacking and they shoot projectiles(arrows) , they inflict the most damage to cavalries and they get the most damage from infantries.

5.6 Cavalry Combat

They're the mounted and fastest units of our title. They inflict the most damage to infantries and get the most damage from ranged units.

```
if (canAttack)
{
    timeLeft -= Time.deltaTime;
    if (timeLeft < 0)
    {
        for (int i = 0; i < target.Length; i++)
        {
            if (target[i].gameObject.CompareTag("Enemy"))
            {

                if (target[i].gameObject.GetComponent<Enemy>().type == 0)
                {//infantry
                    Debug.Log("hit infantry");
                    target[i].gameObject.GetComponent<Enemy>().InflictDamage(5);
                    break;
                }
                else if (target[i].gameObject.GetComponent<Enemy>().type == 1)
                {//ranged
                    target[i].gameObject.GetComponent<Enemy>().InflictDamage(8);
                    break;
                }
                else if (target[i].gameObject.GetComponent<Enemy>().type == 2)
                {//cavalry
                    Debug.Log("hit cavalry");
                    target[i].gameObject.GetComponent<Enemy>().InflictDamage(2);
                    break;
```

Image 5.7: Combat Logic

All types of units mentioned above uses the same logic for attacking, if it collides with the sphere with given range, gets the type of collided object and inflicts damage according to type.

REFERENCES

- Buro, M., & Furtak, T. M. (2004). RTS Games and Real–Time AI Research. Skatgame.Net. https://skatgame.net/mburo/ps/BRIMS-04.pdf

- Churchill, D., Saffidine, A., & Buro, M. (2012). Fast heuristic search for RTS game combat scenarios. Researchgate.Net. https://www.researchgate.net/publication/264890517_Fast_Heuristic_Search_for_RTS_Game_Combat_Scenarios

- Pottinger, D. (1999, January 20). Gamasutra - Implementing Coordinated Movement. Gamasutra. https://www.gamasutra.com/view/feature/3314/coordinated_unit_movement.php?print=1

- Ram, A., Ontanon, S., & Mehta, M. (2007). Artificial intelligence for adaptive computer games. Researchgate.Net. https://www.researchgate.net/publication/221439041_Artificial_Intelligence_for_Adaptive_Computer_Games

- Ontanon, S., Uriarte, A., Synnaeve, G., & Richoux, F. (2015). RTS AI problems and techniques. Researchgate.Net. https://www.researchgate.net/publication/311176051_RTS_AI_Problems_and_techniques