



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

SIMULATION BASED MATCHMAKING OPTIMISATION

OPTIMALIZACE MATCHMAKINGU NA ZÁKLADĚ SIMULACE

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR

AUTOR PRÁCE

IVAN EŠTVAN

SUPERVISOR

VEDOUCÍ PRÁCE

Ing. MILET TOMÁŠ,

BRNO 2019

Zadání bakalářské práce



22016

Student: **Eštvan Ivan**
Program: Informační technologie
Název: **Optimalizace matchmakingu na základě simulace**
Simulation Based Matchmaking Optimisation
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte Unreal Engine 4, a možnosti simulace týmové hry.
2. Navrhněte a implementujte jednoduchý simulátor týmové akční hry s možností ovlivňování schopností botů a sběru statistik k výkonům jednotlivých botů.
3. Implementujte matchmaking systém, který na základě statistiky dokáže vyvážit týmy.
4. Zhodnoťte, proměřte, navrhněte rozšíření, vytvořte demonstrační video.

Literatura:

- Dle pokynů vedoucího

Podrobné závazné pokyny pro vypracování práce viz <http://www.fit.vutbr.cz/info/szz/>

Vedoucí práce: **Milet Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstract

This bachelor's thesis focuses on designing a working matchmaking system and simulation environment for a First Person Shooter like game and their implementation within Unreal Engine 4. It introduces various types of matchmaking systems used in today's games and explains some basic concepts used in Unreal Engine 4 to implement such environments. Implemented system then takes the input data, with information about players, creates matches by using our own matchmaking and performs a simulation of them, providing the simulation results of created matches for further analysis.

Abstrakt

Táto práca sa sústreďí na návrh fungujúceho matchmaking systému a prostredia pre simuláciu hier typu "First Person Shooter", v preklade "strelba z pohľadu prvej osoby," a ich implementáciu pomocou herného enginu Unreal Engine 4. Obsah práce zahŕňa predstavenie rôznych matchmaking systémov, ktoré sa používajú v dnešnom hernom priemysle a ďalej vysvetľuje základné koncepty Unreal Engine 4 použité na implementáciu tohoto prostredia. Implementovaný systém následne na základe vstupných dát a informácií o hráčovi vytvára zápasy pomocou vlastného matchmakingu, simuluje ich a zapisuje výsledky zápasov, ktoré sú použité pre ich ďalšiu analýzu.

Keywords

artificial intelligence, unreal engine 4, matchmaking, c++, blueprint, skills, team death-match, simulation, balance

Klíčová slova

umelá inteligencia, unreal engine 4, herný matchmaking, c++, blueprint, schopnosti, tímová hra, simulácia, vyváženie

Reference

EŠTVAN, Ivan. *Simulation Based Matchmaking Optimisation*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Milet Tomáš,

Rozšířený abstrakt

V tejto bakalárskej práci sa venujem problémom návrhu a implementácie funkčného simulačného prostredia pre matchmaking systém použitý v hernom žánre strieľačky z pohľadu prvej osoby. Cieľom práce je kompletný návrh jednoduchého simulačného prostredia a jeho následnej implementácie v prostredí enginu pre vývoj hier, Unreal Engine 4, a potom použitie tohto prostredia pre otestovanie mnou navrhnutého a implementovaného matchmaking systému.

V prvej časti práce sa venujem krátkemu vysvetleniu základných konceptov používaných pri implementácii hier v Unreal Engine 4, hlavne prepojeniu písania kódu v C++ s Blueprintami vo vnútri enginu. Následne objasňujem potrebné informácie pre porozumenie zvyšku práce ako vysvetlenie toho, čo je to strieľačka z pohľadu prvej osoby, aký herný mód budem implementovať – tímový zápas na život a na smrť a to, aké schopnosti hráčov majú najčastejšie vplyv na výsledky takýchto zápasov.

V nasledujúcej časti práce zachádzam viac do detailov o troch rôznych prístupoch aplikovaných na herný matchmaking a to hodnotiaci systém Elo, Glicko alebo systémy založené na hodnotení schopností hráča namiesto hodnotenia len jeho výsledkov. Tak isto objasňujem ako je vnímané označenie AI v hernom svete, jej využitie v hrách a spôsob modelovania hráča pre simulácie v herných svetoch.

Po objasnení všetkých potrebných informácií sa venujem návrhu prvkov potrebných pre túto prácu. Základom simulácie bude náš vlastný herný level, pri ktorého návrhu popisujem využitie stredovej súmernosti na jeho vytvorenie a celkovú logiku za finálnym návrhom mapy použitej v našej simulácii. Ďalej vysvetľujem ako bude vyzerat náš AI bot a hlavne objasňujem jeho štyri schopnosti. Tieto schopnosti sú presnosť, reakčný čas, schopnosť efektívneho pohybu a povedomie o okolí. Pre každú z nich definujem ako by mala fungovať vo finálnom produkte a jej efekt na priebeh simulácie. Pri vysvetľovaní návrhu AI bota sa venujem aj jeho hlavnému správaniu, ktoré je definované pomocou stromu správania. Hlavné úlohy, ktoré riadia nášho bota sú hľadanie úkrytu, sledovanie cieľa, strieľanie na cieľ, kontrola okolia a hľadanie cieľa. V každej jednej úlohe objasňujem aký vplyv na ňu majú predtým popísané schopnosti. Nakoniec sa v tejto časti venujem návrhu nášho matchmaking systému. Jedná sa o jednoduchý systém využívajúci hodnotenie schopnosti hráča pri tvorbe tímov s piatimi hráčmi. Vyváženie týchto tímov je založené na vyvažovaní jeho jednotlivých hráčov a nie celých tímov. Funkcia použitá na získanie hodnotenia hráča obsahuje štyri koeficienty, pomocou ktorých môžeme meniť váhu daných schopností hráča.

Celý návrh je implementovaný za použitia enginu Unreal Engine 4. Základná implementácia tried je v C++. Triedy sú následne rozširované za použitia Blueprintov už vo vnútri Unreal Engine 4 editoru. Navigácia po nami vytvorenom leveli je zaistená použitím navigačného mashu, ktorý je jedným z prvkov UE4. Pre implementáciu zmyslov našich AI botov je použitý tzv. AI Perception, s ktorého pomocou implementujeme vizuálne vnímanie botov. Pre spracovanie života botov je implementovaný vlastný komponent, ktorý komunikuje so zbraňou priradenou strelcovi a cieľu, po ktorom sa strieľa. V implementácii správania botov využívame prvok poskytnutý UE4, Behavior Tree. Každú z našich 5 úloh implementujeme pomocou troch typov uzlov a to tasks, decorators a services. Pri implementácii pohybu bota používame tzv. Environment Query System (EQS), pre nájdenie a rozhodnutie sa, na ktoré miesto na mape sa má bot presunúť. Pre implementáciu matchmakingu sme použili výhradne systém Blueprintov priamo v editore UE4. Hlavné dáta použité matchmakingom sú držané v tzv. Game Instance, ktorá tieto dáta ne stráca ani pri opätovnom načítaní levelu.

Testovanie a simuláciu sme rozdelili do dvoch hlavných skupín a to základne testy pre určenie toho, ktoré schopnosti majú aký vplyv na výsledky zápasov a simuláciu s použitím dát obsahujúcich informácie o 500 hráčoch vstupujúcich do celého nášho systému. Výsledky boli v oboch prípadoch spracované a na ich základe sme vytvorili vyhodnotenie efektívnosti nášho navrhnutého a implementovaného systému. Úspešne sa nám podarilo vytvoriť a implementovať matchmaking systém, ktorý má pozitívny dopad na výsledky zápasov a na základe simulácie ho ešte následne vylepšiť aby vytváral viac rovnomerne rozložené tímy.

Budúci vývoj aplikácie môže zahŕňať ako rozšírenie schopností a správania AI botov, tak aj vylepšenie výpočtu hodnotenia hráčov a následne použitie tejto hodnoty v systéme matchmakingu. Jedným zo zaujímavejších rozšírení by mohla byť implementácia schopnosti botov dopĺňať si život pomocou bodov rozmiestnených na mape a modelovanie znalosti mapy ako jednej z ich schopností.

Simulation Based Matchmaking Optimisation

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Tomáš Milet. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....

Ivan Eštván
May 15, 2019

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Milet for all his patience, fair treatment and guidance which he provided me with despite everything. And also a big thanks to my computer for not giving up on me despite being turned on 24/7 for almost 2 months.

Contents

1	Introduction	3
2	Unreal Engine 4	4
2.1	Blueprints Visual Scripting	4
2.1.1	Blueprint Class	4
2.1.2	Data-Only Blueprint	5
2.1.3	Level Blueprint	5
2.1.4	Blueprint Interface	5
2.1.5	Blueprint Macro Library	6
2.2	C++ Programming in UE4	6
2.3	Simulation in UE4	6
3	First-person Shooter	8
3.1	Team Deathmatch Gamemode	8
3.2	Player statistics	9
4	Matchmaking	10
4.1	Elo rating system	10
4.2	Glicko rating system	10
4.3	Ratings Based On Performance Measurement	11
5	Artificial intelligence	13
5.1	Player modelling in video games	13
6	Proposed design of simulation environment and matchmaking	14
6.1	Map design	14
6.2	AI design	14
6.2.1	Skills	16
6.2.2	Behavior	17
6.2.3	Go To Cover	18
6.2.4	Follow Target	19
6.2.5	Shoot At Target	19
6.2.6	Check Surroundings	19
6.2.7	Search For Target	20
6.3	Proposed Matchmaking	20
6.3.1	Skill Rating	21
7	Implementation	22

7.1	AI bots	23
7.1.1	Damage Handling	24
7.1.2	AI's Behavior	25
7.2	Matchmaking	28
7.3	Simulating	29
8	Simulations and analysis	32
8.1	Base Line Tests	32
8.1.1	Solo Skill Tests	32
8.1.2	Versus Skill Tests	32
8.1.3	Changing Matchmaking Coefficients	33
8.2	Matchmaking Tests	34
9	Conclusion	37
	Bibliography	38
A	Contents of the DVD	40

Chapter 1

Introduction

Since the early 1990s, when online multiplayer games started to get more and more popular, thanks to the progress and availability of better internet, the need to fairly match-up and divide players into different teams became one of the main problems of such games.

At first the gaming industry relied entirely on the players' own judgment to balance teams on dedicated servers provided either by a game developer or other server provider. This proved to be problematic due to the fact that most of the players didn't have means necessary to find servers with players of similar skill.

Game developer companies were trying to solve this problem and the first dynamic matchmaking based on a ranking system was developed by Bungie and published by Microsoft Game Studios.[2] It let players set rules for a match they wanted to play, then it found a large amount of players with the same rules set, it compared the skill of each individual player using a number of criteria and divided them together into opposing teams. This development started a revolution in team based multiplayer games as the rest of the world tried to copy and develop their own matchmaking for their games.

Even though the dynamic matchmaking made life easier for players searching for a more balanced matches, it provided a new problem for game developers to solve: How to determine skill of the players so the teams facing each other are as balanced as possible. All this had to be done without fragmenting the player base into very small groups of people, resulting in a long matchmaking queue time.

This work aims to study how different metrics of skill in a team based First Person Shooter (FPS) game affect the result of matches by simulating them using a simple model of a typical 5v5 Team Death Match game mode in Unreal Engine 4. Data provided by these simulations should help us optimize and therefor improve our own matchmaking, so it creates more balanced games while not prolonging the queue time.

Chapter 2

Unreal Engine 4

The Unreal Engine is a game engine developed by Epic Games in 1998 for their FPS game Unreal. Since the first showcase the engine has developed and is now in it's fourth official version. Engine's usage has been widely expanded and has been successfully used in a variety of other game genres.[16]

It is written in C++, thanks to which it features a high degree of portability and is a tool used by many game developers today. With it's source code being available for everyone, it allows user to use two different approaches to game development. You can either use the built in Blueprint method [4] or write your own code in C++, using everything the engine has to offer without limitations.

The fact, that it was originally built for FPS games, makes it a perfect tool for our work. Which is a reason why our entire work is built in it, using both Blueprints and it's C++ capabilities.

2.1 Blueprints Visual Scripting

The Blueprints Visual Scripting system in Unreal Engine is a complete gameplay scripting system based on the concept of using a node-based interface to create gameplay elements from within Unreal Editor. As with many common scripting languages, it is used to define object-oriented (OO) classes or objects in the engine. [5]

In their basic form, Blueprints are visually scripted additions to your game. By connecting Nodes, Events, Functions, and Variables with Wires (see fig.2.1), it is possible to create complex gameplay elements (see fig.2.2). However, they are not meant as a replacement for C++. The expectation is that a programmer sets up base classes which are used to create a Blueprint of the class. These Blueprints can then use and extend upon exposed set of functions and properties of the base class.

2.1.1 Blueprint Class

Often shortened as Blueprint, is an asset which allows designers to extend the existing functionality of the base class. These Blueprints are created inside Unreal Editor visually and they essentially define a new class or type of Actor which can then be placed into maps.

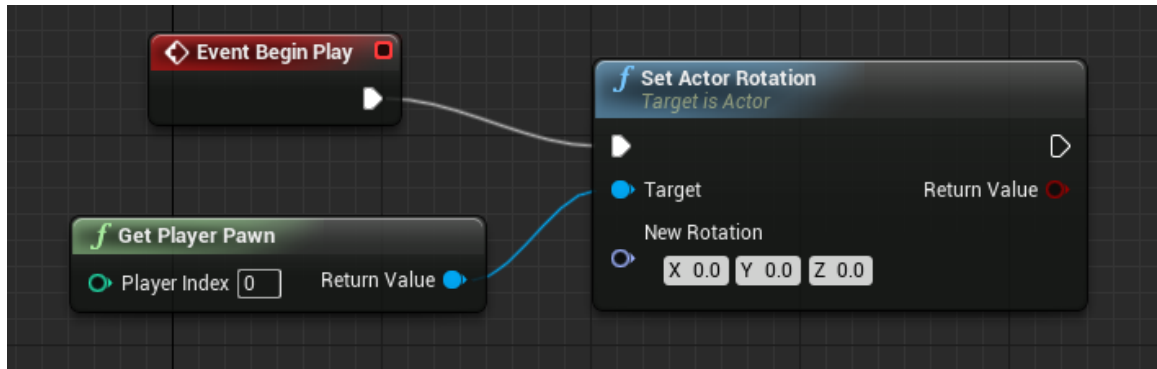


Figure 2.1: Example of a very simple Blueprint Node Graph

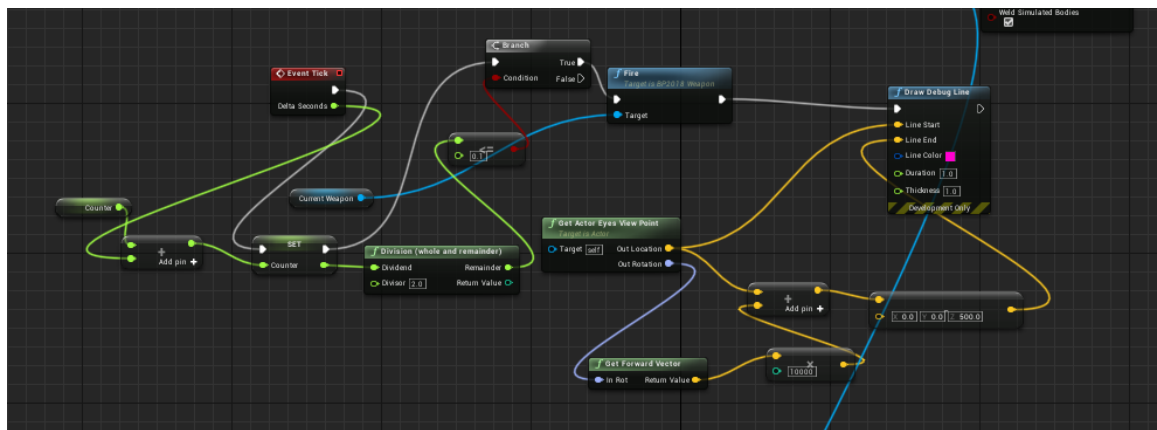


Figure 2.2: Example of a more complex Blueprint Node Graph

2.1.2 Data-Only Blueprint

This type of Blueprint contains only the code, in the form of node graphs, variables and components inherited from its parent. It allows those inherited properties to be tweaked and modified, but no new elements can be added.

It can be converted to full Blueprints by simply adding code, variables or components using the full Blueprint Editor.

2.1.3 Level Blueprint

A Level Blueprint is a specialized type of Blueprint that acts as a level-wide global event graph. Each level has its own Level Blueprint that can be edited within the Unreal Editor, however it can not be created through the editor interface.

Events pertaining to the level as a whole, or specific instances of Actors within the level, are used to fire off sequences of actions in the form of Function Calls or Flow Control operations.^[9]

2.1.4 Blueprint Interface

A Blueprint Interface is a collection of one or more functions - name only, no implementation - that can be added to other Blueprints.

This means you can have completely different types of Objects, which share one specific thing. If you create Blueprint Interface which contains a function to handle this specific thing, it will allow you to now treat the different objects as the same type. Only difference is in the specific implementations of the function inside each Object.[7]

2.1.5 Blueprint Macro Library

A Blueprint Macro Library is a container that holds collection of Macros or self-contained graphs that can be placed as nodes in other Blueprints. Macros are used to store commonly used sequence of nodes to make blueprints which use the same sequence more often look less complicated and also they can be time-savers.

Macros are shared among all graphs that reference them, they can have multiple output pins and also don't need an execution pins as long as the nodes within the Macro are not execution nodes.[3]

2.2 C++ Programming in UE4

C++ programming is a second part of the development when using Unreal Engine 4. Engine's source code is available to everyone and with it the ability to use gameplay API and framework classes within both Blueprints and C++ code. [8]

Using IDE is recommended and in our case we will be using Microsoft Visual Studio which is designed to integrate smoothly with Unreal Engine.[10]

Main idea behind using C++ is to use it to create the base gameplay systems which you can later use inside Unreal Editor and build upon for a level or the game.

2.3 Simulation in UE4

Using built in simulate function allows you to watch the game play out without interaction from players. This allows us to use our own AIs to play against each other while we can monitor the gameplay data.

Unreal Engine let's us see real time data and behavior of AIs on the level. We can also see what they are currently doing and deciding for by checking Blueprint simulation graph (see fig. 2.3) or AI's Behavior Tree (see fig. 2.4).

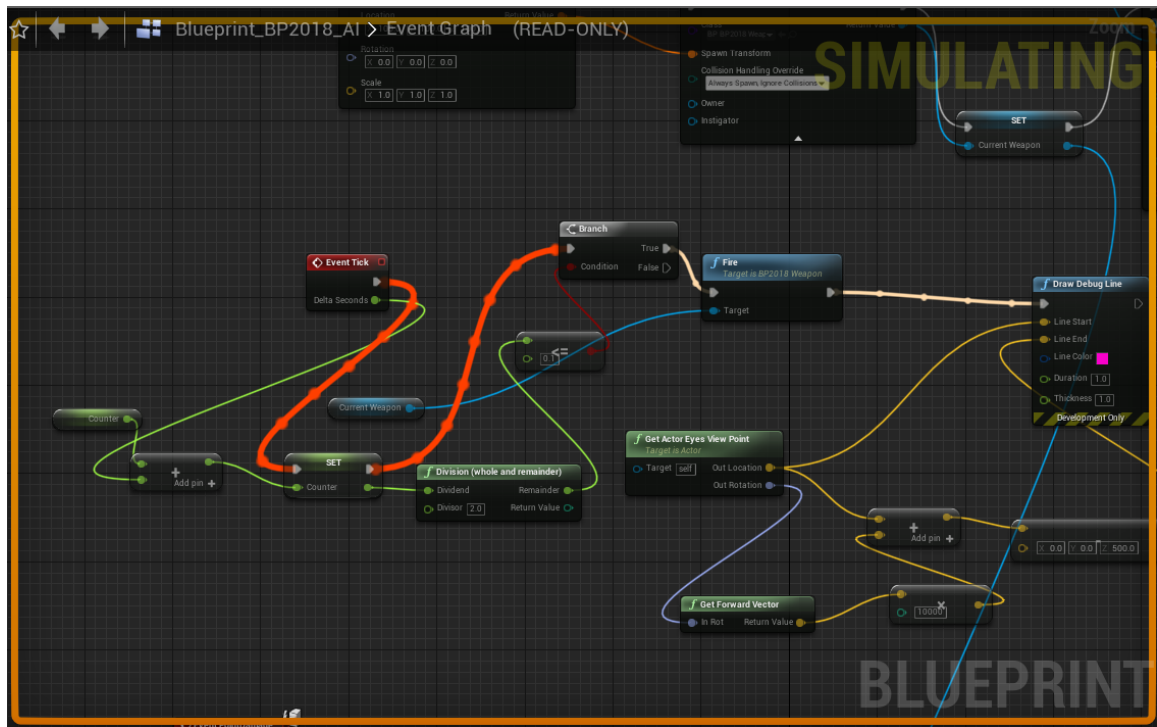


Figure 2.3: Example of Blueprint Node Graph during simulation

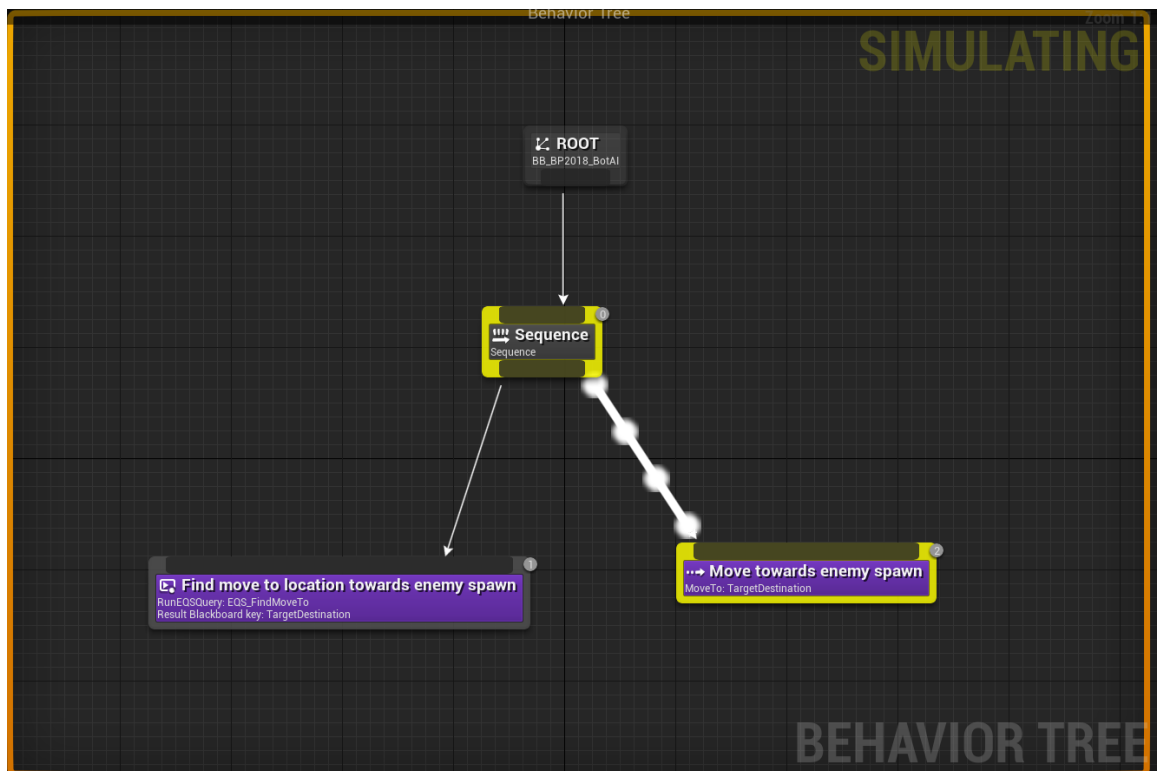


Figure 2.4: Example of AI Behavior Tree during simulation

Chapter 3

First-person Shooter

First-person shooters are one of the subgenre of the wider action game genre. They are a type of three-dimensional shooter game, featuring it's distinctive point of view from a first-person camera(see fig. 3.1). This means that player sees the action through the eyes of it's in-game character.

Like most shooter games, first-person shooters give it's players options to choose from a variety of ranged weapons, maps and gamemodes. In our work we will focus on one of the most popular e-sport FPS game mode, team deathmatch, in a game with different types of firearms.

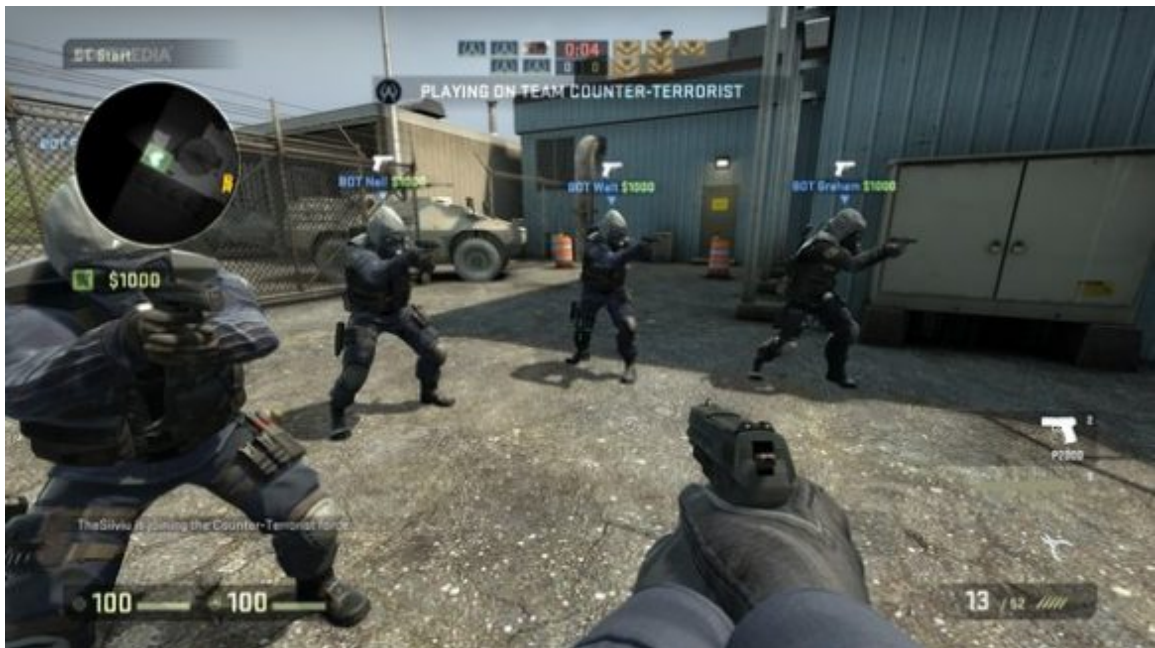


Figure 3.1: Screenshot from the FPS game Counter-Strike: Global Offensive

3.1 Team Deathmatch Gamemode

In a typical FPS Team Deathmatch session, players are divided into two or more teams, their computers connect to each other via a computer network either locally or over the

Internet. And each individual player sees the virtual level or world through his character's eyes, first-person POV(point of view).

Each team has it's own kill count. Each player contributes his kills to this team kill and the team with more kills at the end of the match wins.

In our work we will be simulating a Team Deathmatch between two teams, each with 5 players, where the goal is not to get most kills, but to kill an entire enemy team. Our gamemode won't allow friendly fire, respawns and the game is won when one of the teams kills every single opponent and has at least one player alive.

3.2 Player statistics

Since the First-person Shooters are one of the most competitive games, many of them running their own big e-sport leagues and tournaments, player statistics are gathered in every single one of them. Those are then showed to players, which allows them to see how well they are doing. They are used to determine player's skill which most often results in player's rank. Every game has it's own ranking algorithm to determine player's rank and re-evaluate it every time a player finishes a game.

In our work we will be working with a more specific set of statistics, and try to determine which of the preset AI's skills have the biggest impact on player's performance in a game, or which are sufficient enough to accurately assess player's skill, to achieve as balanced matches as possible.

Some statistics we will be using to determine player's performance are:

- Accuracy - amount of hits on target to shots fired
- Reaction Time - how fast can player react to different in-game situations like getting shot or spotting enemy
- Awareness - frequency of checking players surrounding

Chapter 4

Matchmaking

Matchmaking has become an essential part of every team multiplayer game. It's main purpose is to make finding matches easier for a large amount of players, without needing any input from them. As games developed into competitive environment the need for a balanced teams became a necessary upgrade to the previous matchmaking systems.

Here the gaming industry looked into other zero-sum games, such as chess, for inspiration. Zero-sum games are games in which each participant's gain or loss of utility is exactly balanced by the losses or gains of the utility of the other participants. [17]

Two main rating systems used in multiplayer games nowadays are Elo rating system and Glicko rating system, the latter being a natural improvement of the Elo system.

4.1 Elo rating system

The Elo was originally invented for a chess as a replacement for the old Harkness system, but it became a key feature in many of today's video games as a system to create balanced matches.

A player's rating is represented by a number which is calculated after each game. In chess, the calculation is easy as there are only two players, usually with a different rating. These ratings are based of a number of wins, defeats and draws of each individual. Then after the match is over, the points deducted or add to players rating, are calculated from a difference in their original ratings. Player with high rating is expected to win more often than not when facing a player with low rating, so if the higher rated player wins there are fewer points added to his rating and also the losing side loses fewer points. However, if in the same situation a player with lower rating won, then this would result in more points added to his rating and also more points deducted from the loser with higher rating. The reason for this is simply because it's less probable for a low rating player to beat a high rating player, so if he wins he should make a bigger jump up the ranking table, than after winning against a significantly weaker opponent. [14]

4.2 Glicko rating system

Glicko rating system, being an improvement of the Elo rating system, is not very different to Elo system. It's contribution to measurement is ratings reliability, called RD, for ratings deviation. [15]

Ratings deviation (RD) measures the accuracy of players rating. If the player's results over a period of time are fairly consistent, his RD will be lower than the RD of player whose result are more unpredictable. As an example let's have a player with rating of 1500 and his RD is 50, than the real rating used by Glicko system during matchmaking will be between 1400 and 1600. After the game change to the RD is smaller when it is already a small RD or the opponents RD was very high, since the latter says we don't know the true strength of the opponent, yet. On the other hand if player with big RD matches up against a player with very small RD, which tells us his rating is fairly accurate, then after the game, RD of a player who started match with big RD can change substantially, because we can compare his performance to someone who's skill is evaluated to high certainty.

It calculates rating in rating periods. This period should be long enough to contain good amount of games which are then considered as simultaneous for this period of time.

Glicko rating system has been further improved into Glicko 2 by adding rating volatility. This volatility indicates how consistent the player is with his performance and let's us deal with sudden changes in player's performance.

It becomes overly complicated for large amount of players and time periods, therefore isn't well suited for use when we have to match many players together in a quick succession for multiple matches. [11]

4.3 Ratings Based On Performance Measurement

Performance based matchmaking, also referenced as Skill Influenced Matchmaking (SIMM), uses directly observable data about player's performance to produce a single number representing player's skill. These can be such as players kill count, his overall accuracy, his reaction time to different situations and many others.

Formula to calculate player's rating has to be different for every game since it is based on many factors which are influenced by the specific game design. Formula contains many arbitrary parameters to better fit the game it's being used for. This way we can give more weight behind factors which have bigger influence on the final result.

Once these values are calculated, matchmaking tries to make the best match between players of similar skill and put them into games. There are usually two different approaches to achieving as balanced games as possible.

One way is to balance each player against an equally skilled opponent, by doing this for the needed amount of pairs, system gets a well balanced match where every player has it's opponent of equal skill. Main advantage of this can be the fact that with a big enough pool of players it almost guarantees a well balanced game, if the skill calculation is accurate. Disadvantage is that it needs a bigger pool of players, because with smaller pool the gaps in skill when finding an opponent can be greater, or the time to create match is longer because matchmaking has to wait for a player that fits to fill up the match.

Second way is to avoid balancing each player against it's equal opponent, but to balance skill rating of teams. This way we can mitigate the previous disadvantage of waiting for an equally skilled player to fill up a match, because we can be more dynamic and just switch out players to achieve a team balance. However, by doing this we can cause a big problem by underestimating or overestimating influence of individual or group of players. For example, putting together a Team 1 with 2 very high skilled players and 3 below average players, against opponent Team 2 with 1 high skilled player and 4 average players, can cause an unexpected one sided game, where, and this varies based on the individual game, teams can be balanced rating wise, but still more often than not, end up in a very one way result where

one team overpowers the other since there is not a guaranteed counter to every individual players.

Chapter 5

Artificial intelligence

Artificial intelligence is a technology used in development of computer related programs that need to mimic and use a human like intelligence such as reasoning, learning and problem solving.

In online video games it is often used to create a bot, or NPC (Non-Player Character), that simulates to play in a human style. This way it then provides a sensation of being faced by other human player that is in fact non-human character.

These Non-Player Characters should have high level of perception. This means that they should move, attack and react as close to a human controlled character as possible. They should use senses like sight or hearing and be affected by different events the same way as a player controlled character would be. [12]

5.1 Player modelling in video games

Player modelling is a research area of video games development that is gaining more attention from both game developers and game researchers. It concerns generating models of player behaviour which are needed for determining accurately, and adapting the player experience. Even though, the main goal of player behavioural modelling is to steer the game towards an overall high player satisfaction, they can be used to get useful information by using them in different simulation environments.

This way game developers can test whether the map leads to the gameplay as envisioned by the designers, simulate different story lines or test online game's matchmaking to see if it works properly and provides a satisfactory matches for all teams involved.[1]

To successfully model player like behaviour in video games we have to carefully translate different game aspects into simplified versions which we can base our AI logic on. If we don't want our AI to be predictable a large chunk of AI actions will be based on random decision logic to mimic different player approaches to different situations. Example for this could be a simple take cover action. We know, that we want our AI to go and take cover from the enemy when his health drops below certain threshold, but we don't want our AI to take cover behind the same single object every time we run the simulation. This decision can be then influenced by many pre-set parameters or even some actions which happened previous to AI's current decision.

Chapter 6

Proposed design of simulation environment and matchmaking

6.1 Map design

To ensure the most accurate simulation environment we will be using a simple arena design with two spawns and follow a simple point reflection. To add some map depth we will have two areas, one on each side of the map, with elevation change allowing our AI to get overlooking position over some part of the map. It has been designed to allow mostly for a close quarters combat, but with a chance for a long range shooting fight in the middle of the map. This has been ensured by wall's height, which is high enough to completely obscure the AI's sight sense. Central part of the map is designed to be the main area for a medium range fights with a small obstacle in the middle, this is the only obstacle on the entire map which is not high enough to completely obscure AI's view.

Final map layout can be seen at fig. 6.1 with following points of interest marked on the plan:

- A - point used for point reflection, point reflection has to be as accurate as allowed by the Unreal Engine 4
- S1,S2 - spawns for Team 1 and Team 2, each spawn has 5 symmetrically placed spawning points for our AI bots, where 4 of them are next to each other and 1 is in the middle behind them, distance between every one of them has to be the same
- O1,O2 - elevated part of the map overlooking bottom edge and top edge, respectively, should allow bots a chance for a long range fight, bots from O1 should not be able to shoot bots on O2 which are covering a mentioned part of the map and vice versa

6.2 AI design

In video games, AI model should closely follow real player's model and be similar to them in as many possible ways as possible. However, our goal is to decide the impact of different skills on result of games and then balance matchmaking around gathered data. For this we won't need visually accurate model, very simplified version will be sufficient enough.

Our proposed design is a simple ball with diameter equal to half of player's height, so it can fit into all parts of map without having to change it's shape. This way it will also

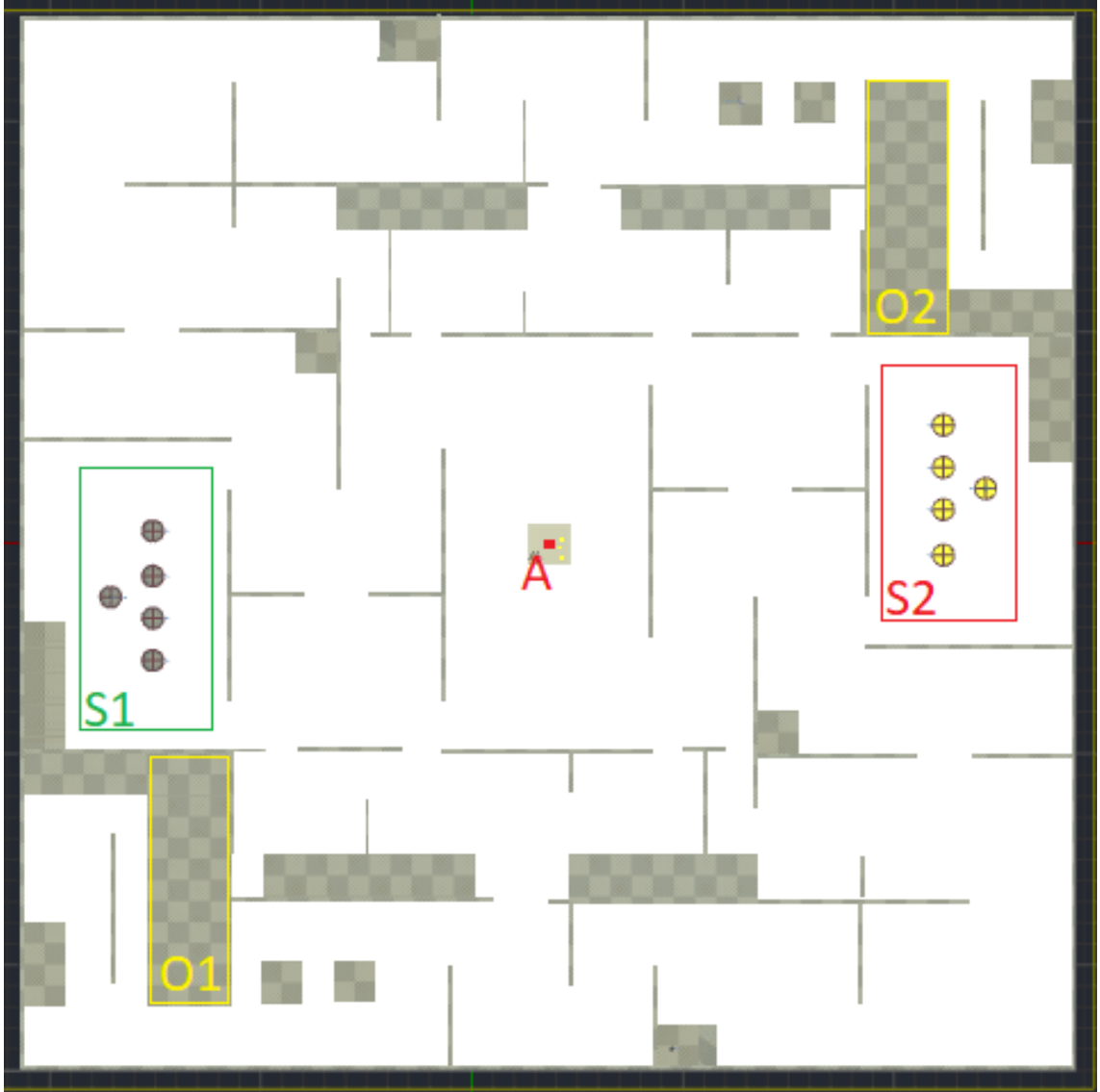


Figure 6.1: 2D view of a proposed map design for our simulation

allow a better flow when two or more models meet on the map in tighter areas, without sharp edges we can minimize a chance of getting stuck into each other. We won't be modelling more vulnerable areas, such as head or chest of player model, since it doesn't provide additional information for our work because to keep simulation simple bots will be aiming at the middle of enemy target with random bullet spread and therefore it would just increase randomness and not skill.

Weapon's slot and senses should be positioned on the very top of the ball and always face the forward direction of the ball. Sight sense should have at least a focused field of view cone of 70 degrees. It is not as big as real player's field of view, but should be sufficient in our environment and also not big enough so bots can still miss an enemy target.

All bots should be physically equal in mass, have same friction and gravity applied to them, and follow the same default movement logic, which includes movement speed and ability to jump of ledges or sprint. They have to be able to rotate 360 degrees around their

yaw axis and change their pitch angle, both allowing them to shoot enemy target anywhere around them.

6.2.1 Skills

Accuracy

In military term accuracy of fire refers to the precision of fire expressed by the closeness of a grouping of shots at and around the centre of the target. In our case however, we will use term accuracy for player's skill to improve the overall accuracy of his weapon when shooting. This way we can simulate a simplified version of recoil control, which is a very important skill in FPS games.

Weapon's accuracy should be defined by the default bullet spread of the gun. Bullet spread is a value defining the angle of the biggest possible deviation from the shot direction and shouldn't change during the gameplay. (see fig. 6.2)

Bullet spread will be set in degrees and accuracy is a value from 0 to 1, where lower is better. With this set, we can take weapons bullet spread and multiply it by player's accuracy to get final bullet spread used in a match for each player. (6.1)

$$FinalBulletSpread = WeaponBulletSpread * PlayerAccuracy \quad (6.1)$$

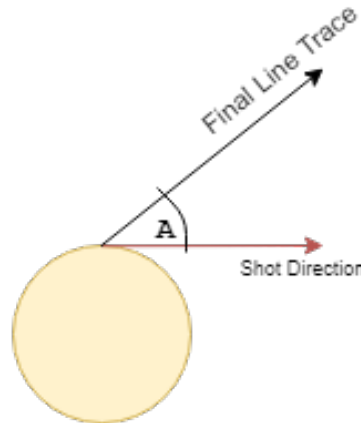


Figure 6.2: Simple example of maximum bullet spread in 2D space where A is the Bullet Spread angle in degrees.

Reaction Time

Reaction time is the time that elapses between a player experiencing a situation and the player initiating a response to the situation. Mean simple reaction time for young adult individuals have been about 190 ms for visual stimuli and 160 ms for sound stimuli.

There are many factors affecting reaction time, most of which result in slower reaction time. To name a few which have the biggest impact:

- Age - shortens reaction time from infancy into the late 20s, then increases slowly until the 50s and 60s, after that it lengthens even faster

- Fatigue - increases reaction time more for harder actions and affects young adults more than people in 50s
- Distraction
- Alcohol

We will be modeling reaction time as value from 0ms to 999ms, lower is faster therefore better. This way we can cover all possible situations from very fast reaction times of young players to very slow reaction times of old players or players affected by one of the factors, and also include some super human reaction times and see their effect on gameplay.

Reaction time in our model should affect at least player's reaction when seeing enemy, getting shot by enemy or before attacking enemy. These three are the most important parts of FPS gameplay and have the biggest impact on result which is why we will be focusing on them. [13]

Awareness

Awareness is a hard and unusual skill to measure so there is no real data from any game which would measure this. However we are not going to have to measure it because we will be using it as a way to model AI's knowledge of it's surroundings during match and it's behaviour outside combat.

It will be modeled as a value from 0 to 1, if we divide it by 100, we can look at it as a % of match time spent gathering information about surroundings. Higher value means player is more aware of his surroundings and current state of the game. It should affect players reaction to his health dropping, or frequency and speed of looking around to see if someone else is following him.

Simulation should show us, that having very high awareness value doesn't have to have positive impact on player's performance in match. That's because if he looks around very often, he can miss more information than gain.

Movement Skill

Movement skill will be modeled for our use as a value from 0 to 1, same as awareness, so we can transform it into % value which can be used in simple decision making. It represents amount of match time spent with different movement logic than default. To simplify simulation this logic should have affect on movement speed and in decision making process when finding next location point where AI is going to move to next.

6.2.2 Behavior

As we have established, that we will be modeling a simple First Person Shooter game environment our AI bots have to behave similar to player, but since our purpose is gather data in matches where only AI bots face each-other we can simplify the final behavior logic, because we are not trying to give impression that AI bots are real players as we would have to in game with real player controlled characters. We are mainly interested in the final results and the way they are affected by different skills and matchmaking settings.

Therefore we propose this simple behaviour tree (see. fig 6.3), with 5 main actions that AI has to do during gameplay. AI has to decide for each of these actions based on the gameplay situation, but actions on the left have higher priority than actions on the right side of the behavior tree.

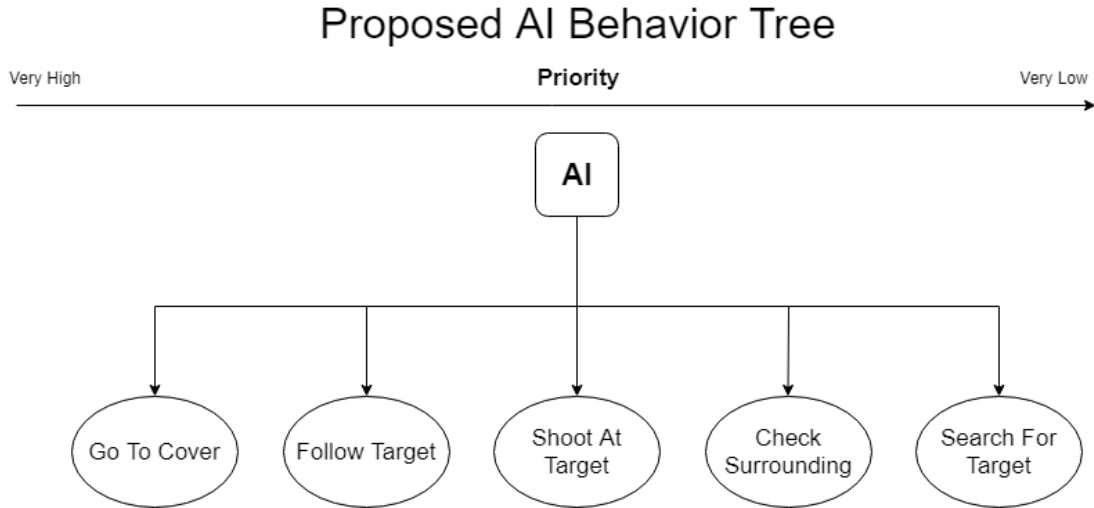


Figure 6.3: Basic AI Behavior Tree Proposition

6.2.3 Go To Cover

Go to cover action is placed on the most left side of the behavior tree, because it has to have priority over everything else, since AI's life in match depends on it. AI should run and hide from the enemy attacking it, when it's health dips below certain threshold, suggested value is 20. Because we don't want all AI's to constantly run for cover and avoid any fights, this action has to be limited by 2 factors:

- Overall Skill
- Average value of awareness, movement skill and reaction time

Overall skill should affect this because we want to model a situation when new players, or players who are not as experienced in game, do not run from a fight because they simply don't consider it as a good option.

For the second factor we chose those three skills for the following reasons. Awareness tell us how aware is the player of current gameplay situations, this includes his own health, so less aware player simply doesn't have to pay enough attention to his health to make a decision to run. Reaction time should affect this because it tells us if player is able to react fast enough when his health drops below set threshold. Players with slow reaction time, might be aware that his health has dropped, but does not have fast enough reaction to make a decision to go and find cover. Movement skill is here because it has big impact on the speed in which the AI finds cover. AI with high movement skill can outrun his opponent which is trying to kill him, get into good position and fight back when the opponent catches up.

Position of cover will be set as position where opponent can't see the AI looking for cover, is at least 500 distance units away from opponent and even further away is better, path to the cover position from AI's current position is the shortest and it's somewhat close to at least one friendly AI.

6.2.4 Follow Target

Follow target is a simple action for our AI. If AI has spotted an enemy it gets set as a target actor. When the target gets out of line of sight, so our AI can't see him, it decides to follow him until it can see him again and has enough time, this depends on reaction time, to start shooting target, when this happens AI jumps into *Shoot At Target* action.

AI should always follow set target as long as it is set. Once the target is destroyed, or hasn't been seen for a period of time, it gets cleared from AI's memory. The only skill that has any effect on this action is *Movement Skill*. It should allow our AI to make a decision between running or walking when following target.

This entire action can never happen without at least one target actor being set.

6.2.5 Shoot At Target

This action follows the same simple rule as *Follow Target*. If it wants to be executed, it has to have at least one target actor set. Once that is set, AI's focus should be set on target actor.

AI has to make a decision about which target is the best to shoot at in current situation and then it has to stop moving in order to start shooting. It should reset it's walking speed in case it was sprinting prior to stopping. After this *Reaction Time* should come into play and affect a delay between getting into position and start of shooting at the enemy target. Shooting accuracy is affected by players *Accuracy* skill, other than that gun's rate of fire should stay at it's default value which is proposed to be 7.5 shots per second (450 shots per minute) and each shot should cause 10 points of damage to the hit target, since there is no friendly fire we can expect that our hit target will always be shooter's opponent. Shooting should continue until the enemy target is either destroyed or breaks line of sight.

6.2.6 Check Surroundings

Awareness skill defines our bot's ability to understand and check his surroundings. It is defined as a value from 0 to 1, where higher value means better skill. This skill gives our bot a decision making capability to decide when he wants to look around himself and check if there is enemy target somewhere else than in front of him.

This decision should take place randomly based on the situation and it decided purely by Awareness skill. Once the bot decides to look around, he should stop and take his time to look around. Speed of which he is looking around is also defined by awareness skill, more aware bots will look around faster than their less aware counterparts. This way we can model a case when more aware player can notice same things, but it doesn't have to spend a lot of time doing so. While checking around all senses should rotate with the bot. Once bot finishes his full rotation, it then continues looking for a player. However, if bot notices enemy target while looking around, it should stop it's current action and start engaging with opponent.

Proposed base rotation time for a bot should be 0.8 seconds. It should take our bot 8 repeats to finish a full circle, because on every time interval it rotates by 45 degrees so *DefaultSectionRotationTime* is 0.1s. This gets faster based on the *Awareness Skill*.

$$FinalRotationTime = (DefaultSectionRotationTime - \frac{AwarenessSkill}{10}) * 8 \quad (6.2)$$

We are expecting there will be a visible maximum for awareness skill, where everything above this maximum will have opposite effect on player's performance because he will be either checking too often or too fast to notice things. This is intended and it's our simplified way of modelling player's who try to check too often, but don't pay enough attention or do it way too fast.

6.2.7 Search For Target

This action should have the lowest of priorities compared to the other four, however it is still very important. AI is running in search for target loop when it doesn't have any targets to engage with.

We don't want our AI to be cheating and to know position of it's targets so they should be walking around the map trying to find enemy. At the start of the game their priority should be walking around enemy spawn, since that's a place where enemy should be. Later in game, they should start checking around their own base and entirety of the map. AI should never be stationary for a long period of time.

While searching for target the only skill having any affect on this action should be *Movement Skill*, which should allow AI's with higher value in this skill to move faster and more effectively around the map.

6.3 Proposed Matchmaking

For our simulation we propose a simple skill influenced matchmaking which we will try to optimize based on the simulation results. Our matchmaking should be unbiased and match players only by their skills. We will be creating two teams, each with 5 players, balanced on a player-to-player basis, so we are not going to be balancing team's combined ratings, but rather each individual player should have an equally skilled player on the opposing team.

In our simulation we will provide matchmaking with a set of data containing information about players. This data set will act like a current matchmaking queue of players. To create matches, matchmaking should be looking over this set of data and finding the best possible match combinations with the lowering amount of players.

To create a match we have to find 5 pairs of players with their skill rating as close to each other as possible. We always start with one player, we calculate his *Skill Rating* and then start looking for the best match from the rest of players which are still in queue waiting to be placed into a match. To find the best possible match we should use a *Matchmaking Interval*, which is a number defining our default interval in which the *Skill Difference* (see eq. 6.3, eq. 6.4) between Player1's skill rating and Player2's skill rating has to be to be considered as a match between players. Proposed default interval is +25 and -25 from Player1's skill rating. If we don't find a matching player in queue we will increase *Matchmaking Interval* and repeat the process of looking for Player2(see fig. 6.4).

$$SkillDifference = Player1SkillRating - Player2SkillRating \quad (6.3)$$

We get a matching pair of players if:

$$SkillDifference \in \langle -MatchmakingInterval, +MatchmakingInterval \rangle \quad (6.4)$$

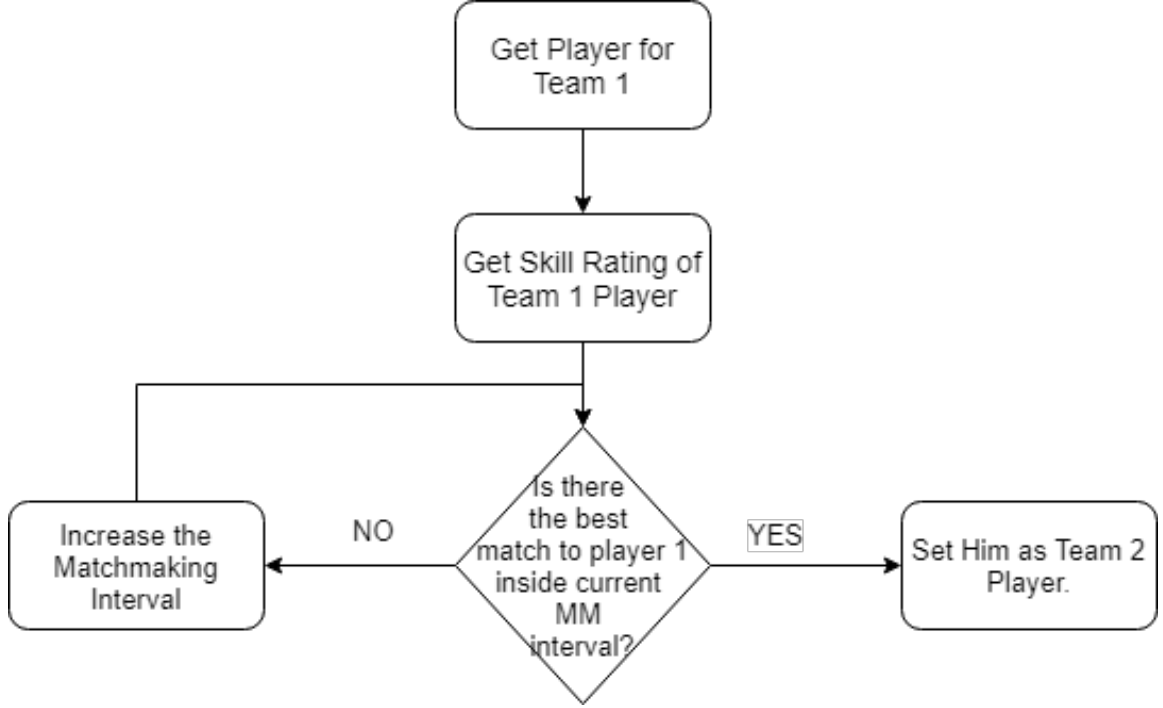


Figure 6.4: One cycle of Matchmaking looking for a pair of matching players.

6.3.1 Skill Rating

Skill rating is a calculated value used to rate player's performance in our game. In our simulation we will be using 4 skills, which we have already talked about, to measure player's performance. This value is then used in our proposed matchmaking to find players of equal skill to play against each other.

This way we want to show that even simple skill rating, based on the skills which are easily measured if needed, can have a big impact on the final balance of matches.

Proposed function to calculate player's skill rating:

$$SkillRating = (1000 - Accuracy * 1000) * a + (1000 - ReactionTime * 1000) * r + Awareness * 1000 * w + MovementSkill * 1000 * m \quad (6.5)$$

All skills are values from 0 to 1 with three decimal places, while *Accuracy* and *ReactionTime* are better when smaller, the *Awareness* and *MovementSkill* are better when higher.

We then take these values and multiply them by 1000 to get value above zero, but without decimals. Since we want our matchmaking to allow us tweaks to the *Skill Rating* based on the simulation results, we then further multiply each skill by *Importance coefficient*. These coefficients, *a*, *r*, *w* and *m*, tell us which skill has bigger impact on a game's result. They are values from 0 to 1, allowing us to well balance our system using decimal values, but their default matchmaking value should be set to 0.5 so we can not only make something more important, but also lower it's importance from the start.

The highest Skill Rating a player can get with a default coefficient settings can be 2000, lowest possible is 0. This extremes can vary once we adjust our matchmaking based on the simulation results.

To ensure simulation's accuracy and consistency, once the AI's skills rating is set and calculated it can't change during simulation.

Chapter 7

Implementation

In this chapter we will be talking about final implementation of the proposed design in Unreal Engine 4. Reason behind using Unreal Engine 4 is because I wanted to learn and understand it's main feature of combining written code in C++ with Blueprints.

To have a good testing environment I have created the proposed map directly within the engine using only the built in geometry elements. Everything is created from simple boxes to keep the demand on computing resources as low as possible once packed as final project. I have also limited the lighting to the bare minimum for the same reason.

For navigational purposes I am using a built in *Navigation Mesh*, which is set to include the elevated position in the corners, accessible by stairs from both spawns. Other elevated features on the map are inaccessible for our AI bots. Map size is 50x50m with outside walls being 10m high and most of the inside corridor walls are 3m high. (see fig.7.1).

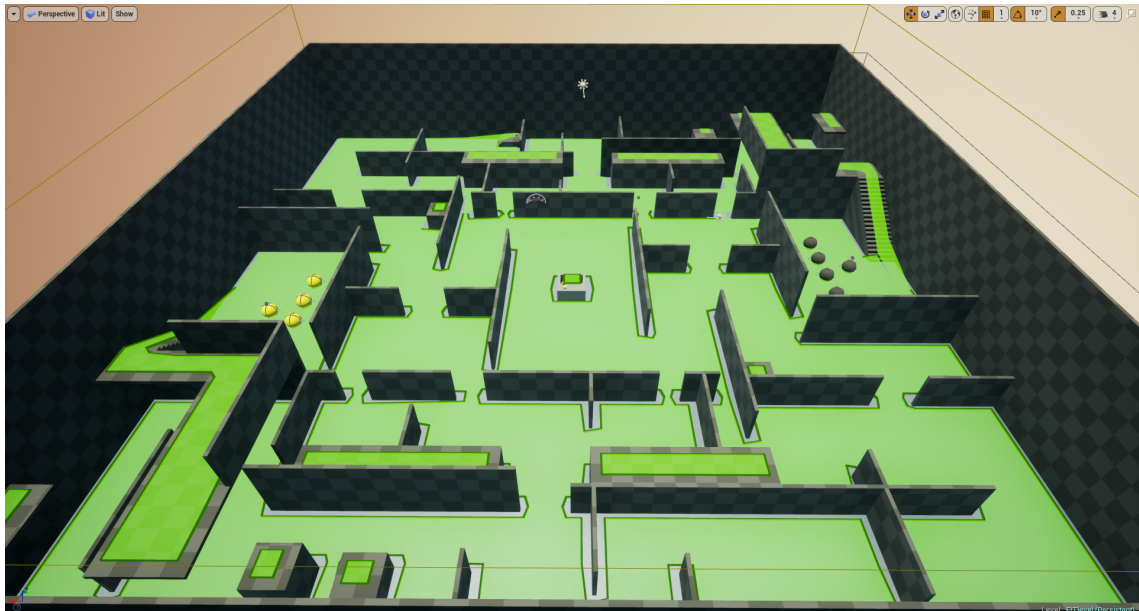


Figure 7.1: In UE4 editor view of the navigation mesh (green) in our project.

7.1 AI bots

Our AI implementation starts with a C++ character class. We are setting everything we will need later in Blueprint here in a C++. All of the variables for skills are protected with corresponding functions to get and set their values, these functions are public and have to be set as visible for our UE4 Editor. AI has to have a health component so we are using our custom BP2018_HealthComponent class and add it to our BP2018_AI class. AI's default health is 100. This is all a basic setup of our AI class in C++.

In UE4 editor we create a blueprint of our AI class. To our blueprint AI class we add *AI Perception* component. This component let's our AI to see other objects on the scene. We are using a *Sight radius* of 3000, which is roughly a 3/5 of distance from one side of the map to the other and *Lose Sight Radius* of 3500. Difference between the two is, that the first tells us in what radius we can see our target for the first time while the second one tells us in what radius we can still see our target after we have already seen it. Field of view is set to 70 degrees from the AI's forward vector. AI should be forgetting seen targets after 5 seconds of not seeing them.

We are also adding a static mesh to give our AI some physical body. This body is represented by a sphere. We are not adding any animations, since those are not important for our simulation. With static mesh added we have to adjust it's physical collision model to match our sphere and add a weapon socket. This socket is placed on top of the sphere, at the same position as our AI perception component and facing in the same direction (see fig. 7.2).

On *Begin Play* event we add weapon to our AI's weapon socket, set newly attached weapon it's new bullet spread which is based on owner's *Accuracy Skill* and attach our AI Behavior Tree to it.

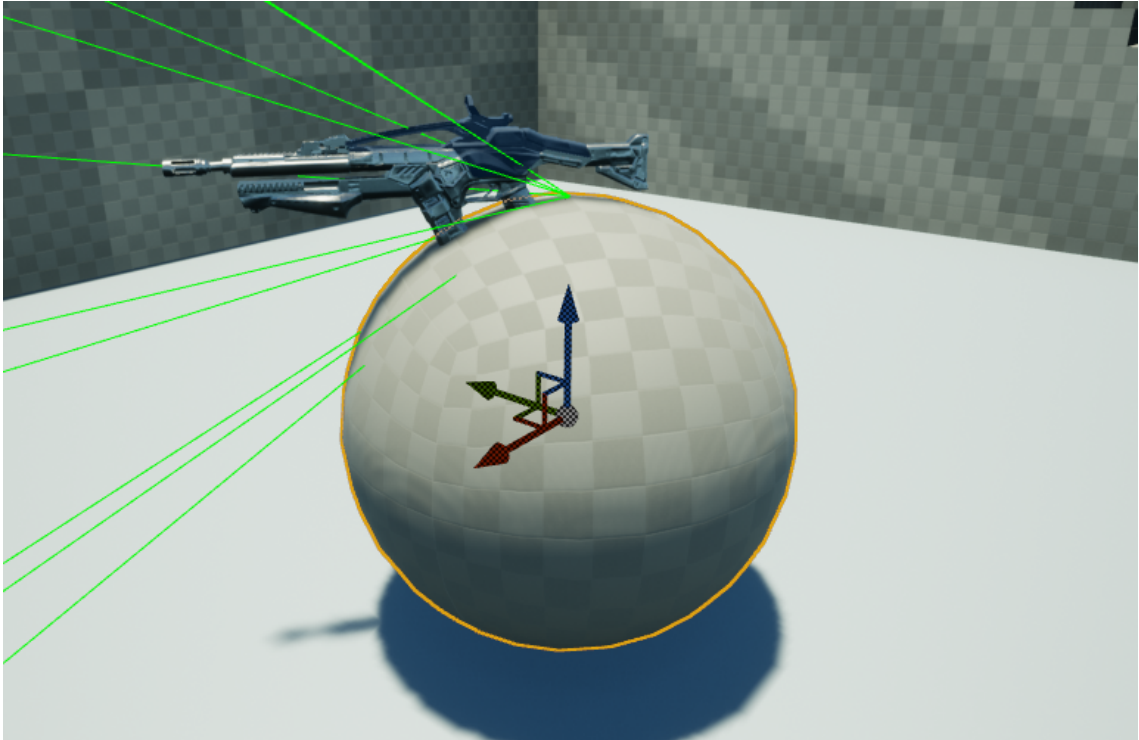


Figure 7.2: Our AI bot inside the simulation. Green lines represent sight sense FOV.

7.1.1 Damage Handling

Causing damage to opponents is a key feature of First Person Shooter. We will be handling this in three steps. First we have to have a damage causer, in our case it's weapon, second we have to handle taken damage, this is done inside our Health Component, and last step is to react on health change inside our AI.

Causing Damage

Damage is caused inside a *Fire()* function of our *BP2018_Weapon* class. Every weapon has it's own base damage, rate of fire and bullet spread. In general, we use base damage of 10, rate of fire of 450 shots per minute and default bullet spread of 10 degrees. These values are set inside weapon blueprint.

Inside our *Fire()* function we have to make sure that we set proper parameters for our line trace query. Our shot starts at bot's POV and is shot in direction where our AI faces. To not be 100% accurate we apply our bullet spread angle to this direction, by converting it into radians and then generating a random vector inside a cone with opening angle of our bullet spread, which is our final shot direction. If our line trace hits a valid target, we then proceed to apply a point damage to the target's health component.

Handling Damage

Handling any type of damage not only point damage takes place inside our *Health Component*. When the damage is applied on our AI bot, we have to do a couple of checks to make sure we handle it properly. Caused damage can not be below 0, since that would not make sense and we have to make sure that the caused damage is being applied by a non-friendly AI.

This process is handled by a *IsFriendly()* function within our *Health Component* by comparing assigned team numbers of both *Damage Causer* and *Damaged Actor*. If we are sure that it isn't friendly fire, we then change an AI's health value and broadcast a signal that the AI's health has changed.

Health Changed

Broadcast signal from within the *Health Component* is handled by our AI both inside C++ and Blueprint. Inside our C++ we have to check whether AI's health is still above 0 and if not we are stopping AI's movement and disabling AI's collision model right at that moment to prevent it from any further actions with zero or below zero health. After a brief moment we are destroying our AI from the level.

We are using Blueprint to draw and print info about the hit into console, but more importantly we are adding a damage causer to a list of AI actors which have caused damage to the *Health Component*'s owner, which is later used to select targets for our AI inside AI Behavior Tree. Since we are adding and attaching our weapon to AI inside Blueprint, we also have to destroy attached weapon object in case, that it's owner died after Health Changed.

7.1.2 AI's Behavior

One of the main strengths of the Unreal Engine 4 is its AI capability. It allows us to create a behavior tree filled with tasks and decisions for our AI. We are modeling the proposed behavior tree (see fig. 6.3) all inside Unreal Engine 4 using three different types of nodes:

- Tasks - they tell our AI what action he has to do in its current stage
- Decorators - allow us to model all types of decision making
- Services - provide necessary data for our behavior logic

Search For Target

Since search for target has the lowest priority it doesn't use any decorator, because it has to be executed only once all other decorators fail to succeed. The most crucial part of our Search For Target system is moving around the map in more or less random way, with some basic guidelines mention in the proposition. To do this we use *Environment Query System*, which is a system used for data collection about the environment, asking questions of the data through different types of tests, then returning then one item that best fits the questions asked. [6]

In our case we are using three tests (see fig. 7.3), first to make sure we get a point of interest further away from Querier, which is our AI bot, second test is to get a position closer to the enemy spawn location and third check is to make sure that we can actually get to the point by discarding all unreachable points. We get a set of points generated by this query, each rated by some score, which of whom we choose a single random item from the best 25% scored items.

Once we have our destination point, we set it as a *Target Destination* and use built in *Move To* task node to reach this point. Our AI is then looking for targets on its way to the new location and everything it sees is then passed to a list of possible target actors.

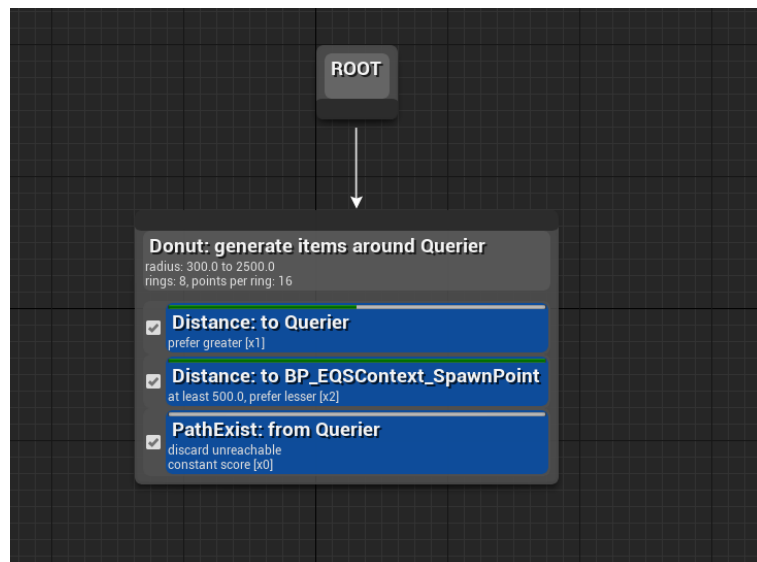


Figure 7.3: Our Environment Query System inside Unreal Engine 4 Editor used to get a move to location when searching for targets.

Getting Target To Attack

Getting a target to attack is a main goal for our AI. To get it, our AI has to move around the map, spot target and then decide whether it's enemy or not and also whether it is the best possible target. To make this decision we are using a service node (see fig. 7.4), which is being executed 5 times per second and with each tick it is completely evaluated from scratch.

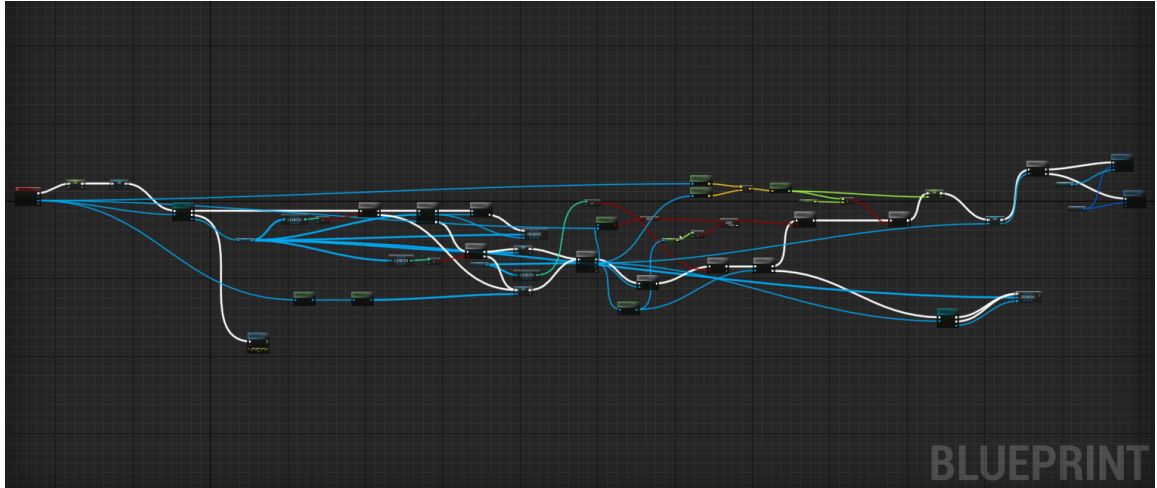


Figure 7.4: Zoomed out view of the Select Target service blueprint within Unreal Engine 4 Editor.

We have two options to choose our target from, the first option is from a list of actors which attacked our current owner in some time during the game, the second option is from a list of actors which have been spotted by current AI's sight sense. Targets from a list of actors which shot our AI have higher priority so if the list is not empty we are choosing the best possible target from there, if the list is empty we are choosing from spotted targets. The best target is based on the distance from AI to the target, we want it to focus on the closest targets first. Since targets from both lists are removed after they are killed or some short period of time we avoid cheating when our AI would know a position of the target even though it has not seen it or got shot by it for a long time.

When we already have a target set but start getting shot by another enemy, we evaluate which one is a better target to focus on and update our target actor.

Attacking Target

Once we have our *Target Actor* set, our AI can finally proceed with attacking this target. If our AI can clearly see the *Target Actor* it changes it's default focus towards it, basically it turns and looks directly at the target. It then starts to shoot with a delay based on the *Reaction Time*. Every shooting task is executed for 1 second, which is enough to fire 7 shots based on our default rate of fire. Then we have to check if our target is still valid and if so we repeat entire process. If we lose line of sight with the *Target Actor*, shooting gets aborted and we try to recover it.

Recovering line of sight with the target happens by following our *Target Actor*. This only happens if it is still on one of the lists mentioned above so again, we won't be trying to recover line of sight with deleted *Target Actor* or if we have lost contact with it for a

long time. Reason behind losing contact for a long time can be our AI's ability to run for cover.

Getting To Cover

This action has the highest priority in our behavior tree, because AI's live depends on it. It can abort any lower priority action and is using a decorator *CheckLowHealth* within which we take into consideration 3 of our 4 AI skills and also overall *Skill Rating*, everything following our proposed design.

Inside our *CheckLowHealth* decorator we have two variables, *Lowest Health* and *Min-SkillRating*, which can be set from other blueprints. First one is used to compare with AI's real time health and if AI's health drops below it we then proceed to deciding whether AI can run for cover. In this decision process the second variable comes into play, by limiting the minimum overall *Skill Rating* which our AI has to have to even consider running away, this has been explained in proposition. If our AI is able to run away, it's decision to do so is based on the result of comparison between randomly generated value from 0 to 1 and it's average value of *Reaction Time*, *Awareness* and *Movement Skill*. The higher the average value is, the higher is the chance for our AI to decide it's time to run for cover.

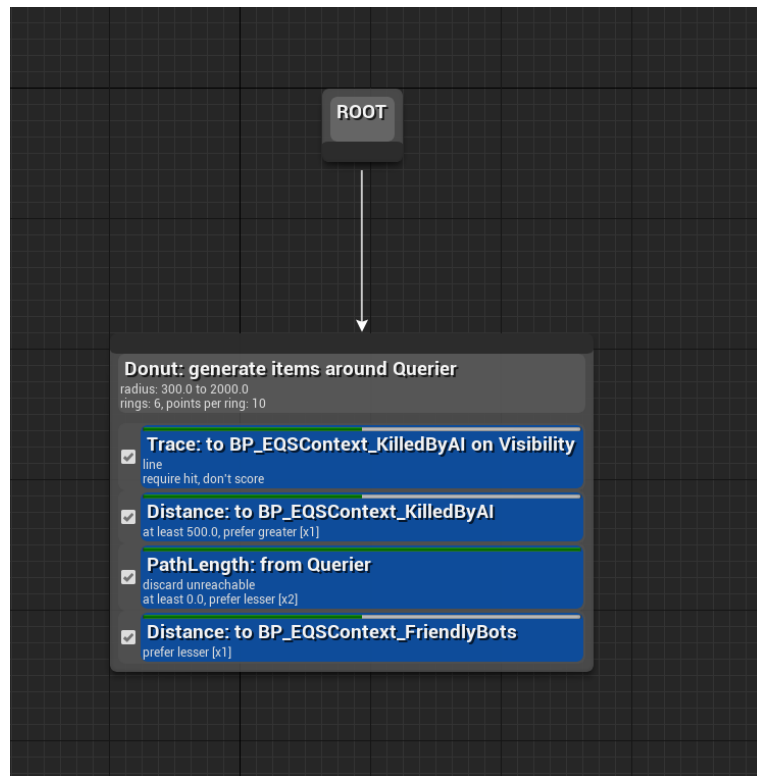


Figure 7.5: Our Environment Query System inside Unreal Engine 4 Editor used to get a location providing cover.

Position of cover is found by another Environment Query (see fig. 7.5). In this query we are running 4 tests. We are looking for a location from where we can not see any of the AIs shooting at our *Querier*, and is at least 500 distance units from them, also the path to the location has to be reachable and we will score more points for location point which is closer

to our *Querier*. Very important test is the last one, we want the cover location to be as close to some of the friendly AI's as possible. This way we simulate a player running towards his teammates expecting their help with the enemies shooting at him. Once again, we are choosing a single random item from the best 25% locations provided by our Environment Query and setting it as a next destination for our AI. After reaching this destination AI has to decide whether it wants to run again or it goes and does a different action.

7.2 Matchmaking

Entire implementation takes place inside Unreal Engine 4 Editor using blueprint. On the beginning of each simulation instance we have to create matches which are gonna get simulated in that instance. Our matchmaking implementation takes up to 1000 players in queue simultaneously and proceeds to create the best possible 5 versus 5 matches with these players.

This process starts with loading a selected data table containing information about all players entering our matchmaking queue. Data table contains columns with information about each player, these columns are player ID, Accuracy, Reaction Time, Map Knowledge, Movement Skill and Awareness. After getting this data our system proceeds to calculate *Skill Rating* (see fig. 7.6 of each individual player and it fills up a structure containing playerID, Skill Rating and a Boolean which tells us whether the player already has a match or not, then it proceeds to add this information into an array of players prepared to be placed into matches.

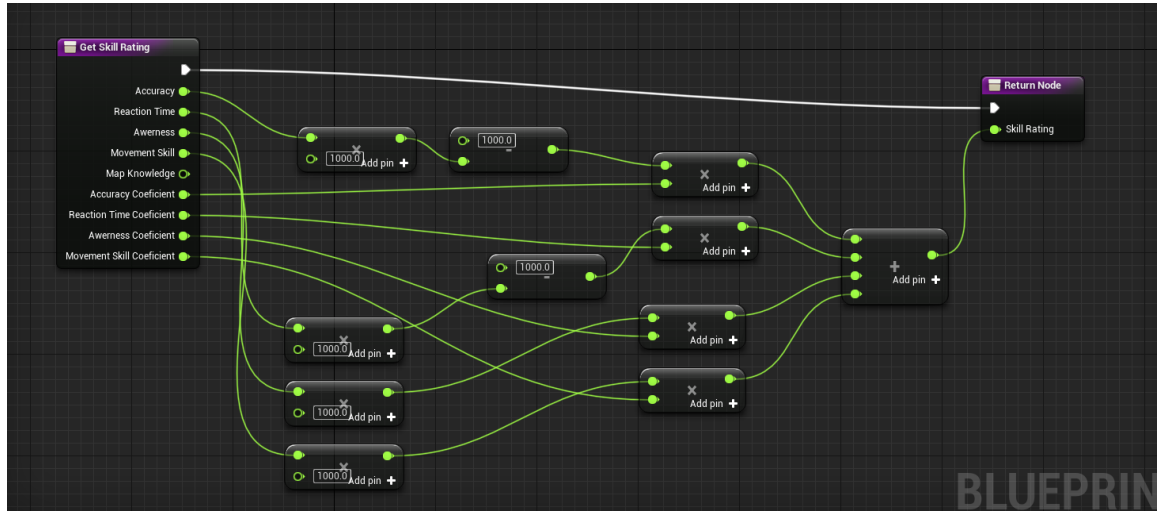


Figure 7.6: Function to calculate Skill Rating implemented in Blueprint according to the proposed calculation.

Once this array is filled a final matchmaking process begins. We are looping over this array while selecting player after player and finding a matching opponent for him. Matching process takes the first available player from our array, which hasn't been assigned to a match yet, takes his *Skill Rating* and starts to loop over the same array looking for an opponent of equal skill. To qualify as an opponent of equal skill the player has to be within the *Skill Rating Interval* and we have to filter out players who already have a match assigned to them, also we are filtering out the original player, for whose opponent we are looking,

from this search (see fig. 7.7). Once we have finished looping over our array, we have to check whether we have found an equal match. If we have done so, we can proceed with adding them to the match and looking to pair the rest of the players for current match. If we haven't found a matching opponent yet, we increase a *Skill Rating Interval* and repeat this process (see fig. 7.8). Increasing *Skill Rating Interval* happens until we have found the opponent, but gets reset back to the default once we start matching a new pair of players.

After filling up the current match with 10 players, we finish creating it by writing it down into array of *Ready Matches*. We also save the time it took to create every single match for analysis purpose. Once we have matched all players that entered our matchmaking we take our *Ready Matches* and save them into our game instance object and also write them into a file. By saving them in game instance object, we make sure that we can access our array of created matches once the simulation starts in the next step and also after every level restart.

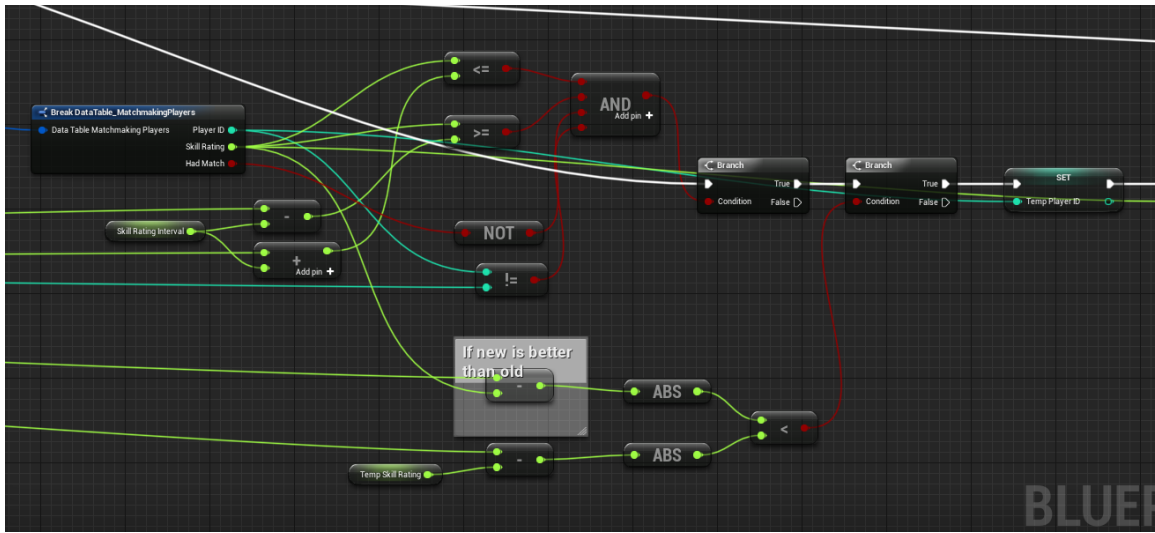


Figure 7.7: The finding a match part of the matchmaking process. Shows the simple Boolean logic and comparison of old best match with a new one.

7.3 Simulating

After the matchmaking phase a simulating phase begins. In this phase, we take an array of matches prepared by matchmaking from our game instance and simulate every single match as many times as we want. Since we have talked about AI implementation the last thing to mention here is the start of each match when we are setting up all of the bots.

At the start of the every simulated round we are looking at the array of matches which still have to be simulated. We take the match based on the *Current Game* counter which is set inside our game instance for safekeeping in case of a level restart, and get information about every player of that match from our Data Table containing player details. To finish set up, we have to get all our AI bot actors from the level and split them based on their *Team Number* parameter, set within the editor when adding AI actors to the scene, into two teams. Once we have everything ready, we start setting our AI actors' skills based on the data from Data Table. We had to pay a lot of attention not to mix and match players from team 1 with players from team 2 when setting their skills. During this we are also

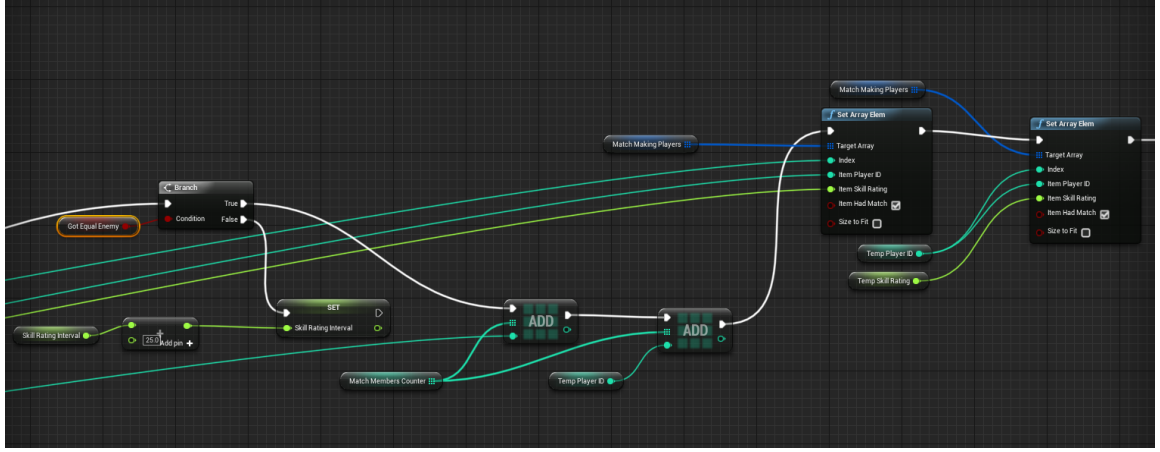


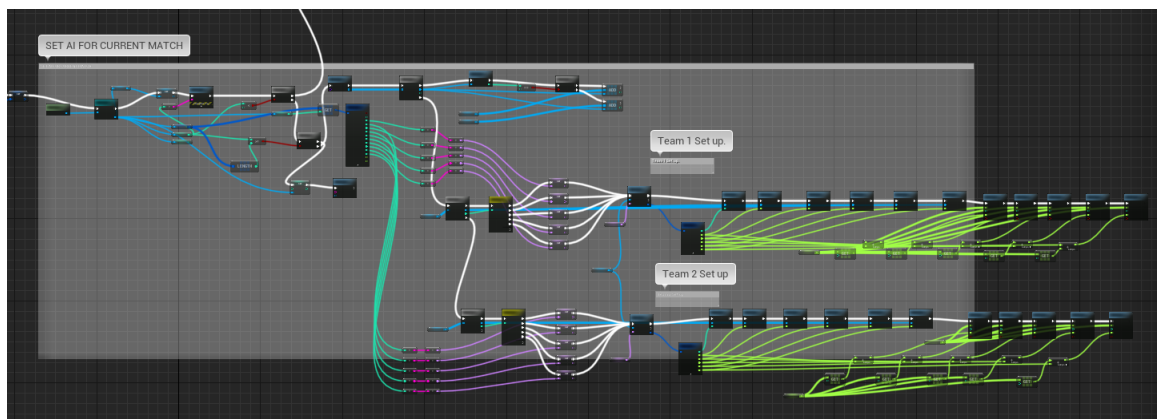
Figure 7.8: Part of the matchmaking process where we check whether we have found equal enemy and adding them to the match being currently created.

calculating average values of all skills for both teams and saving it for later use in simulation file output (see fig. 7.9).

After all of the AI actors are set a simulation can start. AI controllers take control of our AI bot actors on the scene and play out the match using the set up *Behavior Tree*. Each match can be played multiple times for data gathering. After the match has played a desired amount of rounds, this is set inside game instance's variable *Current Game Counter*, we increase a *Current Game* counter by 1 and move to the next match, while resetting *Current Game Counter*.

Check for the end of each round is ran at every simulation tick. Inside this check we simply compare the amount of alive players of both teams and in case of one team having zero players the round is over and winner is set. Rounds are not time limited, the only way to finish a round is for one team to kill the other. When each round is finished the result gets written into the output .csv file.

Simulation is over once we have reached the end of the array containing matches created by matchmaking and every match has been played out a desired amount of times set inside *Current Game Counter*. However, thanks to logging every round separately a simulation can be stopped at any moment without losing already accumulated data.



Chapter 8

Simulations and analysis

To properly analyze how effective is our matchmaking and what skills have higher influence on the results we first have to run a set of base simulations to compare effect of each skill on the match result and then compare different skills against each other and their effect on the results. After we have established some base line we progress to doing tests evaluating our Matchmaking efficiency by using the data from our base line simulations.

8.1 Base Line Tests

8.1.1 Solo Skill Tests

In the first set of simulations we want to compare effect of each skill on the match result. We do this by setting one team of AIs with a very high proficiency in the selected skill, while the other one with a very low proficiency. Rest of the skills are equal for both AI teams.

We can see on the results (see table 8.1) that *Accuracy* and *Reaction Time* have a clear decisive effect on the result of the games. Out of 20 games played with each setting in both cases the team with higher proficiency in the selected skill wins 100% of the games. And not only that but in *Accuracy* simulation we get 75% of the matches finish with more than half of the winning team alive, while in *Reaction Time* simulation this percentage goes up to the 80%. This two information clearly show us that these two have a massive impact on the match result and if not properly balanced they will create a very one sided matches.

On the contrary, *Movement Skill* and *Awareness* have a much different impact on the match results. Data shows us that on their own, these skills have quite a negative impact on the team's performance. Team with a very high *Awareness* has only 40% win rate while the team with a very high *Movement Skill* has even lower win rate of only 25%. However, despite a below 50% win rate in both cases, we can see that a score difference at the end of each game is very balanced in both cases only 50% of rounds end up with one team having more than half of it's players still alive. We will have to focus on this in our next tests to see if any other skill makes these two more effective or even less effective.

8.1.2 Versus Skill Tests

In the second set of our base line simulations we want to compare effect of skills against each other. This way we can find out which skill can counter which skill and therefor has a bigger impact on the match result. We are running 6 different simulations so every skill is

	Team 1 Main Skill	Team 2 Main Skill	Team 1 Wins	Team 2 Wins	Score Differences Count				
					1:0	2:0	3:0	4:0	5:0
Accuracy Test	0.1	0.9	20	0	0	5	4	9	2
Reaction Time Test	0.1	1.0	20	0	0	4	6	5	5
Awareness Test	0.6	0.0	8	12	5	5	5	4	1
Movement Skill Test	1.0	0.0	5	15	7	3	6	4	0

Table 8.1: Simulation results where one team has a very high proficiency in the chosen skill and the opponent has a very low proficiency in the same skill.

compared with the rest of the skills and 2 extra simulations to see how the most important skills from our *Solo Skill Tests*, *Accuracy* and *Reaction Time* affect the match result when combined with the less important skills *Movement Skill* and *Awareness*(see fig. 8.2).

Looking at this data we can safely assume that *Movement Skill* and *Awareness* have close to no effect on the match result when faced against opponents with superior *Accuracy* or *Reaction Time*. All of these simulations ended in a very one sided win rate of 100% in favor of the latter two skills. And also in terms of surviving players the winning team had in 95% more than half of it's players still alive and even more surprising result is that in 35% of the matches the winning team didn't lose a single player resulting in a 5:0 win.

Accuracy and *Reaction Time* seem to have a pretty balanced impact on the results, despite the fact, that team with better *Accuracy* wins 65% of the games. Critical information here is a score difference, we see that in this case the winning team wins by having more than 3 players alive only in 42% of the games and in only 7% it ends with a 5:0 result.

And in the final simple versus test between *Movement Skill* and *Awareness* we can see that neither of them provides it's team with a decisive advantage over the opponent. We get a 52% win rate for the team with better *Awareness*, but this is the closest we got to it being a 50 to 50 ratio. In this test however, we see that the most common final score is either 2:0 or 3:0 which can tell us that the winner depends more on the team which gets the first opening kills faster, or makes better decision in the game, rather than on the individual players.

By doing final two tests where we combine our least effective skills with most effective skills, we try to find out which combination gives us a better result so we can figure out a way to better balance our matchmaking. In the first case we can clearly see that a combination of *Awareness* and *Accuracy* wins over *Awareness* and *Reaction Time*. The difference is quite marginal, where the first combination wins 63% of our games. However, winning by having 3 or more players still alive is a case in only 47% of games which means that despite win percentage being so different, matches tend to be fairly close and in only 3% the match is won by a decisive 5:0 victory.

When we compare this with the second set of combined test, where we match players with good *Movement Skill* and *Accuracy* against players with good *Movement Skill* and *Reaction Time*, we can see that the first combination wins fewer matches by 10%. Since it includes *Accuracy*, that had a big impact in the previous test scenario, we can assume that *Movement Skill* has a lower effect on it than *Awareness*. Amount of wins by having more than 3 players alive is in this case slightly lower at 43%.

8.1.3 Changing Matchmaking Coefficients

By studying the previously mention data of our simulations, we can try to make our default matchmaking more efficient by changing it's coefficients. Previously in our default

	Team 1		Team 2				Score Differences				
	Skill 1	Skill 2	Skill 1	Skill 2	Team 1 Wins	Team 2 Wins	1:0	2:0	3:0	4:0	5:0
Accuracy Vs. Reaction	0.1	1	0.9	0.1	39	21	15	20	14	7	4
Accuracy Vs. Movement	0.1	0	0.9	1	60	0	1	3	12	21	23
Accuracy Vs. Awareness	0.1	0	0.9	0.6	60	0	0	2	6	28	24
Reaction vs. Movement	1	1	0.1	0	0	60	1	4	11	24	20
Reaction vs. Awareness	1	0.6	0.1	0	0	60	0	2	9	32	17
Movement vs. Awareness	0	0.6	1	0	29	31	9	21	20	7	3
Awareness+Reaction vs. Awareness+Accuracy	Aw=0.6 Re=0.1	0.9	Aw=0.6 Acc=0.1	1	22	38	11	21	14	12	2
Movement+Accuracy vs. Movement+Reaction	Mo=1 Acc=0.1	1	Mo=1 Re=0.1	0.9	27	33	15	19	7	15	4

Table 8.2: Results of Versus Skill Tests. Skill 1 is always the first mentioned and Skill 2 the second one, for example in the first simulation Accuracy is a Skill 1 and Reaction Time is the Skill 2.

matchmaking we used a default value of 0.5 for every skill coefficient. To try and balance our matchmaking better we want to adjust this for every skill coefficient separately.

We know that *Accuracy* and *Reaction Time* are the most impacting skills. From the first test we see that *Accuracy* is 30% more effective in changing the result in it's favour compared to *Reaction Time*. And both of them are 100% more effective in changing the result in their favour when faced against opponents more skilled in *Movement Skill* or *Awareness*. When players with good *Movement Skill* face *Awareness* the games seem to result in a good and balanced battle, so we should keep their coefficients fairly close to each other.

However, combined tests tell us to change this statement. *Awareness* seem to have much bigger impact on the match result when combined with *Accuracy* and since we have established, that *Accuracy* seem to have the biggest impact on it's own, *Awareness* has to become a little bit more important in our matchmaking than *Movement Skill*. By our data we can say that *Awareness* has a 36% bigger effect on the result when combined with *Accuracy*, compared to *Accuracy* and *Movement Skill* combination.

Our new proposed coefficients are then calculated as seen in equation 8.1.

$$\begin{aligned}
& MovementCoefficient = 0.2 \\
& AwarenessCoefficient = MovementCoefficient * 1.36 \\
& \Rightarrow AwarenessCoefficient = 0.272 \\
& ReactionTimeCoefficient = AwarenessCoefficient * 2 \\
& \Rightarrow ReactionTimeCoefficient = 0.544 \\
& AccuracyCoefficient = ReactionTimeCoefficient * 1.3 \\
& \Rightarrow AccuracyCoefficient = 0.7072
\end{aligned} \tag{8.1}$$

8.2 Matchmaking Tests

We are using our implemented environment and matchmaking to run three different tests. In all three of these tests 500 players enter our queue to get into 5 vs. 5 matches. We will be analyzing this data in this section to see whether our matchmaking makes any difference in terms of game results.

	Score Differences					Skill Rating Difference				Average Round Duration (seconds)
	1:0	2:0	3:0	4:0	5:0	Average	Median	Minimum	Maximum	
No Matchmaking	112	163	151	96	28	143.476	128.6	4.299927	363.1	43.490
Default Matchmaking	131	153	140	95	31	4.480004	1.150055	0	67.09998	48.695
Optimized Matchmaking	147	175	135	74	19	4.162268	1.28186	0.032349	79.97528	47.329

Table 8.3: Table showing accumulated result data from 550 rounds simulated in our environment.

Following three scenarios were simulated, in each case every created match has been played 11 times so with 500 players we created 50 matches which result in 550 different match rounds being simulated.

- No Matchmaking
- Matchmaking with all skill coefficients set to 0.5
- Matchmaking with skill coefficients calculated after our *Base Line Tests* ran in the previous section

One of the big worries for our matchmaking has been it's time effectiveness. However, we can see (see table 8.4), that despite having to properly match 500 players which are all in queue at the same time, the average time to do so with both *Default Matchmaking* and our *New Matchmaking* is below 15 milliseconds on average, per match. This means, that we managed to get all our players sorted into matches in below 1 second, and start simulating rounds. That is a very positive result.

Now looking at our round results (see table 8.3), we can see that in terms of score difference it doesn't seem like our matchmaking made a significant difference at first sight. However, looking at the data more carefully we can see that our *Optimized Matchmaking* lowers the amount of wins by a score difference greater than 4 by 6%, that is 33 rounds. This means, that our *Optimized Matchmaking* made games more balanced, and more based on the decision making in the each individual round, rather than purely on skill. Also looking at the game duration we can clearly see, that despite *No Matchmaking* score differences looking very similar to the other two, it's *Average Round Duration* was 5 seconds shorter, that is around 11%. This tells us, that even our *Default Matchmaking* made it possible for our games to not be a fast one sided battles where one team dominates over the opponent.

Where we can see a huge improvement is the *Skill Rating Difference* of individual players. When running simulation without any matchmaking we get an average skill rating difference between teams of over 143, while when running with either of our matchmaking systems we reduce this huge gap down to roughly 4 points, that is over 97% improvement. Also overall median is much lower for our matchmaking systems and the maximum skill difference out of all created matches is has been reduced by some 81% when compared to a *No Matchmaking* results.

Last thing we want to compare is the amount of created matches, which when ran in simulation, resulted in over 70% of rounds going in favour of one of the teams (see table 8.4). With *No Matchmaking* 27 of our matches resulted in a very one sided results which is 54% out of all matches, this is also very close to *Default Matchmaking* which has 50% of matches result in one team winning over 70% of rounds. However, in case of *Default Matchmaking* this number gets worse as we create more and more matches and therefor reduce a list of available players in queue. When we compare the amount of matches with one sided results once over 60% of players, that is 300 players, are already matched and so not available for our matchmaking anymore, more than 52% out of the one sided

	Matchmaking Time(milliseconds)	Matches Created	Matches In Which One Team Won				Matches In Which One Team Won In Last 20 Created Matches			
			11 rounds	10 rounds	9 rounds	8 rounds	11 rounds	10 rounds	9 rounds	8 rounds
No Matchmaking	0	50	5	6	8	8	2	1	3	4
Default Matchmaking	14.44	50	5	5	8	7	2	3	3	5
Optimized Matchmaking	13.34	50	1	3	5	8	1	2	1	4

Table 8.4: Table showing statistics about created matches and their results after simulation.

results are after this point of matchmaking for our *Default Matchmaking*, but only 37% for running with *No Matchmaking*. So our *Default Matchmaking* still provides us with a big improvement compared to *No Matchmaking*, because it gives us better matches with a lot of available players, slowly dropping in performance as queue of players decreases.

When we compare the previous too with our optimized *New Matchmaking*, we can see that it provides even bigger improvement over them both. Only 34% of created matches result in one sided games and that is 20% improvement over *No Matchmaking* and 16% improvement over *Default Matchmaking* (see fig. 8.1).

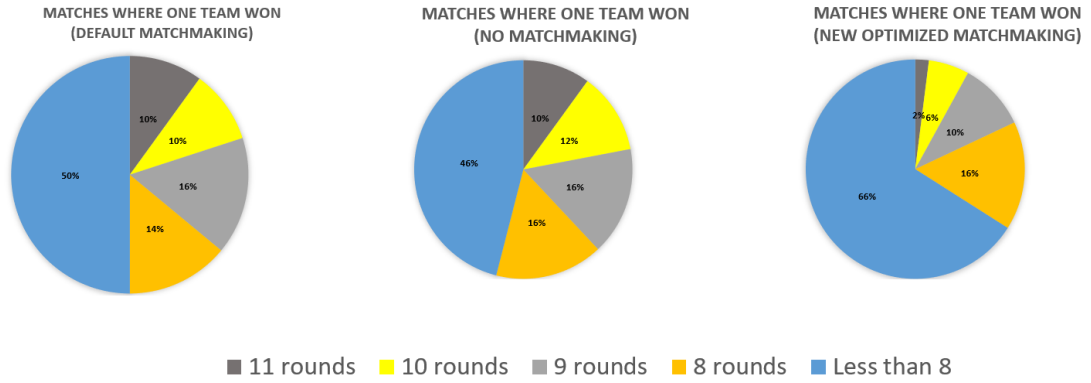


Figure 8.1: Chart visualization of difference in matches won by one team in more than 70% of rounds between all three matchmaking tests.

Chapter 9

Conclusion

The goal of this thesis was firstly to get familiar with game developing software Unreal Engine 4 developed by Epic Games, learn about it's features and find out it's possibilities to implement a simulation environment for a simple First Person Shooter game, secondly to design and implement full First Person Shooter simulation environment with working matchmaking system, which will allow us to further balance out the matchmaking based on the simulation results and thirdly to analyze the results of simulations ran in our simulation environment.

All three of these goals were successfully completed and the implemented application was properly tested. Simulation results show us successfully designing a working matchmaking system which creates balanced matches while also being very time effective within Unreal Engine 4.

With proper analysis of simulation results we managed to further improve fairness of created matches by adjusting important coefficients of the matchmaking system to even better evaluate the skill of players and this way we achieved an improvement of over 20% in some areas.

This work can be further developed in the future by implementing more skill factors which are to be considered inside our matchmaking as well as improving the complexity of matchmaking by using more advanced ways to calculate player's skill rating used in it. Another interesting way to further develop this work is to implement a different type of matchmaking system, for example Glicko 2, rather than Performance Based Matchmaking which is the case in our work.

Simulation environment can also be developed in the future for example by simulating matches in multiple levels designed within Unreal Engine 4 or by improving AI's behavior by adding more tasks and decisions. One of such expansions on our work could be implementation of multiple weapons and healing system for our AIs.

Bibliography

- [1] Bakkes, S. C.; Spronck, P. H.; van Lankveld, G.: Player behavioural modelling for video games. *Entertainment Computing*. vol. 3, no. 3. 2012: pp. 71 – 79. ISSN 1875-9521. doi:<https://doi.org/10.1016/j.entcom.2011.12.001>. games and AI. Retrieved from: <http://www.sciencedirect.com/science/article/pii/S1875952111000486>
- [2] Bungie: *Halo 2 Matchmaking Overview* . [Online; navštíveno 15.01.2019]. Retrieved from: <http://halo.bungie.net/stats/content.aspx?link=h2matchmaking>
- [3] Games, E.: *Blueprint Macro Library* . [Online; navštíveno 15.01.2019]. Retrieved from: <https://docs.unrealengine.com/en-US/Engine/Blueprints/UserGuide/Types/MacroLibrary>
- [4] Games, E.: *Blueprints Visual Scripting* . [Online; navštíveno 15.01.2019]. Retrieved from: <https://docs.unrealengine.com/en-US/Engine/Blueprints>
- [5] Games, E.: *Blueprints Visual Scripting* . [Online; navštíveno 15.01.2019]. Retrieved from: <https://docs.unrealengine.com/en-US/Engine/Blueprints/GettingStarted>
- [6] Games, E.: *Environment Query System* . [Online; navštíveno 10.05.2019]. Retrieved from: <https://docs.unrealengine.com/en-us/Engine/AI/EnvironmentQuerySystem>
- [7] Games, E.: *Interface Blueprint* . [Online; navštíveno 15.01.2019]. Retrieved from: <https://docs.unrealengine.com/en-US/Engine/Blueprints/UserGuide/Types/Interface>
- [8] Games, E.: *Introduction to C++ Programming in UE4* . [Online; navštíveno 15.01.2019]. Retrieved from: <https://docs.unrealengine.com/en-US/Programming/Introduction>
- [9] Games, E.: *Level Blueprint* . [Online; navštíveno 15.01.2019]. Retrieved from: <https://docs.unrealengine.com/en-US/Engine/Blueprints/UserGuide/Types/LevelBlueprint>
- [10] Games, E.: *Setting Up Visual Studio for Unreal Engine* . [Online; navštíveno 15.01.2019]. Retrieved from: <https://docs.unrealengine.com/en-US/Programming/Development/VisualStudioSetup>

- [11] Hubík, B. J.: Player Skill Rating for Games with Random Matchmaking. 2016.
Retrieved from: <https://is.cuni.cz/webapps/zzp/detail/165714/>
- [12] Ishwarya, T. A.; Appala Naidu, R.; Meghana, K.; et al.: A modern approach to design and integrate conceptual methods in video games with artificial intelligence. *Materials Today: Proceedings*. vol. 4, no. 8. 2017: pp. 9100–9106. ISSN 2214-7853.
- [13] Kosinski, R. J.: A Literature Review on Reaction Time. 2009.
Retrieved from: <https://web.archive.org/web/20100611222125/http://biae.clemson.edu/bpc/bp/Lab/110/reaction.htm>
- [14] Wikipedia, t. f. e.: *Elo rating system* . [Online; navštíveno 15.01.2019].
Retrieved from: https://en.wikipedia.org/wiki/Elo_rating_system
- [15] Wikipedia, t. f. e.: *Glicko rating system* . [Online; navštíveno 15.01.2019].
Retrieved from: https://en.wikipedia.org/wiki/Glicko_rating_system
- [16] Wikipedia, t. f. e.: *Unreal Engine* . [Online; navštíveno 15.01.2019].
Retrieved from: https://en.wikipedia.org/wiki/Unreal_Engine
- [17] Wikipedia, t. f. e.: *Zero-sum games* . [Online; navštíveno 15.01.2019].
Retrieved from: https://en.wikipedia.org/wiki/Zero-sum_game

Appendix A

Contents of the DVD

Attached DVD contains:

- this document in PDF format
- L^AT_EXsource of this document
- directory with source code
- directory with Windows 64bit executable application