Level Design Parameterization & Automated Playtesting in 3D Action Games

Bachelor's Thesis

Video game design and development

Surname: Doctor Pedrosa Name: Marc

Pla: 2014

Director: Pons López, Joan Josep

Index

Summary

In the current video game map-making era, automated workflows have begun to arise as industry standards. Beyond automated map generation, there exists a technique which combines human design with AI feedback: automation testing.

Unbeknownst to many, an automated playtesting can be carried out with a heavy focus on level design. That is to say, a video game level can be numerically designed so that an AI navigates said level to then extract feedback via objective evaluations.

These evaluations must be based on a level design "parameterization" model, in which the designer defines which variables are involved in the map's fitness as a proposal and, at the same time, how these variables affect each other.

Beyond map navigation, the AI is meant to recreate human behaviour, by implementing some sort of player type, with different motivations and skill levels so that each AI variation prioritizes certain actions above others.

Figure 0.1: Unity logo

Unity Game Simulation, an extension of the video game engine *Unity*, the tool of choice, offers up to 500 simulation hours, which translates to the possibility of running each instance with different parameters to recreate a sample of players.

Key Words
Parameterization, game, level, design, Unity, automated, playtesting, AI, simulation

Tables Index

Figures Index

Glossary

1. **Weighted graph**: representation with nodes that have numeric values and connections between them which can also have numeric values.

2. **Shooter**: video game subgenre where a player engages in combat by using firearms (can include use of grenades, armor, etc)

3. **A-RPG** (also “Action-RPG”): video game subgenre that combines action and RPG elements

4. **Beat ‘em up**: video game subgenre that emphasizes hand to hand combat with large numbers of surrounding opponents.

5. **Dungeon Crawler**: video game subgenre with a labyrinth-like map and elements such as treasures, traps, etc.

6. **Platformer**: video game subgenre where a great part of the challenge relies on moving between disconnected points.

7. **RLD** (Rational Level Design): paradigm where each element (obstacles, distances, etc) within a video game level is quantifiable.

8. **Player persona**: set of psychological nuances that represents a human-like profile.

9. **Unity**, **Unreal**: free to use, popular game engines in the world.

10. **Engine**: set of pre-made tools designed for game creation and similars.

11. **Package**: in the context of Unity, an extension to the core engine.

12. **Triple A** (also “AAA”): video games classification according to high standards in terms of budget, publishing, quality, innovative technologies involved, etc.

13. **Convolutional Neural Network (CNN)**: deep neural network applied to images.

14. **Genetic algorithm**: algorithm inspired by natural selection; focused on crossing variables of instances with the best fitness function

15. **Fitness function**: function used to the determine the best performing instance in a genetic algorithm

16. **Bartle type**: *"Classification of video game players according to their preferred actions within the game"* (https://en.wikipedia.org/wiki/Bartle_taxonomy_of_player_types)

17. **Playthrough**: from start to finish, in the context of a game, level or prototype

18. **Pathfinding**; spatial navigation of an agent through a grid of nodes

19. **GOAP (Goal Oriented Action Planning):** "*Goal oriented action planning is an artificial intelligence system for agents that allows them to plan a sequence of actions to satisfy a particular goal"* (https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793)

20. **Machine Learning:** *"Machine learning is the science of getting computers to act without being explicitly programmed"* https://www.coursera.org/learn/machine-learning

21. **A*:** *"A* is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency"* (https://en.wikipedia.org/wiki/A*_search_algorithm)

22. **Deathmatch:** game mode where the primary objective is to kill enemies on other teams

23. **Prefab:** in Unity, gameobject stored in the project's content that permits changing itself so as to update all instances in the scenes.

24. **Pickup:** within a video game, object present in space that can be collected

25. **Asset:** within Unity, any file inside the project that can be viewed via the file explorer

1. Introduction

1.1. Motivation

From the very moment I discovered video games, I have been inevitably attracted to their maps and environments.

Those maps, as I would discover on my own while creating levels, are crafted by employing vast amounts of design guidelines and all sorts of tricks.

As a map creator, I thus concluded that, in order to speed up the process and better the result, I could apply those techniques by means of an automated workflow.

Said workflow would permit level designers analyze their work by using AI capable of translating those design guidelines into map navigation and feedback.

Finally, I decided to delimit my thesis inside the 3D action genre provided that therein lie a series of subgenres with substantial map differences that can share level design variables/principles, albeit with some singularities.

Figure 1.1: CSGO's (a tactical shooter) popular map, "Mirage"

1.2. Problem Formulation

Traditionally, to test and iterate their work, level designers had one tool at their disposal: human playtesting.

This concept, however, quickly became tied with a set of hindrances:

- Humans give highly subjective and likely contradictory feedback.
- Human playtesting is slow in comparison to computation.
- Humans require accommodation.

Therefore, video game companies have shyly started to combine human playtesting with AI automation.

Concretely, some studios rely on automated tests to evaluate isolated mechanics in their games, although there are no guidelines for automated playesting applied to maps.

That is to say, *"level design parameterization"*, also referred to as the phenomena where a set of variables with different magnitudes are attributed to a level, lacks a reference model in the global map making scene and thus an automated playtesting nowadays cannot emanate from said model.

1.3. General Objectives

- Study parameterization in different 3D action subgenres. Design a weighted graph[1] model for the main subgenres
  - *Shooter*[2]
  - *A-RPG*[3]
  - *Beat 'em up*[4]
  - *Dungeon Crawler*[5]
  - *Platformer*[6]

  The weighted graph would contain variables such as "cover", with a value ranging from 0 to 10. In this particular case, a *Shooter* would have a much higher value than a *Beat 'em up*

- Create an automated playesting environment and gather level data via AI.
  In order to achieve this ecosystem, the automated playtesting must be based in one of the above subgenres. The following elements are therefore needed:
  - A navigable map based on *RLD*[7]

  - Intelligent AI that can analyze the level and compare it with the model.
  - A simulation tool that permits several playthroughs with input variations.
  - Enemies, coins, and/or similar perks that can entail challenge and/or objectives.

- Compare the automated playtesting with human playtesting. Within the same conditions, humans will play the level and give their corresponding feedback. Analyze the advantages and disadvantages of both in regards to giving useful feedback that can convert into tangible level improvements.

1.4. Specific Objectives

- Gather or create an RLD based map. Either way, said map must be based on RLD and must also have apparent imbalances that should be pointed out during both playtestings.
- Implement different *player personas*[8] for the AI. These variations will be used as input parameters to provoke different results and to mimic real-life player motivations, i.e. “kill all enemies”, “gather all coins”, etc.

1.5. Project Scope

As previously mentioned, the theoretical section will encompass a total of 5 subgenres and, in order to define a model for each, it is my intention to expand on map case studies.

As for the prototype, the tool of choice is *Unity*[9] and its *Unity Game Simulation package*[10], which provides for up to 500 free simulation hours.

The resulting tool (and even the theoretical parameterization models) could be used by level designers around the globe, although the prototype is meant to be case-specific, since it has map design requirements.

2. State of the Art

Automated playtesting is not an industry standard per se. Concretely, this technique is usually limited to simple puzzle games. Moreover, other techniques such as procedural level generation are more established.

Commercial video games, even *triple A*[12] games, are beginning to introduce automated playtesting into their pipelines.

However, there are three main domains that provide valuable information on the topic:

- Commercial video games.
- Video game engines with automated playtesting plugins.
- Various papers that propose techniques so as to expand on level analysis and automated playtesting.

As far as level parameterization is concerned, there are no studies that define a model that represents level design by means of pure numbers. In relation to the topic, nonetheless, there is one concept that is needed for a level to be measured accordingly: *rational level design (RLD) [6]*

2.1 Unity

Unity provides for a framework, *Unity Game Simulation* [1], that offers up to 500 cloud simulation (using their own servers) hours. In their proposal, the application in context is executed a certain number of times.

The simulation is composed by metrics, parameters and test cases:

"Metrics are the results of a playthrough that are important to the balance or health of the game. Test cases are the different scenarios that you'd like to measure within a game. Finally, parameters are the configurations that can change your game, thus directly impacting your metrics." (Willis Kennedy, 2020, p.1) [1].

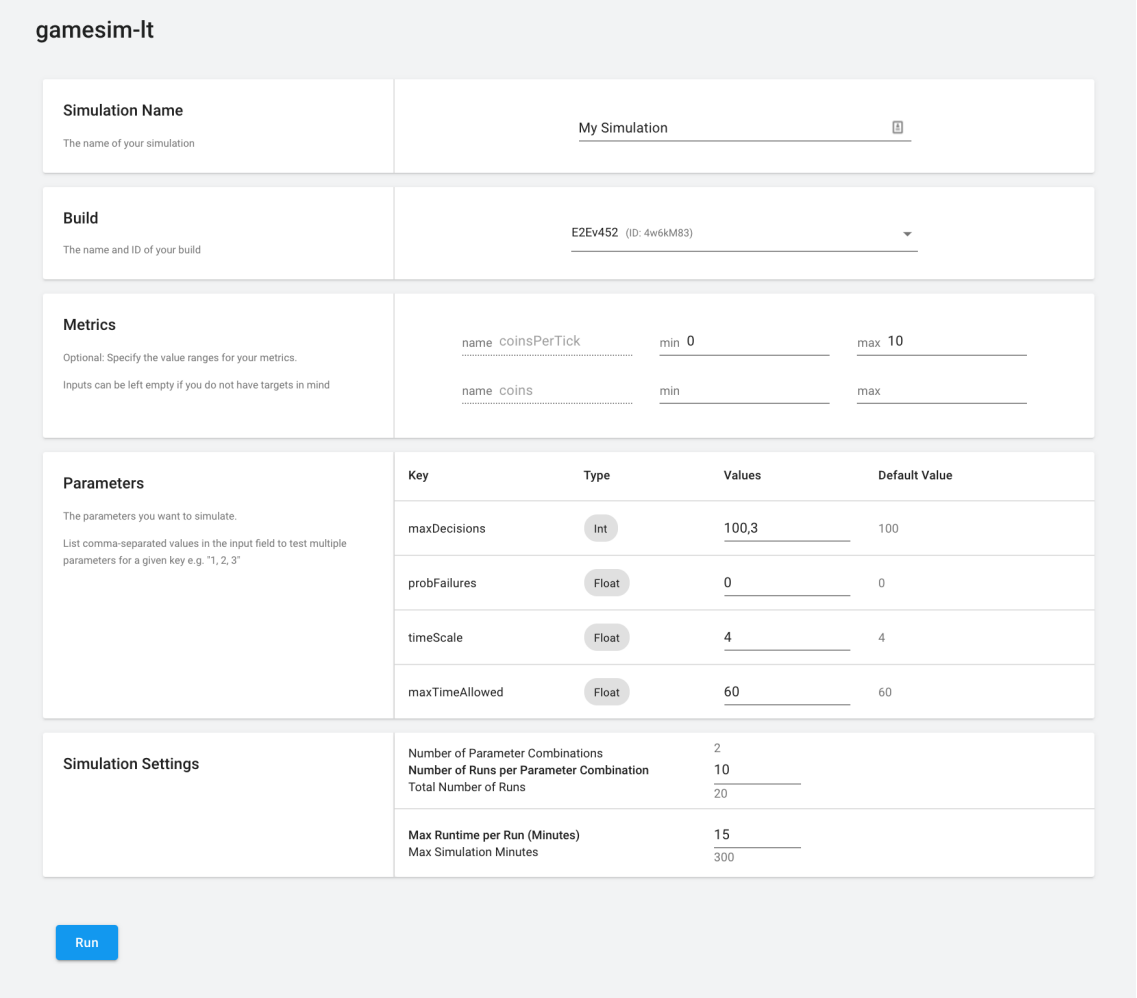

Figure 2.1: Simulation details and settings in *Unity Game Simulation*

2.1.1 Death Carnival

Death Carnival, developed by *Furyion*, *"is a fast-paced top-down shooter with extreme weapons and online multiplayer mayhem" (Furyon Team, year not specified, p.1) [2].*

As mentioned in the article, *Death Carnival* utilized *Unity Game Simulation* to balance the weapons in the game. Concretely, three base weapons were used as test cases, weapon variables as parameters and survivability was the chosen metric.

According to the development team, by using this package they were able to achieve the equivalent of 165 million human playthroughs in limited time.

2.1.2 Rogue Racers

Rogue Racers, developed by *iLLOGIKA*, *"is a mobile freemium racer pitting heroes against each other as they dash through richly illustrated landscapes".*

In the case of *Rogue Racers*, each character's cards (powers or abilities), chosen from a pool, needed to be balanced.

Instead of having to manually balance each possible combination, they were able to run 25000 games in four hours.

2.2 Unreal

Unity's competitor, *Unreal*, also incorporates an automation system.

Test Type	Description
Unit	API level verification tests. See **TimespanTest.cpp** or **DateTimeTest.cpp** for examples of these.
Feature	System-level tests that verify such things as PIE, in-game stats, and changing resolution. See **EditorAutomationTests.cpp** or **EngineAutomationTests.cpp** for examples of these.
Smoke	Smoke tests are just considered a speed promise by the implementer. They are intended to be fast so they can run *everytime* the Editor, game, or commandlet starts. They are also selected by default in the UI . **WARNING** All Smoke tests are intended to complete within 1 second. Only mark Unit Tests or fast Feature Tests as Smoke Tests.
Content Stress	More thorough testing of a particular system to avoid crashes, such as loading all maps or loading and compiling all Blueprints. See **EditorAutomationTests.cpp** or **EngineAutomationTests.cpp** for examples of these.
Screenshot Comparison	This enables your QA testing to quickly compare screenshots to identify potential rendering issues between versions or builds.

Figure 2.2: *Unreal*'s Automation System test types

Unlike *Unity Game Simulation*, Unreal's proposal does not provide for options such as running the test a certain amount of times per simulation or altering parameters for each simulation instance.

2.2.1 Sea Of Thieves

The triple A[12] video game *Sea Of Thieves* (about sailing and pirates) expanded *Unreal*'s automation system to fit their needs.

Although they did not provide any example of what a “map test” constitutes, one can still notice that map tests are relatively less employed than actor tests (“actor” meaning entities such as a character).

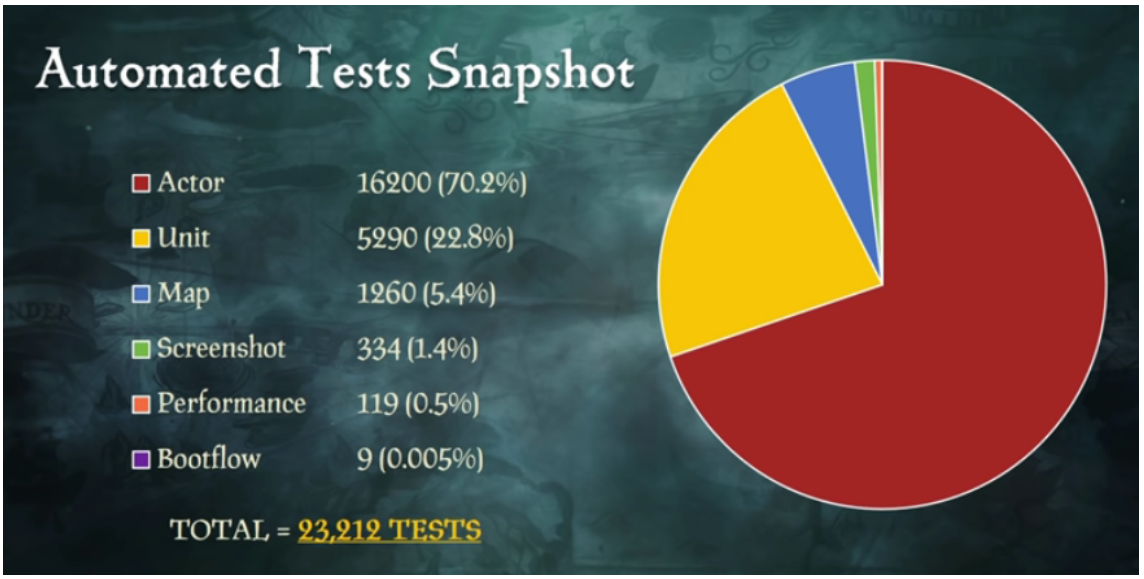

Figure 2.3: Proportion of tests according to type

Actor tests are commonly performed by mimicking the player’s input.

Figure 2.4: Sample test where the boat’s wheel is steered

Another aspect about automated testing, and in video games in general, is latency. They expand on how latency can alter a test result, since the time it takes to communicate with the simulation server can result in the logic being executed in bigger time intervals.

This phenomenon can lead to, for instance, a cannonball being represented before and after hitting a ship, but not while inside the ship’s area, so collision may not be detected.

2.3 Other Games

Beyond the mainstream video game engines, other games use their own solutions.

2.3.1 Riot Games

Similarly enough to *Unity*, *Riot Game*'s star game's (*League Of Legends*) automation system is suited for, but not exclusively, altering input parameters for each test instance in order to, for instance, test each champion's' (character) abilities.

Figure 2.5: Sample test in *League Of Legends*

Report for AllChampsAllAbilities 191310, run November 25, 2015 at 11:41 AM

Errors 0 | Successes 0 | Show All

Run Result	PASS
Client Code/Content ID	
Server Code/Content ID	
Runtime	Run time unavailable
Jenkins Job	

All Tests:

AllChampsAllAbilities : KogMaw (Base)

Get Champion Setup - KogMaw (Base)	
Status	PASS
Start Time	2015-11-25T19:39:54Z
End Time	2015-11-25T19:40:03Z
Machine Name	JMERR1WD1
Details	No comment

Champion - KogMaw (Base) - Cast Spell - slot 0 - Icathian Surprise - A bit after dying, Kog'Maw explodes and damages enemies	
Status	PASS
Start Time	2015-11-25T19:40:03Z
End Time	2015-11-25T19:40:09Z
Machine Name	JMERR1WD1
Details	No comment

Figure 2.6: test result interface in *League Of Legends*

2.4 Level Design Parameterization & Other Approaches

Although popular video game *engines* and several video games in general have begun to incorporate automation testing into their pipelines, there is a lack of real-world examples in the field of level design parameterization.

Nonetheless, there is not a lack of research. Concretely, there exist papers that expand on innovative techniques more targeted towards level design.

2.4.1 Learning the Patterns of Balance in a Multi-Player Shooter Game [3]

This first experiment proposes the use of a *Convolutional Neural Network (CNN)*[13] with the purpose of evaluating the balance in a level with two teams, depending on the weapons used in each team.

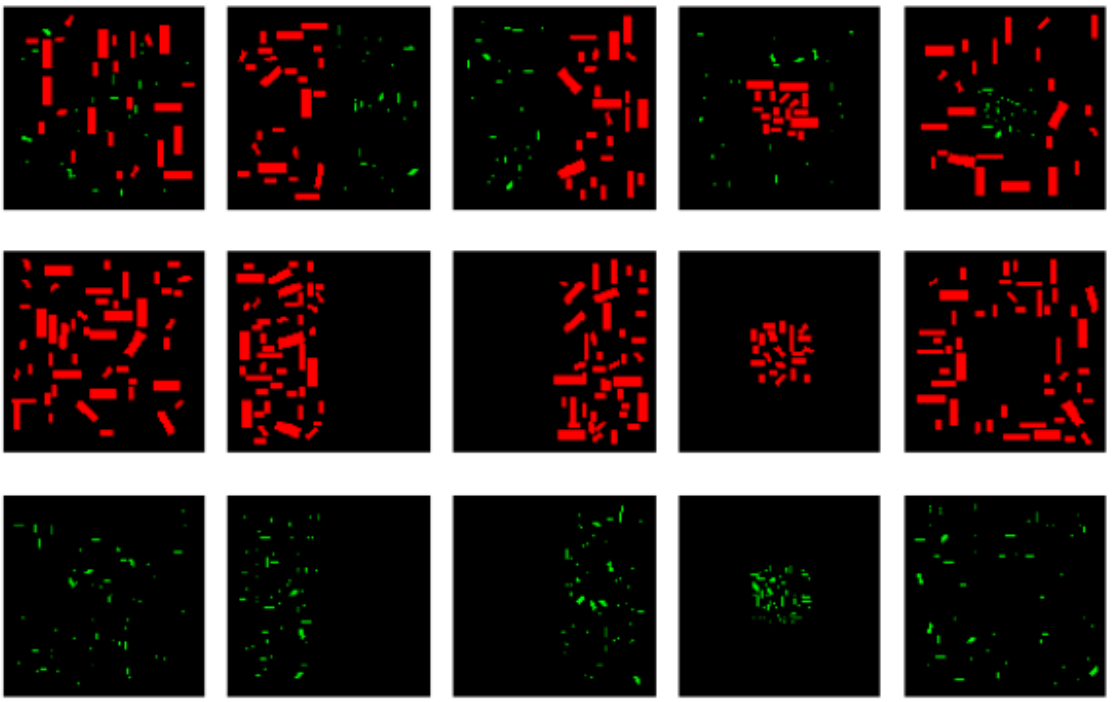

Figure 2.7: Sample level top-down images

To achieve that goal, levels are represented as the image above, from a top-down view where 3 types of geometry configure the map: wall, high wall and empty space. The *CNN* then receives these images as input, along with weapon type by team, to calculate balance and conclude if the level is one sided or balanced.

2.4.2 Evolving Interesting Maps for a First Person Shooter [4]

This paper combines the concepts of *genetic algorithm*[14] and *fitness function*[15].

Firstly, the experiment employs 4 different generators to create a set of maps. To evaluate these maps, the *fitness function*, which is represented by the sum of:

- Average fighting time per player
- Surface of all empty tiles

Only then, a *genetic algorithm* is applied, for 50 generations, in the case of each generator. That is to say, each of the generator's levels were evolved 49 times by tweaking parameters, selecting the ones with the best fitness score and then crossing their variables.

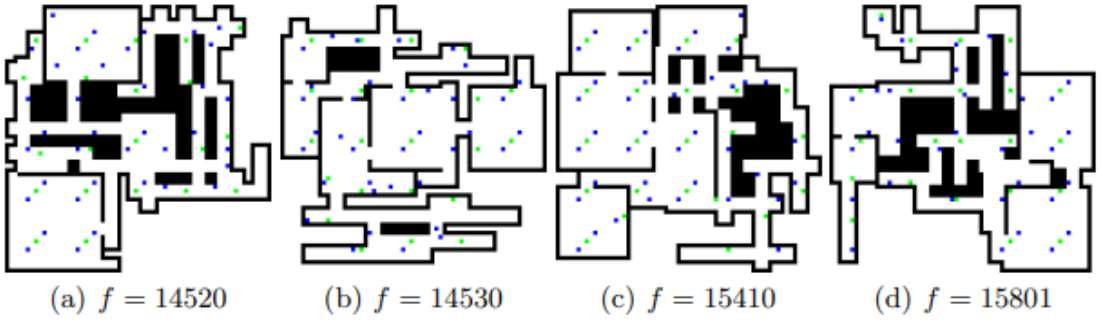

Figure 2.8: 50th generation maps pertaining to one generator

2.4.3 Automated Playtesting with Procedural Personas through MCTS with Evolved Heuristics [5]

In order to achieve an automated playtesting, the video game is in need of an entity, a character, that can represent humans in regards to their primal needs and psychology.

In the study, a total of 4 "*player personas"* are generated, which, in practical terms, means that for each of them there is a function that prioritizes certain actions/goals according to their *Bartle type*[16]: killing monsters, collecting coins…

Each player has a set of altered variables and the experiment employs the same technique as the study above, genetic algorithms. although applied to players instead of levels.

Evolving personas can simulate real-life imbalances between different skill levels and mimics more accurately a sample of gamers.

2.4.4 Rational Level Design (RLD)

The concept of Rational Level Design stems from that of Rational Game Design, which overall refers to a design that can be measured and therefore scaled and evaluated numerically.

The example provided in [6] proposes a series of platforms that are numerically displaced and their numbers altered in order to increase difficulty in a controlled manner.

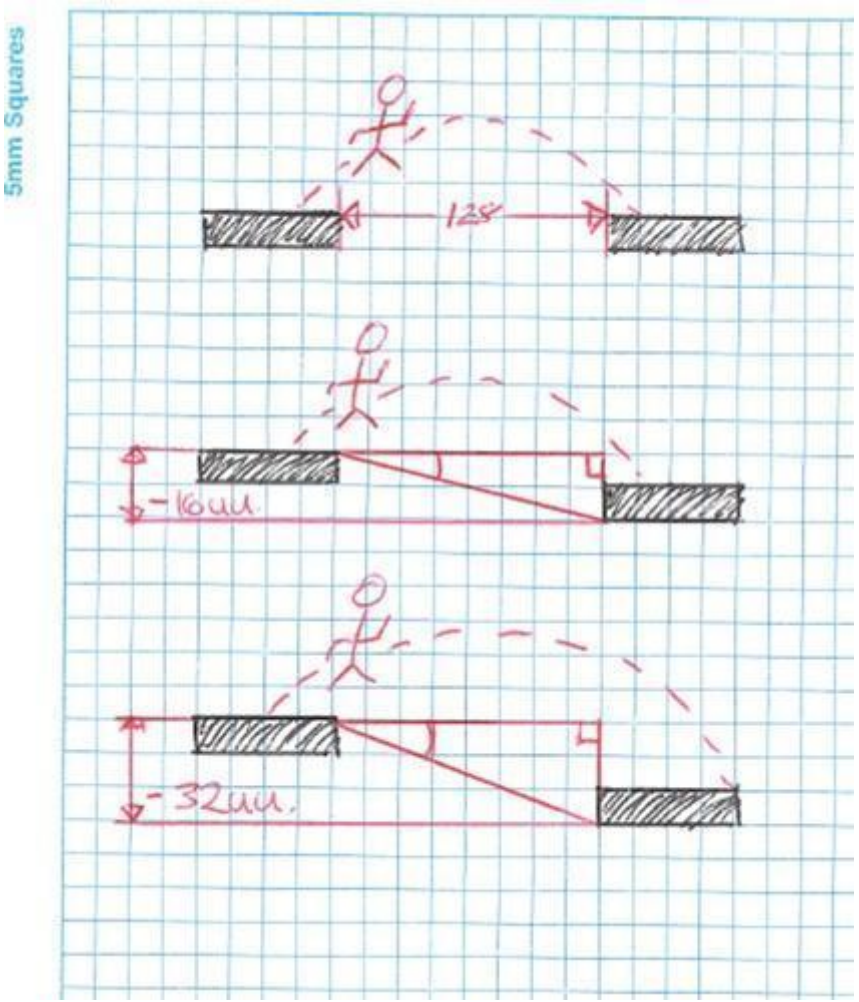

Figure 2.9: As shown in [6], an example of a numerically designed level section

In terms of automated playtestings, this approach to level design facilitates the AI's evaluations while navigating the level.

2.4.5 Level Design Theory

Although not expressed in terms of concrete parameterization, ample research has been done in the field of level design, with a popular forum specially dedicated to the shooter genre, *Mapcore*.

Concretely, a level design website is cited, http://level-design.org/, which contains a section for relevant level design elements.

Amongst others, there are selected few that could potentially be evaluated in an automated playtesting, or at least be more suited:

- **Balance**:

  From creating team vs team environments with equal possibilities for both (although with some imparities that offer interest to each side) to the art of tweaking events in the level to keep the match competitive (*rubberbanding*), balance is a key element in level design across the genres.

Figure 2.10: map balance in the competitive online shooter *Valorant*

In regards to automated playtesting, balance is one of the most, if not the most, common element of interest to be tested.

As explained in 2.4.1, Daniel Karavolos *et al.* trained in 2017 map generators that would create different levels. The results of this experiment were expressed both in terms of balance between each team (win rate), also depending on weapons used.

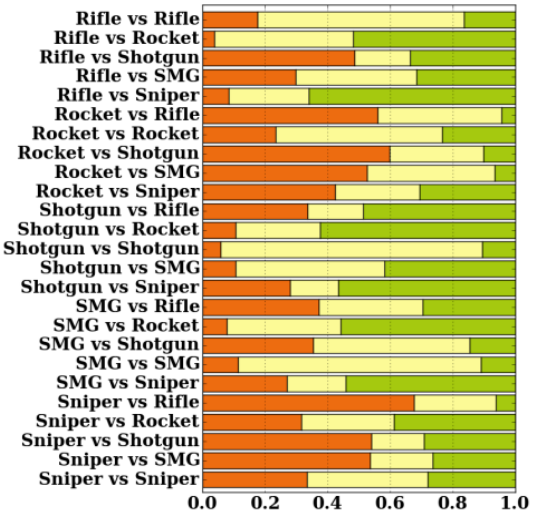

Figure 2.11 (D. Karavolos *et al.,* 2017, p.5)

- **Challenge, Fun, Flow**:

  Within any game level, the element of challenge arises. In an automated playtesting, the AI is meant to replicate a player with a certain skill level.

  Basic human psychology tells us that challenge and fun are directly related and thus a flow theory can be generated, where “flow” represents a satisfactory mind state where the player is neither overwhelmed nor underwhelmed by the gameplay.

  A possible method for evaluating the AI’s flow state is to take into consideration the amount of surrounding enemies, time between combats, skill level of the machine, *etc.*

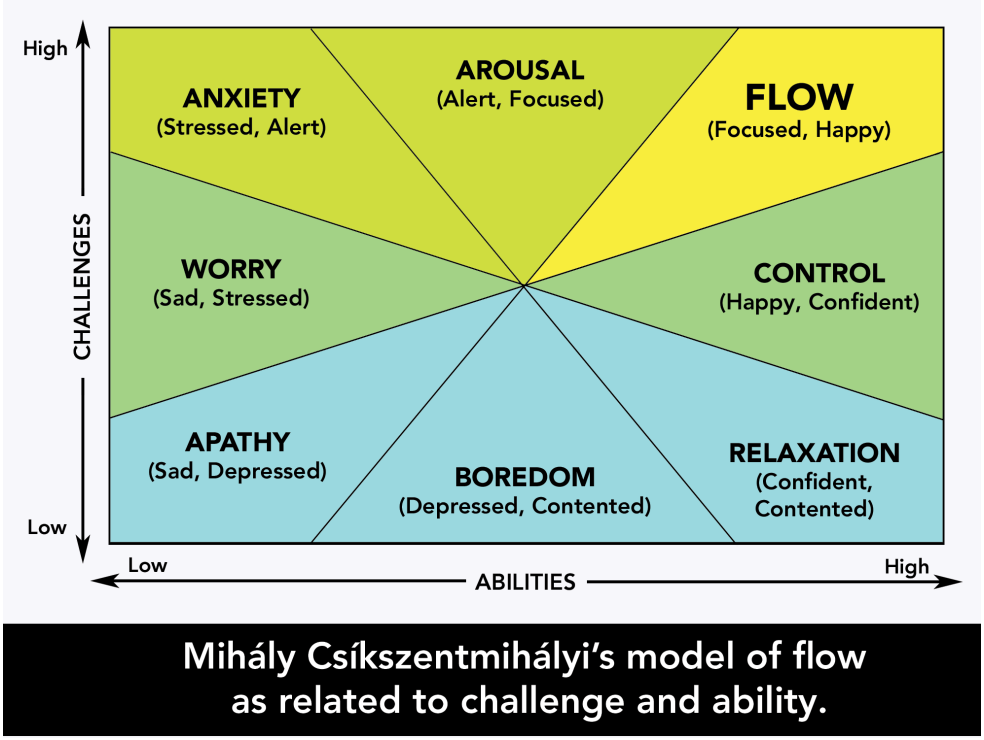

Figure 2.12 Game flowchart

- **Covers, Obstacles, Negative Space**:

  Negative space is a term that has been proposed as the absence of geometry or obstacles.

  Within each action genre, there is a different amount of negative space that suggests more or less exposed combat situations. Not only is negative space used in combination of cover and obstacles, but also as a way to aid navigation and avoid over-crowded levels, both physically and visually.

  In regards to cover, normally shooters offer the most out of any 3D action genre.

  Overall, in an automated playtesting negative space and obstacles can be evaluated via raycasts, or lines from the character to the world, for instance a certain amount in different directions, to check how much time they intersect with the level and at which distance.

- **Enemy placement**:

  Since the behaviour of an enemy is more of a combat design decision, there is a remaining task for level designers to take on: enemy placement.

There is no better example of enemy placement than the *Half Life* saga.

Half-life is a series of sci-fi FPS games with a narrative and episodic nature. The game's maps are designed firstly on a per-room basis. That is to say, every room has a series of items, enemies and possible interactions.

Figure 2.13: Original *Half-Life* barnacle

In the above image, a *barnacle*, pertaining to the first game of the series back in 1998, is spawned on the roof and has the ability to catch humans, some enemies and items with an elongated sticky tongue.

Within a typical room, barnacles would be placed so the player can force enemies into their tongues without the need of wasting ammo.

More often than not, barnacles served a double-purpose and did also present a challenge to the player while moving forward, usually by offering two ways: one slower, more secure, without entering in contact with them, and the other with a barnacle sequence.

In the saga, enemy placement and introduction is directly related to the player's current weapon:

"Players love having exactly the right weapon for a particular job, but having the wrong weapon out to deal with the next problem - just for a moment - helps elevate the sense of tension and excitement" (Valve, unspecified date, Enemies and item placement) [12]

Indeed, a new type of enemy or even an enemy group with an innovative mix on a tough spot will force the player to evaluate their current weapon, and said though process can be translated to an automated playtesting, where the AI could potentially "try out" different weapons and strategies based on the upcoming enemy spawns.

- **Pacing**:

  In the field of level design, pacing is often linked to the time between events. Within the action genre, said events could be enemy encounters, map destruction, weapon upgrades, etc.

  Normally, and specially in combat-focused games (*hAck 'n slash),* pacing is subsequently related to *time to kill (ttk)*, or the time delta between kills.

  Calculating the *ttk* in a simple level is a suitable test within an automated playtesting and said *ttk* variable is a popular metric of choice to determine a level's fitness, or score (a similar metric is explained in 2.4.2).

- **Playstyles, Strategy, Choice**:

  Most video games rely on *Bartle's taxonomy*, a solution that divides players in 4 basic groups, according to which is their ultimate goal or desire.

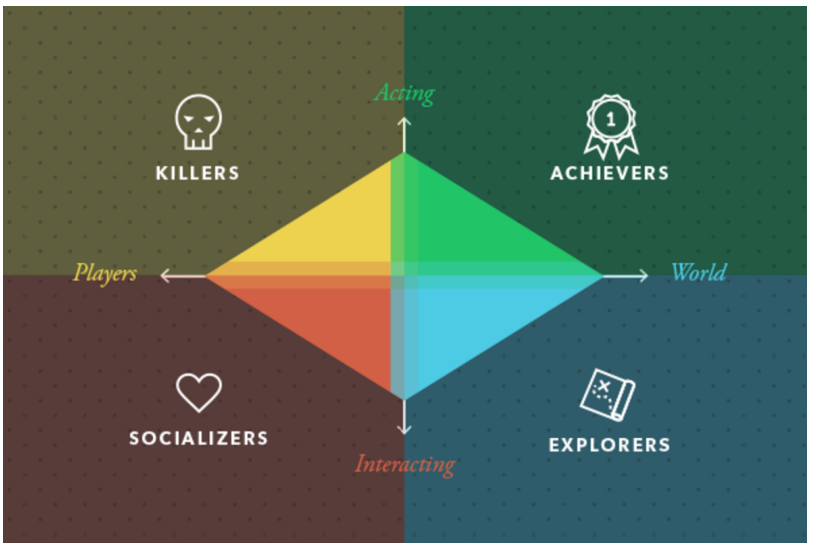

Figure 2.14: Bartle taxonomy

In practise, 3D action games address bartle types so that in every area or room there are many choices for the player to make.

Within the automated playtesting realm, every room can be coded as having strategies represented by finite routes of actions that the AI can take, for instance:

- Room 1 has got a health item and 2 enemies surrounding a treasure.
- A “killer” type of player would directly run into the enemies and maybe pick the treasure by convenience once they are killed.
- A collector would lay out a plan to separate the enemies from the treasure and then sneak in to take it and quickly exit the area.
- An explorer or completionist would first gather the health item if needed to then kill the enemies with time and collect the loot.

- **Vantage Point, Choke Point**:

Vantage and choke points can be traced back to the medieval era where archers inside a castle launched arrows from high, well-protected towers (vantage point) to intruders that had to enter the fortress via a single narrow, exposed entry, which led to many casualties (choke point).

The same architectural paradigm applies nowadays in level design.

As for shooter games, *CSGO’s* map *inferno* provides us with a choke point example.

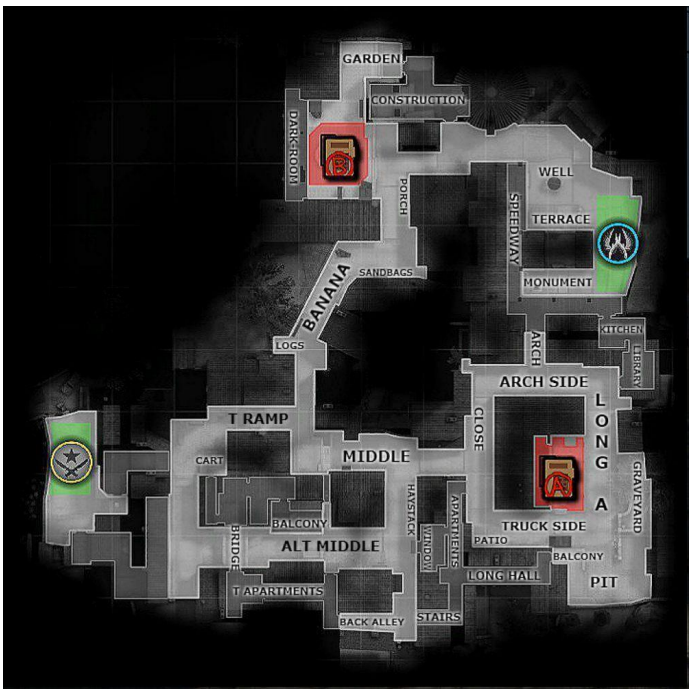

Figure 2.15: *Inferno* map layout, viewed from the top

Within the above image, an area called “banana”, a choke point, stands out, since it is the only immediate option for the attacking team to reach bombsite

B. Said area is cramped and represents a choke point for both teams, since the defending team does not have a clear vantage point either to defend.

Another example, which belongs to the A-RPG genre, resides in *Dark Souls 3's* MAP *Anor Londo.* The area is based on gothic architecture and there are multiple high flying buttresses that the player can navigate on top of.

While navigating those buttresses, a handful of archers shoot arrows from ledges above. The composition of this area, thus, represents an example of both a choke point (the lone, uphill buttress) and a vantage point (the ledge from where archers shoot downwards, with the possibility of making the player fall to death).

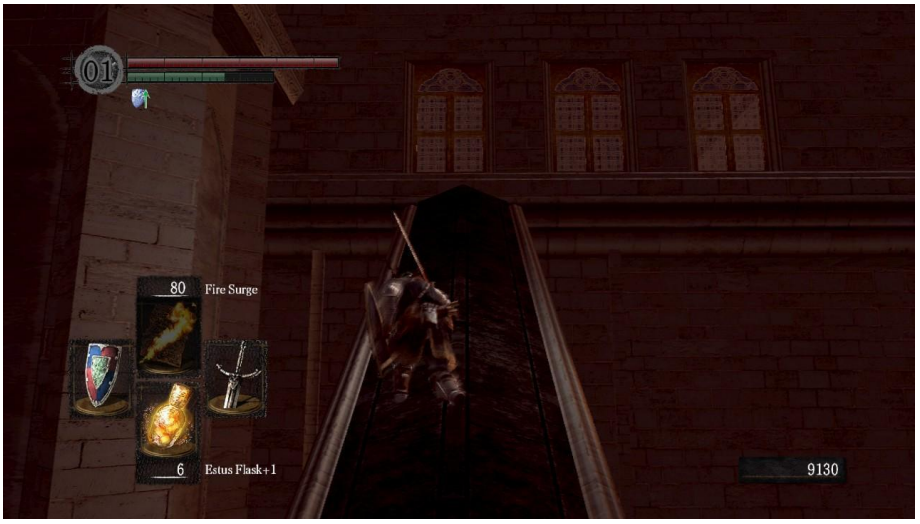

Figure 2.16: *Anor Londo's* flying buttresses and ledges, in first person

Figure 2.17: *Anor Londo's* flying buttresses and ledges, in third person

In an automated playtesting, one can create visualization maps that represent where the player killed an enemy and the opposite, which would give hints towards the existence of a vantage point in a more concentrated kill area.

3. Project Management

3.1 Tasks Management

3.1.1 Timeline: GANTT

The project is divided in theory and a prototype, the latter being scheduled after the first rubric deadline.

As for the prototype, the main pipeline consists of first producing a navigable map with variants, then coding the AI and lastly performing automated tests.

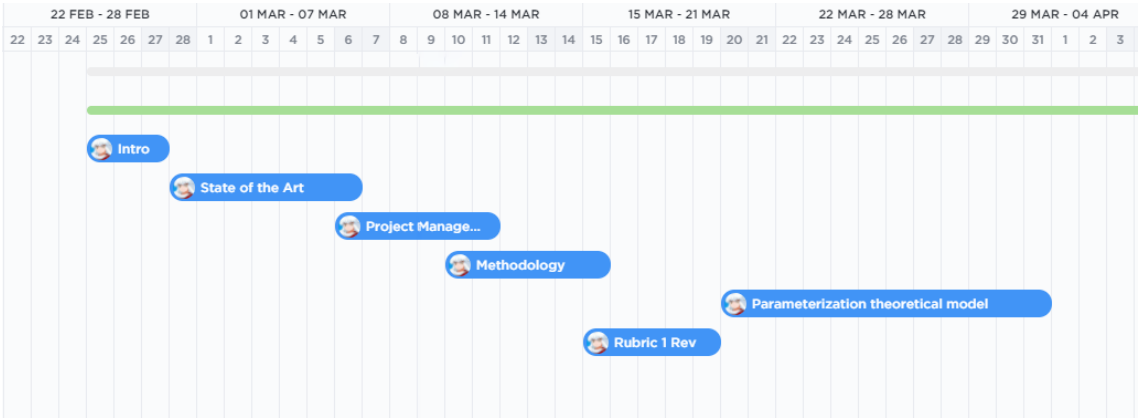

Figure 3.1: Rubric 1 Documentation

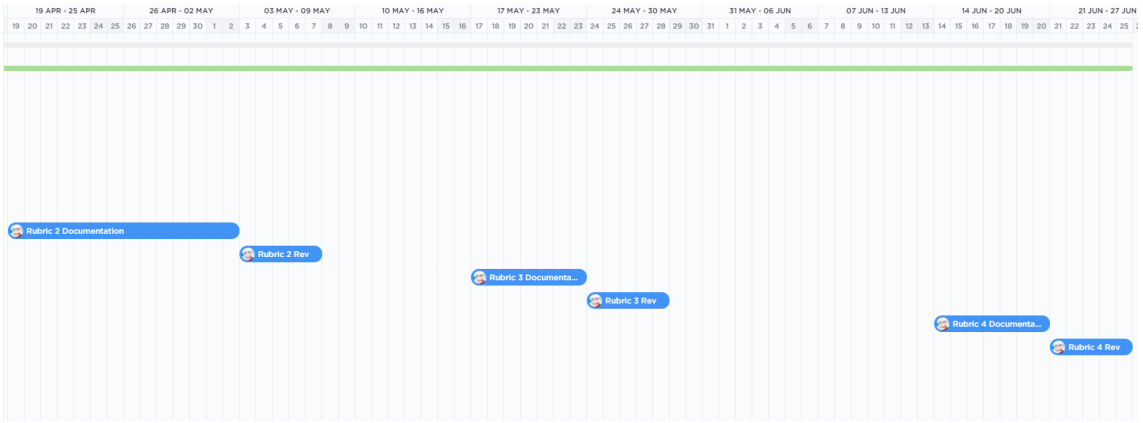

Figure 3.2: Rubric 2,3 & 4 Documentation

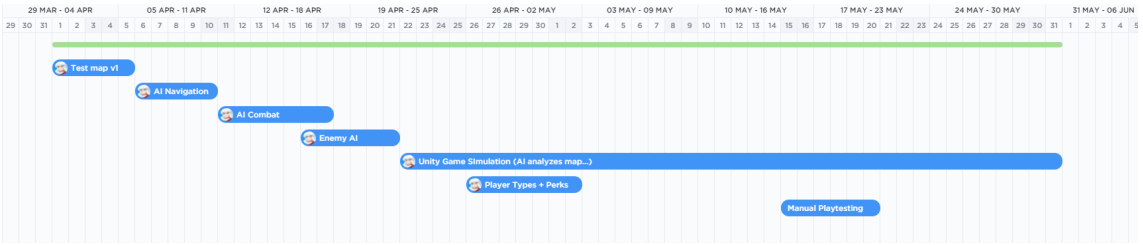

Figure 3.3: Prototype

3.1.2 Task Dashboard: ClickUp

ClickUp is a workplace suited both for personal and enterprise environments. Some of the functionalities include: multiple task views (list, board, GANTT…), task dependencies, time tracking, *etc*.

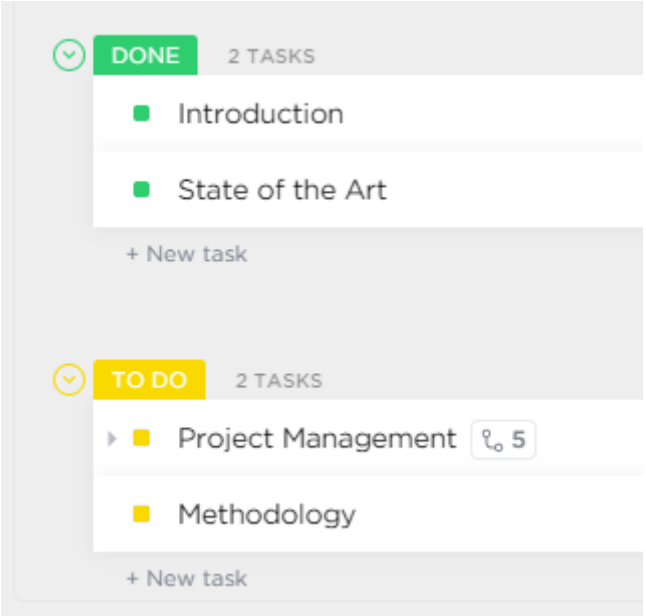

Figure 3.4: *ClickUp* tasks “List” view

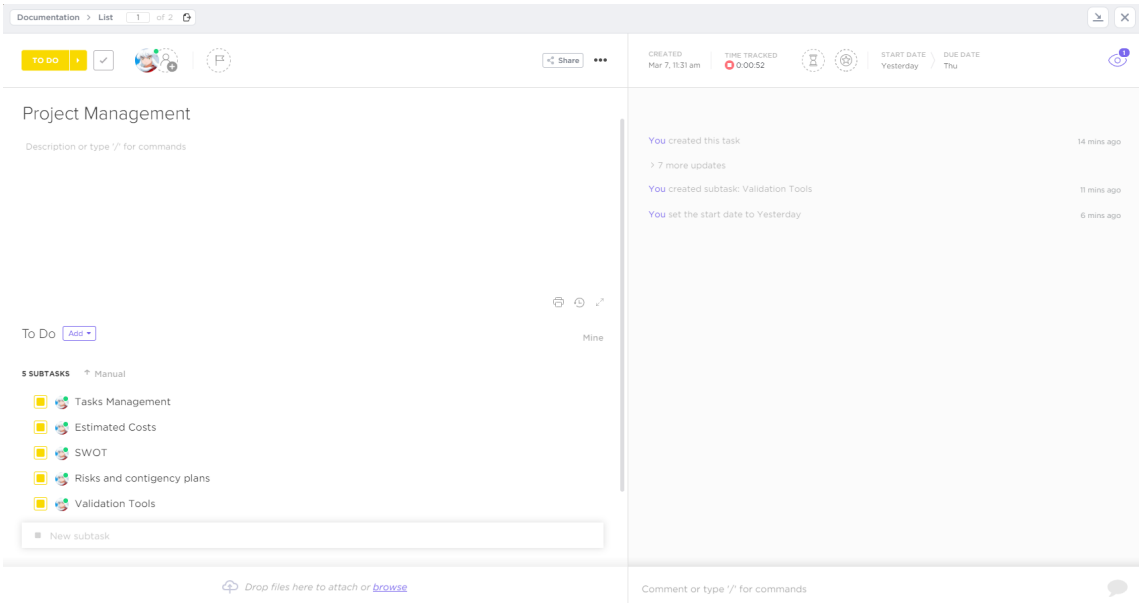

Figure 3.5: *ClickUp* task view

3.1.3 Cloud Repository: Github

Github is a free-to-use cloud repository and source control tool that offers a workspace of ideally up to 5GB of total files.

The site provides a “.gitignore” file with different options to choose from, in order not to push unnecessary files linked to your application of choice (for instance, *Unity*).

Amongst other utilities, *Github* offers: working in different branches with the ability to merge code between them, an issue dashboard to post problems and feature requests with tags, licensing, *etc*.

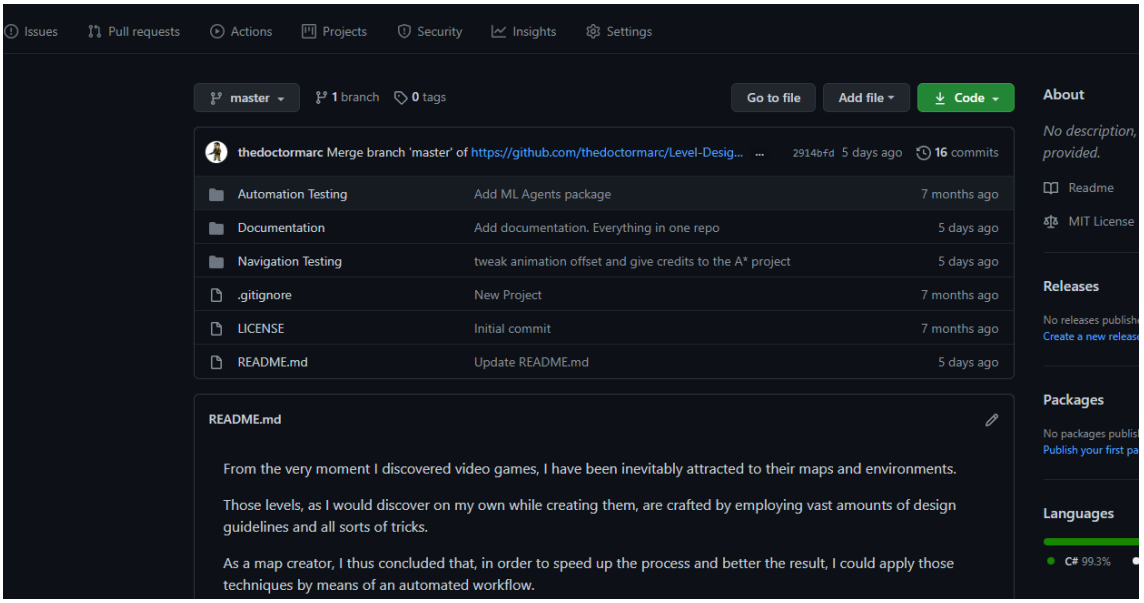

Figure 3.6: *Github* project view

3.2 Validation Tools

Firstly, in order to validate the theoretical level design parameterization models, the AI character that navigates the level will make evaluations based on said model, according to the map genre.

Secondly, so as to incite more interesting findings, the AI will be tested in contrasting map variants, that will present obvious biases in map design.

Only when the first iteration of evaluations is complete, the obtained feedback will be used to manually modify the levels accordingly.

Thereafter, many iterations will follow, until the levels' design has a palpably higher quality.

Lastly, manual playtesting is going to be employed as another validation tool. That is to say, the maps are going to be shared within a game development community so that human feedback can be compared with the machine and thus conclusions are drawn.

3.3 SWOT

This analysis has been generated prior to the development phase:

Strengths	**Weaknesses**

- Level design knowledge and avid 3D action gamer - Experience with Unity3D and AI behaviour	- Lack of experience with Unity Game Simulation - Unipersonal team that has to coordinate many areas of expertise: design, programming, analytics...
Opportunities	**Threats**
- No generic automated playtesting tool targeted to level design - Area of little to no research that lacks design models	- Triple AAA video games do not usually employ automated playtesting on a large level design basis - Automated playtesting may be too case specific for a generic tool: each studio creates their own solutions

Table 1.1: SWOT

3.4 Risks and Contingency plans

This analysis has been generated prior to the development phase:

Risk	Plan
3D Pathfinding for the AI in Unity has no proper native solution (NavMesh is poor)	Use an external plugin, such as: https://arongranberg.com/astar/

Credibility of the automated playtesting analysis depends on player skill	Develop an AI with different skill levels or evolve the base skill
Gameplay and analytics code interference, which could cause logic problems, lack of readability, performance issues, etc	Implement the observer pattern
Difficulty to **compare automated and manual playtesting** feedback	Propose a form that lets manual playtesters address their feedback in a numerical way similar to the AI

Table 1.2: Risks and contingency plans

3.5 Estimated Costs

The table below emulates an environment where different specialists co-develop the tool: a designer, due to the level design theoretical model, a programmer, an analyst and finally testers that would evaluate both the tool and serve as playtesters for the manual playtesting comparison experiment.

Section	Cost	N Units	Unit Value ($) x Month	N Months	Total ($)

Personel					
	Programmer	1	2701,91	3	8105,73
	Designer	1	2876,66	2	5753,32
	Analyst	1	2840,69	2	5681,38
	Tester	3	2021,39	1	6064,17
Software					
	Unity (Pro)	2	150	3	900
	Visual Studio (Business)	2	45	3	270
	ClickUp (Unlimited)	3	9	3	81
	Github (Team)	2	4	3	24
Office Equipment					
	Desktop computer	6	400	1	2400
	Computer accessories	6	600	1	3600
	Desk, chair...	6	250	1	1500
Office Maintenance					
	Rental	1	300	4	1200
	Gas	1	50	4	200
	Electricity	1	50	4	200
				Final Cost ($)	**35979,6**

Table 1.3: Estimated Costs

3.6 Rubric 2 Modifications

The development has, for the most part, been progressing according to the GANTT timeline.

One misconception cited on the diagram is that “Enemy AI” is a task to be developed independently. In practice, I have developed the AI so that each agent is equally

important, without having an AI that mimics a human player and other AIs that act as enemies.

This adjustment is more suitable for the prototype, since the AI will fight each other in a totally symmetrical manner, that is to say in a *deathmatch*[22] environment.

Another deviation is the adjournment of the theoretical parameterization part. During this period, my main focus has been programming the AI, which acts as a base for the whole process.

Since I have a clear understanding of what the actions of a shooter AI can be, the following parameterization and evaluation of the level can be done afterwards, once the AIs are able to recreate a shooter environment.

As for the estimated costs, the tool of choice for the AI's behaviour (*see section 5.1.3)*, *Node Canvas,* demands a one time payment of **37,50$** to be used by **1** programmer, which at the same time increases the total cost from 35979,6$ to **36017.1$**.

3.7 Rubric 3 Modifications

After finishing the AI's development, manual playtesting was set to be completed already, and a total of 2 playtestings have been carried out, although I have decided to continue with them at the same time as the AI automated playtestings, so that they feed back to each other.

Stemming from some suggestions in said playtestings, I have scheduled new tasks that will enhance future playtestings:

- *Risk-reward*: every shooter map has areas that precede objectives, normally referred to as "choke points". Alter the level and evaluate it accordingly.
- *Height map:* in terms of analytics, as will be explained in the methodology section, some maps are generated after a simulation. A remaining map is the height map, which translates to data in relation to kills and deaths relative to the height of the player's position. Furthermore, the level will be tweaked to maximize this effect.

Not mentioned previously, the manual playtesting has been materialized by means of a new task: a human controller that can perform the same actions as the AI.

4. Methodology

Since the proposal encompasses design, programming and analytics, the pipeline will be divided as follows:

4.1 Research Phase

Firstly, one video game is going to be selected for the five genres to be studied.

Each game's levels represent case studies which contribute to the ultimate goal of formulating a level design variables graph, similar to a weighted graph, as explained in section 1.3.

These weighted graphs will be translated to functions. For the selected genre, three map variants will be developed and thus the function is going to determine their score.

4.2 Development Phase

Once the theory is laid out, all tasks will orbit around creating an environment suitable for an automated playtesting. These tasks focus mainly on developing an AI that mimics a human player, to later explore the concept of different player types and skill levels.

Apart from the AI, there are plans for map perks and items such as coins or weapons.

4.3 Analytics Phase

Only when the prototype has enough logic to go through a *playthrough*[17] will *Unity Game Simulation* come into play.

That is to say, automated tests are scheduled to start before the gameplay development is fully complete, and are meant to prolong until then.

In this phase, the AI character will have input variables designed for the simulations, which will evaluate the player's results within the level, upon completion.

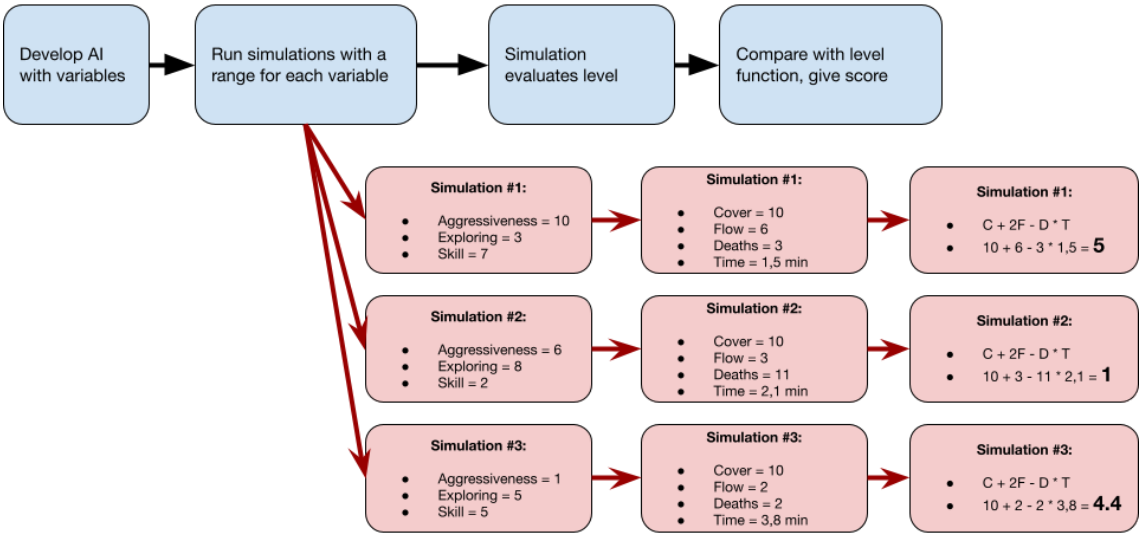

Figure 4.1: Automation pipeline

4.4 Manual Playtesting Phase

Once there is enough data on the previous simulations, a manual playtesting is scheduled in order to compare human vs machine feedback.

Playtesters will have access to the same levels and will be asked to play a certain amount of times each.

Once the playthroughs are completed, they will be handed a *Google Forms* to express their feedback so as to be compared with the AI's.

Figure 4.2 Google Forms logo

4.5 Agile Methodology

The thesis will follow a more *Kanban* approach rather than the usual, more advanced *Scrum*, because of the following reasons:

- The project is developed by an unipersonal team, which almost nullifies the value of Scrum sprints, reviews, etc
- Between all phases there are 4 roles involved, a fact that complicates task workload estimation, among others.

4.6 Rubric 2 Modifications

On the one hand, the analytics and manual playtesting phases are to be followed as planned.

On the other hand, the research phase has been reformulated. First of all, the order of research and development has been swapped, as mentioned in *section 3.6.* On top of that, whilst it is my intention to generate a theoretical model that suits the shooter genre, I have decided to drop the other genre's models, after having performed an analysis and comparative between them, since the parameterization for shooters is independent and does not benefit from the study of the other genres.

Following, I will expand on a notion that has not been modified *per se*, although it has been mentioned vaguely and is in need of clarification: the origin and concept behind the base level to be used in the playtesting.

In relation to the Validation Tools, I came up with the idea of creating a replica of an existing, functional level in order to iterate said level and prove that even existing commercial video game levels can be iterated and improved via an automated playtesting to appear more suitable within a certain environment.

The level of choice is *Dust2,* from the shooter *CSGO (Counter-Strike Global Offensive)*.

(see https://counterstrike.fandom.com/wiki/Dust_II for reference).

4.7 Rubric 3 Modifications

As for the AI, some scripts have been internally merged into one, so that each agent can perform all actions: collect points, help teammates, etc, depending on their priority.

Previously described, the Analytics Phases and the Manual Playtesting Phase were established as separate. Nonetheless, since manual playtestings can generate feedback regarding the analytics themselves, I have joined both phases thematically.

In terms of analytics, the field has evolved into 3 categories:

a. **Manual playtesting**: 1 human player combats amidst bots.
b. **Local AI automated playtesting**: a local simulation at a time, to store maps such as the heatmap.
c. **Cloud AI automated playtesting**: multiple server-side simulations via *Unity Game Simulation*, storing kills/deaths per team and other metrics.

Figure 4.3 AI heatmap

The combination of these procedures is meant to incite the strength of both methodologies: manual playtesting and automated playtesting, to collect unique feedback for the level.

5. Development

5.1 Development Tools

5.1.1 Unity

In the field of video game development (and subsequent tools), *Unity* and *Unreal* are the main options to consider:

- **Unreal Engine 4**: among the engine's attractives, *Unreal* offers a *visual scripting* system, where code can be encapsulated in nodes with inputs and outputs.

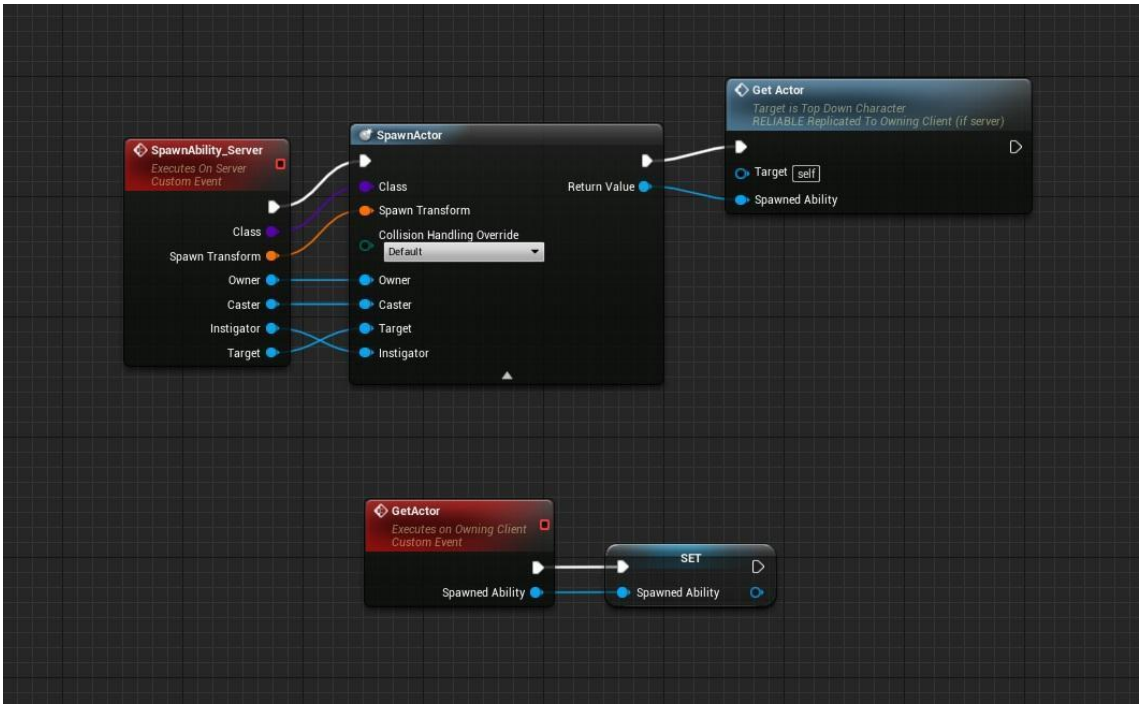

Figure 5.1: Sample Unreal Engine 4 blueprint

Other advantages, such as accurate lightning, post-production and extensive physics systems, are beyond this project's scope.

- **Unity:** in contrast to unreal, *Unity* is an engine that at first glance has less native functionality. Nonetheless, there is support for as many or even more extensions, via the *Package Manager.*

Concretely, *Unity* offers multiple solutions to 3D *pathfinding*[18], AI programming and game simulation, which have representation in the project.

Ultimately, it is because of this ample catalog that *Unity* is the engine of choice.

5.1.2 Unity Game Simulation

In regards to simulation within *Unity*, there exists a native proposal through the *Automation Testing* module which involves running one simulation at a time and checking minimal tests within the code.

A more suitable alternative is *Unity Game Simulation*, as explained in 2.1.0, which constitutes the tool of choice.

5.1.3 Node Canvas

Some of the most popular approaches to AI programming in videogames are:

GOAP*[20]*: As an example, the chosen proposal, *Node Canvas*, is a visual scripting *GOAP* tool that lets the user organize the totality of an AI's behaviours inside a single unit of representation, the so called *Behaviour Tree.*

The nodes within a *Behaviour Tree* offer a great flexibility: ability to check conditions, to execute code, to run for a certain amount of time…

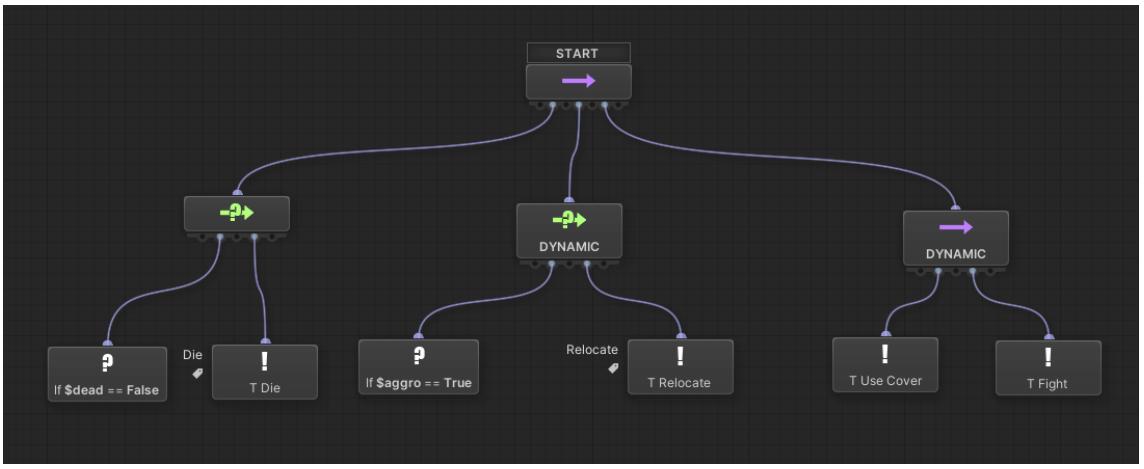

Figure 5.2: sample Node Canvas Behaviour Tree

ML Agents: A more recent approach to AI is *machine learning*[20]. Within this context, *ML Agents*, an extension, integrates the concept in *Unity*, so that an agent runs

simulations with input variables and slowly tries to improve its performance at accomplishing a task.

Even though the approach by itself is noteworthy, the fact that, in order to be able to rely on a competent AI, one has to set up a whole separate environment and train this AI for a unknown amount of time and simulations, takes away from the project's own simulation ambitions (which are related to map playtesting, not AI improving).

Finite State Machine: A FSM executes a portion of the code or another depending on the AI's current state. Although once a practical approach in game development, FSMs have slowly but surely become deprecated and GOAP trees are the spiritual successors, due to their extensive flexibility, node operations and readability.

5.1.4 A* Pathfinding Project

In regards to map navigation, I have selected a couple of approaches that are contrasting and at the same time crowded with functionalities.

NavMesh: A common resource for any 3D project developed inside *Unity* is the *NavMesh* workflow.

The term *NavMesh* accounts for "Navigation Mesh", a term that refers to a complex shape that avoids objects interpreted as obstacles. The agent, therefore, moves between the shape's points.

This *NavMesh* supports slopes, jumps between different meshes, etc.

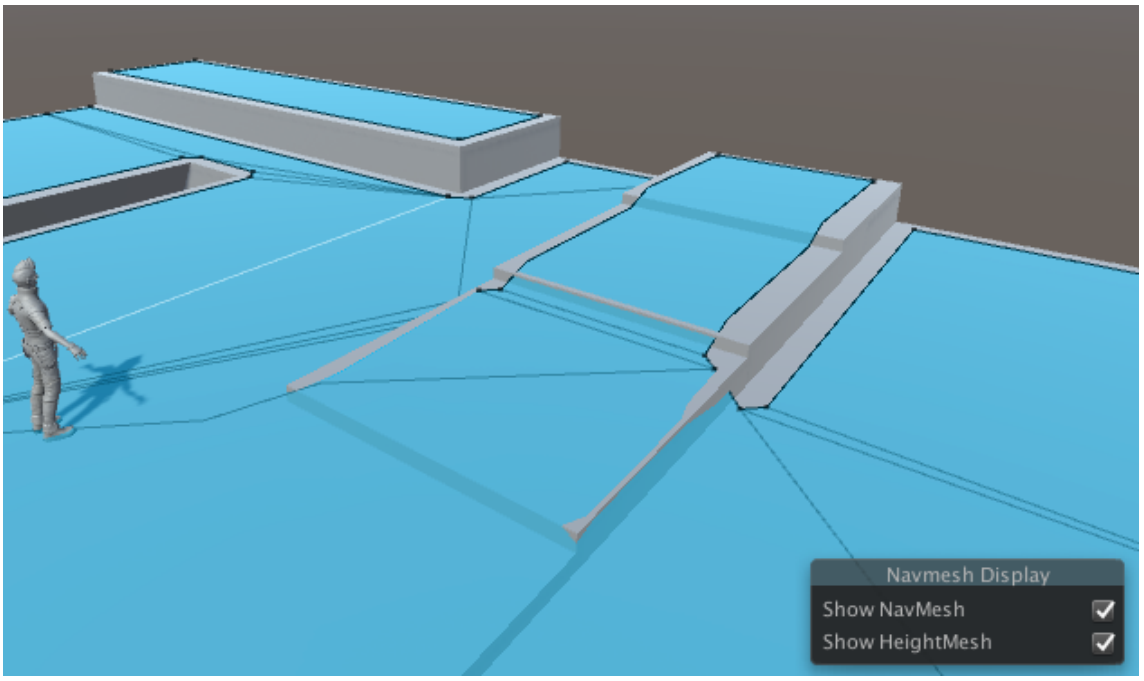

Figure 5.3: sample NavMesh

The obvious caveat with the approach is the fact that between those points there is a "void", which in some cases, if the are is mostly open, translates to a low point density for an AI to traverse and, therefore, there is no room for *tactical pathfinding,* that is to say, a theoretical scenario where an AI evaluates nodes around it in order to select with ones are most suitable to navigate next (according to, for instance, enemy presence).

A* Pathfinding Project: Through the years, many algorithms have been employed to solve the problem of agent navigation in a 3D space. One of the most flexible is *A**[21], which can be integrated in Unity via the *A* Pathfinding Project package.*

The algorithm populates a uniform grid of nodes so that the AI can move between them. It is because of this fact that the *A* Pathfinding Project* is used in the project, in order to achieve behaviours such as searching the optimal node for cover.

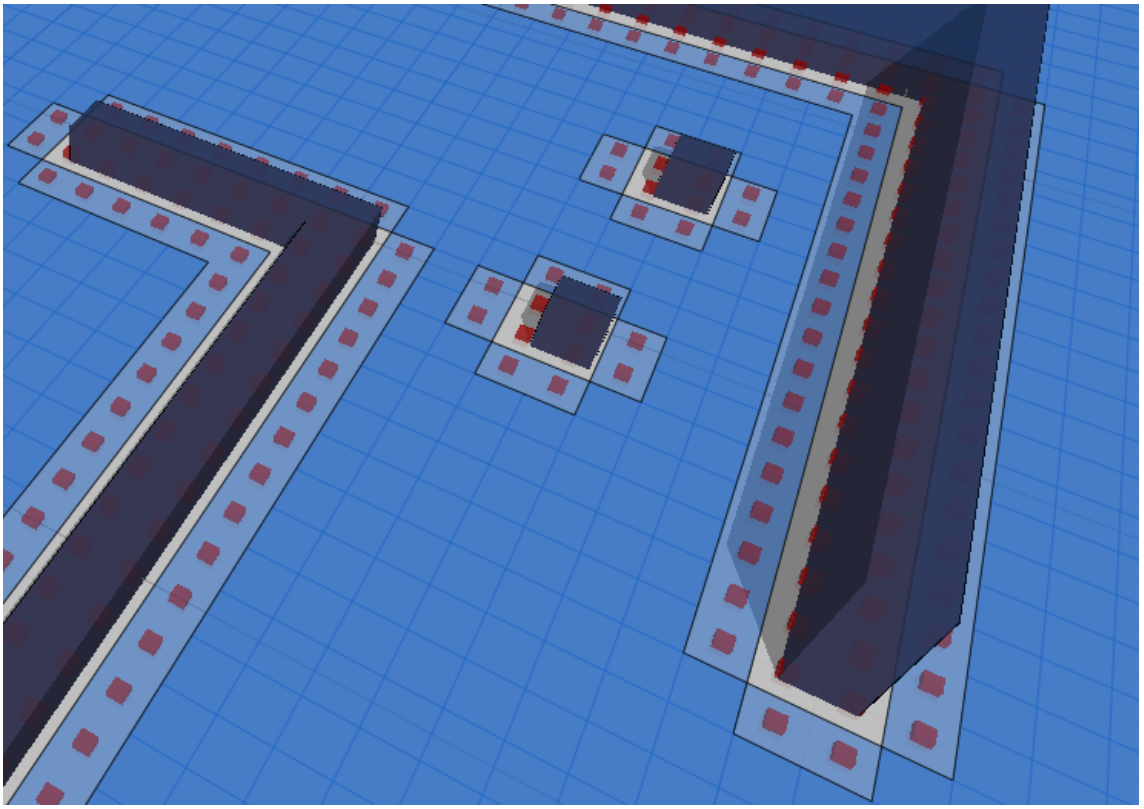

Figure 5.4: uniform A grid that wraps around obstacles*

5.1.5 Blockout

Finally, to satisfy the need for a simple shooter map, I have made use of the package ***"Blockout",*** a tool that comes with pre-made 3D shapes, options easily control the move and rotate tools (snapping) and both structuring and coloring of objects depending on their type (wall, floor, etc.).

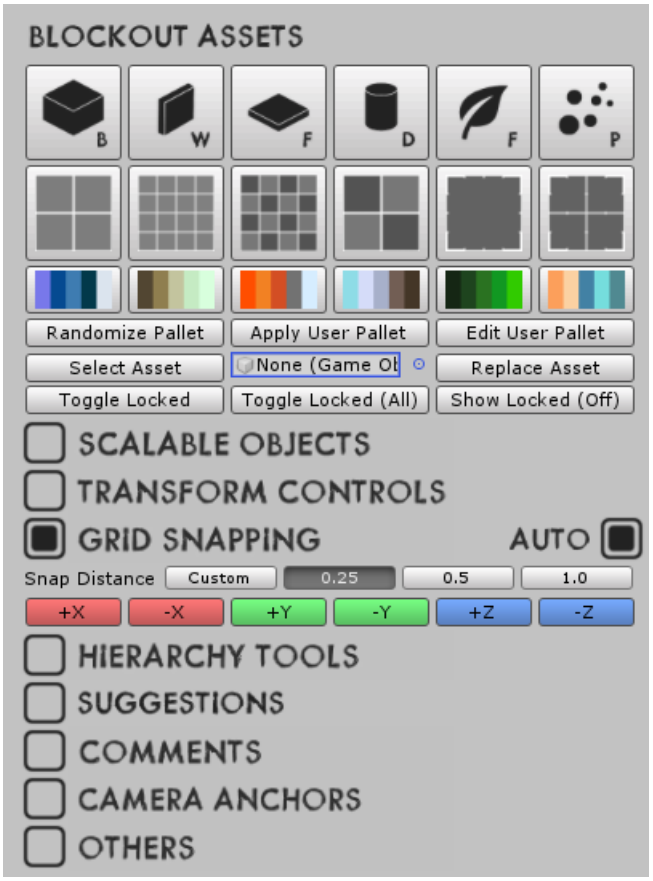

Figure 5.5: Blockout menu interface

The results that can be obtained with this extension are the same as those that could potentially be achieved with the base engine, although the process is time-saving. Other options include ***Pro-Builder,*** a more advanced tool that permits custom shape creation, *vertex painting*, etc, although these functionalities are far beyond the needs of the map.

5.2 Project Development

5.2.1 Unity Overview

With the ultimate goal of developing several disciplines (level design, automation, AI…) in parallel, the *Github* repository contains one *Unity project* for each one.

An example of a recurrent work pipeline would be developing or iterating the map in the correspondent project, then exporting it to a file (by using a package by the name of *"FBX exporter)* and then importing it into the automation project, to first generate the navigation mesh and then proceed with the tests.

Inside *Unity,* the interface (image below) is mainly divided in:

- **Hierarchy** (left): holds the totality of *gameobjects*, that is to say actors that contain *components*, which at the same time provide the object with logic. For the automation project, it includes: AIs, map, managers...
- **Scene** (middle): shows the current spatial representation of the whole hierarchy. In this case, the custom map with a navigation mesh in blue.
- **Inspector** (right): Once a *gameobject* is selected, the inspector shows the current *components* attached to it. In the example, an AI is selected, thus some navigation and other components are shown.
- **File explorer** (bottom): this section visualizes almost all the files within the project. Normally, it is composed of *scrips* (code files that can be added as *components*), *scenes* (files with an encapsulated hierarchy) and others.

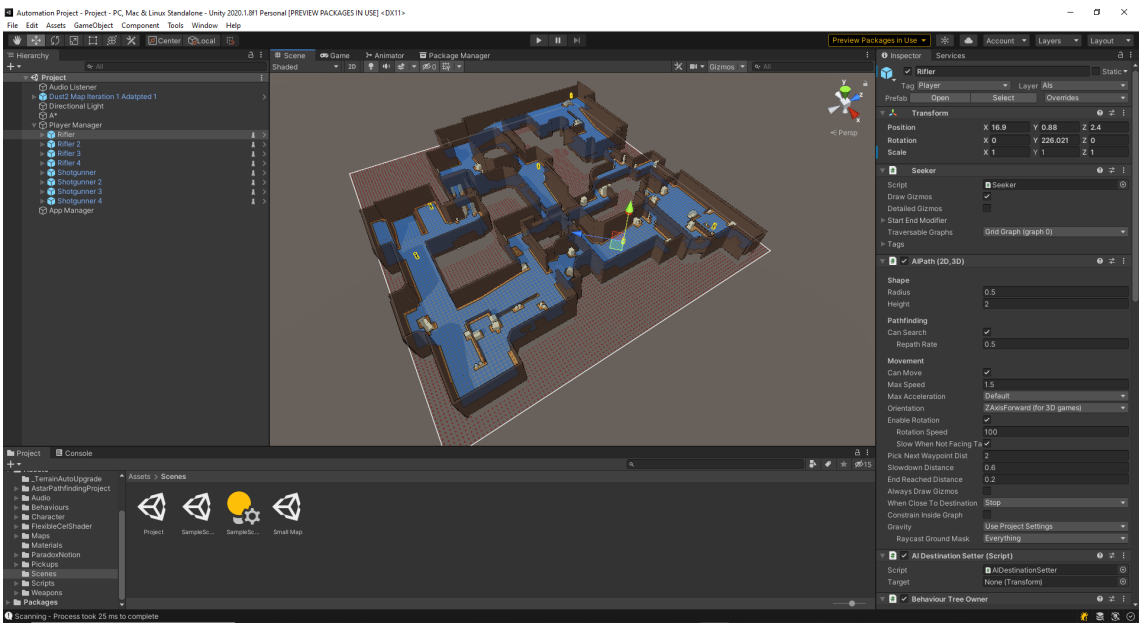

Figure 5.6: Unity overview

5.2.1 Blockout

As mentioned in the last section, the "*Blockout*" package was used to generate the geometry needed for the map, that is to say: floors, walls, obstacles and even optional roofs.

To establish a base, I used the original map in the game *Counter-Strike (1999)*, which is available to be downloaded here:

https://sketchfab.com/3d-models/de-dust2-cs-map-056008d59eb849a29c0ab6884c0c3d87

Once the floors and walls matched up with said version of the map, I switched focus to the current version, in *Counter-Strike Global Offensive*, as a reference for obstacles and in order to modify some walls.

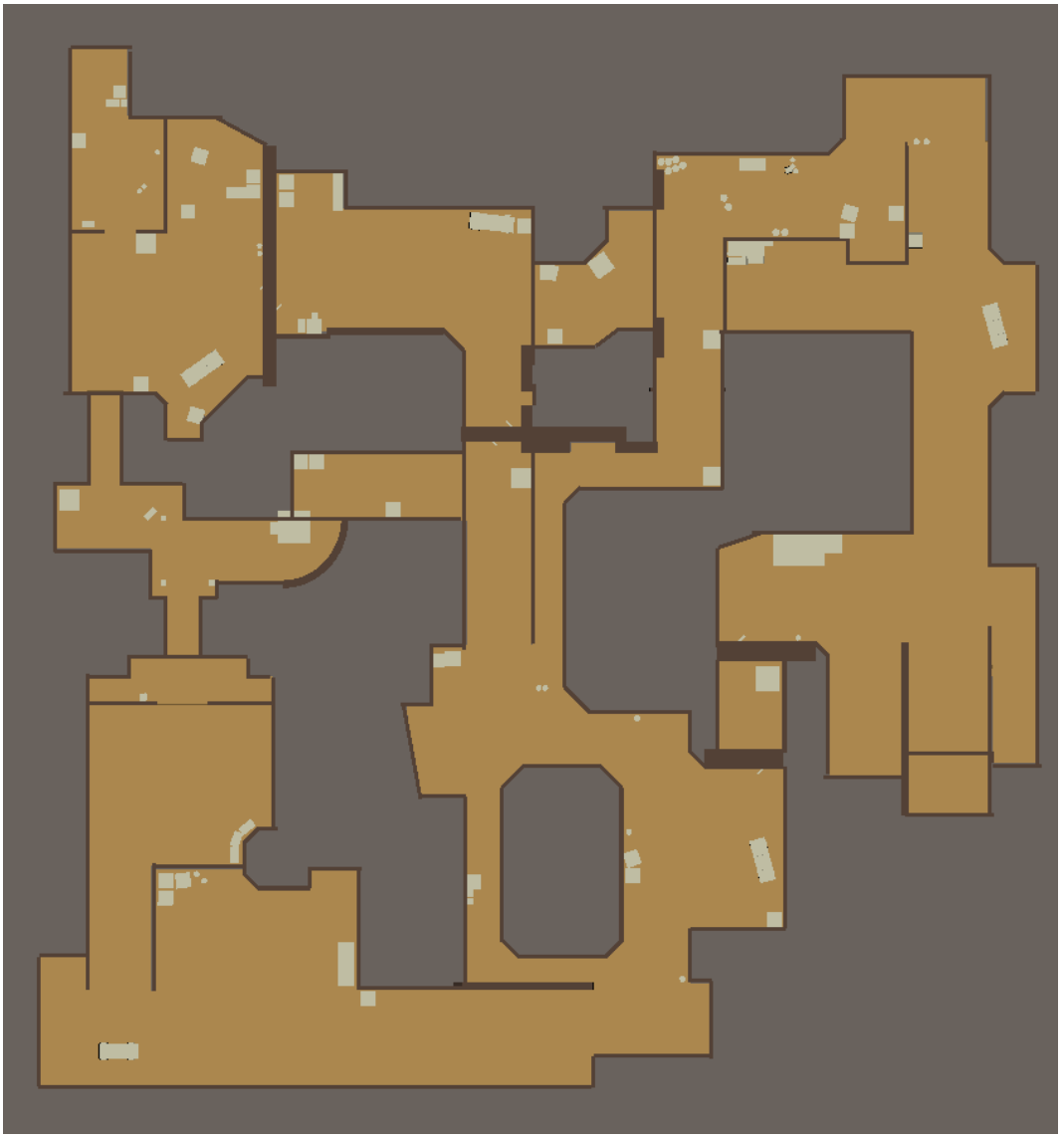

Figure 5.7: Custom Map top-down view

Figure 5.8: CSGO's Dust2 Map top-down view

The result is a close-enough simplified adaptation that lets me proceed with the playtesting with the certainty that the current proportions, balance and design of the map make sense.

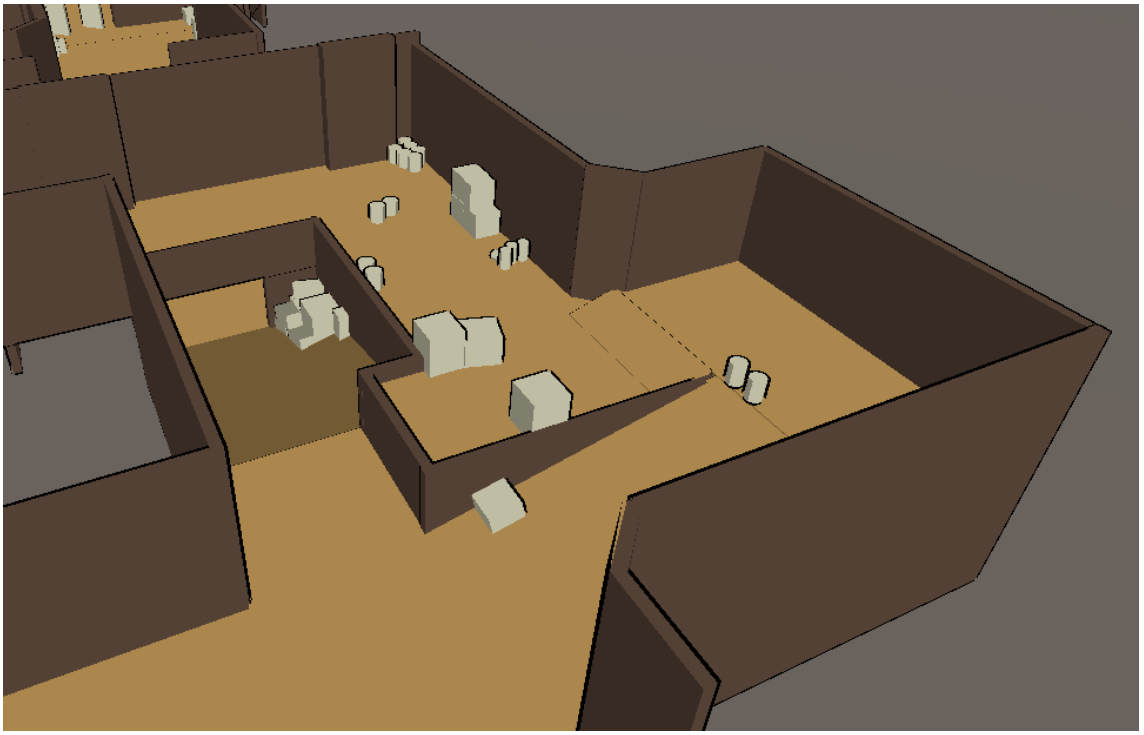

Figure 5.9: Original custom Map "A bombsite"

Figure 5.10: CSGO's Dust2 "A bombsite"

There has been one alteration of the so-called *"CT Ramp"* area, present in both images above, which now does not generate an underpass.

The reason being that the pathfinding module does not provide support for floors one on top of the other. Fortunately, modifications of this sort are actually planned as part of the playtesting process.

Figure 5.11: "A bombsite" overview with "CT Ramp" modification

5.2.2 Pathfinding

The pathfinding module is able to scan the map for obstacles and generate a navigation mesh, or graph, with native support for ramps.

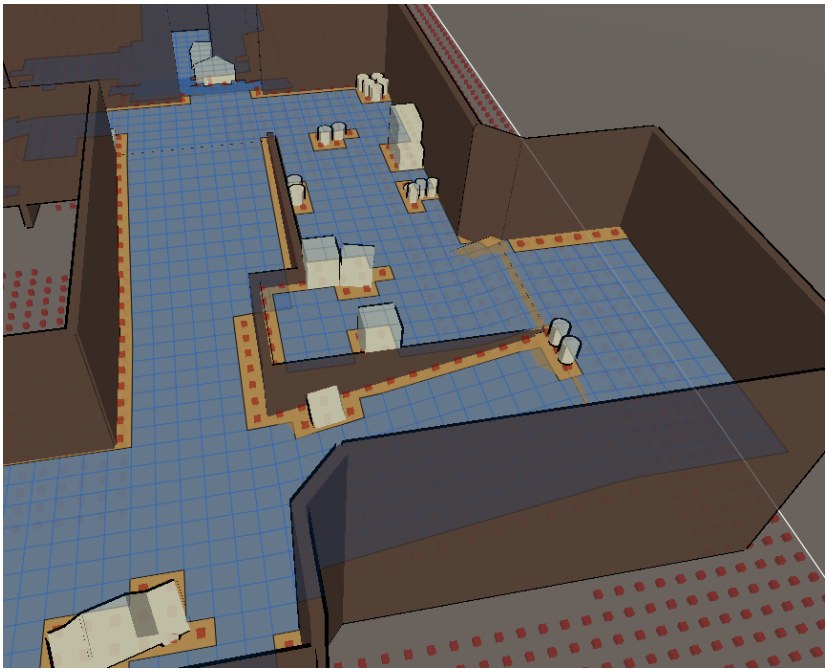

Figure 5.12: "A bombsite" navigation graph

There is one gameobject in the scene that has a "*Pathfinder*" component, which dictates several parameters of the graph (connections, slope, erosion...)

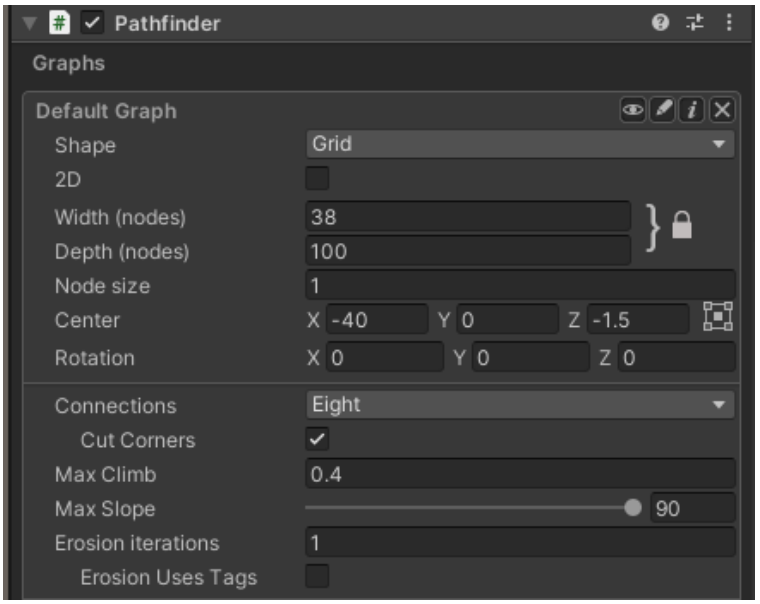

Figure 5.13: "Pathfinder" component

The scanning is normally done prior to execution, although it can also be updated at runtime.

As for the AI, each agent has components such as *"AIPath"*. This script handles the calculation of optimal new paths and at the same time offers parameters that simulate the agent's size and speed.

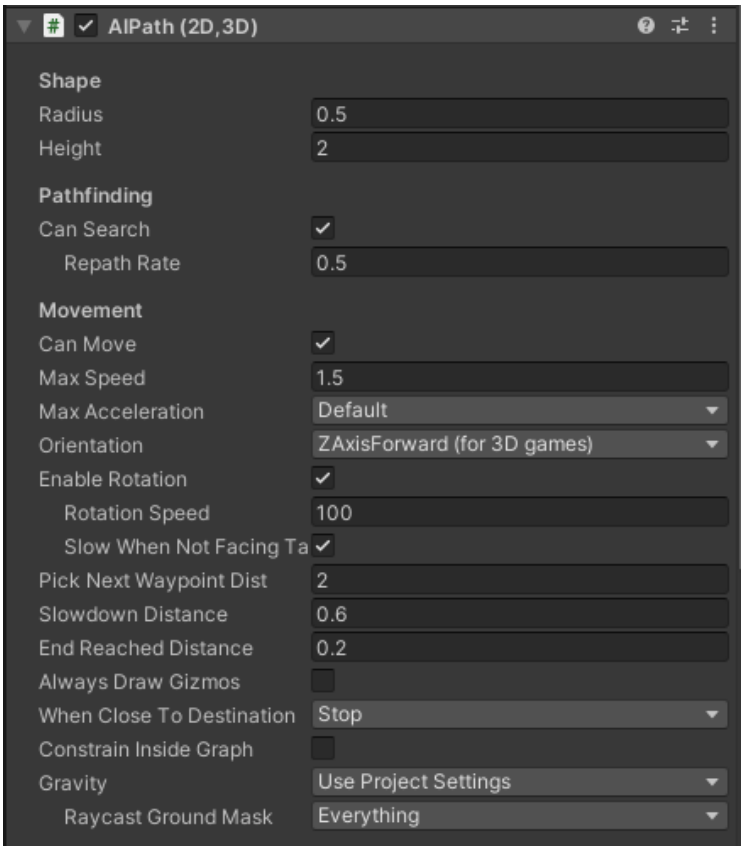

Figure 5.14: "AIPath" component

Lastly, an interesting parameter is erosion: each erosion unit represents one node away from obstacles. With this in mind, one can use tags to penalize nodes closest to obstacles, or the opposite.

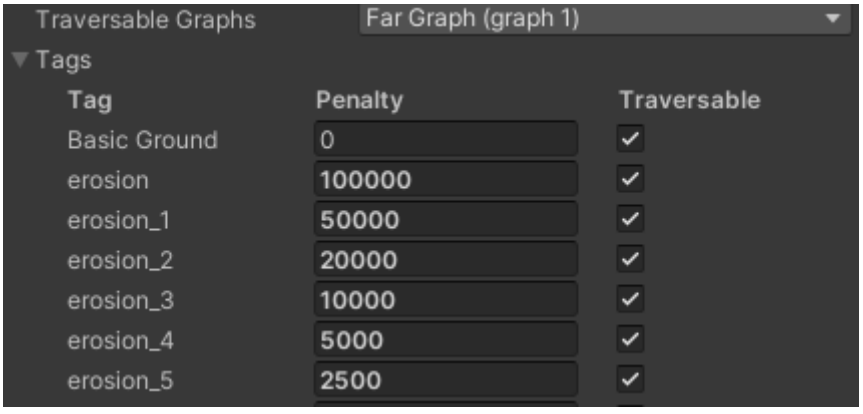

Figure 5.15: Erosion

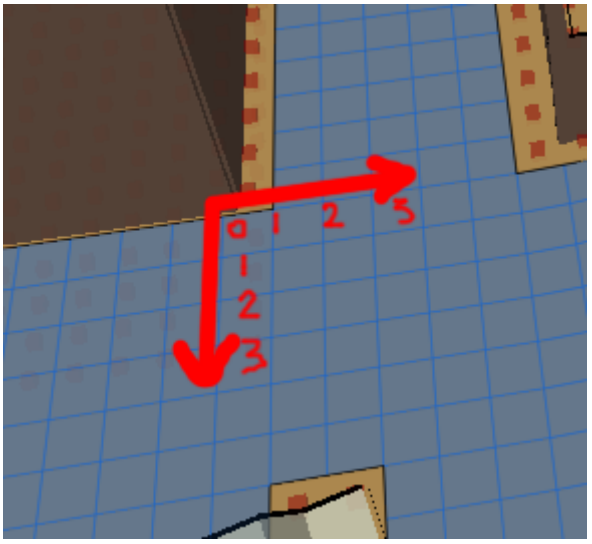

Figure 5.16: Erosion units example

5.2.3 AI

In this section, I will expand on the AI's coding and set-up. Firstly, the "types" section will give a glimpse of which playstyles and weapons can characterize the AI. Then, the "Prefab" section will expand on which components and child objects the AI has. Behaviour Trees will be next and finally there are some comments on weapon and pickup functionalities.

Types

As I have established previously, the AI is divided into agents with different playstyles:

- **Killer**: when hearing an enemy, the agent will go to the position of interest. Prioritizes fights; ignores point pickups[24].
- **Collector**: wanders around actively searching for point pickups. In order to achieve said purpose, the logic tries to find line of sight to a pickup and, if there is none, a random pickup in the map is chosen as the destination.

  Once the AI is navigating through the map, if there is a visible pickup different from the previous target, the new destination is set to this pickup.

```
aIPerception.VisualDetection(true);
List<GameObject> detected = aIPerception._visuallyDetected();
int closestIndex = 0;
float nearDist = float.MaxValue;
bool targetFound = false;

if (detected.Count > 0)
{
    for (int i = 0; i < detected.Count; ++i)
    {
        GameObject go = detected[i];

        if (go.CompareTag("pickup"))
        {
            if (go.GetComponent<Pickup>().pickupType == lastSearched)
            {
                targetFound = true;
                float dist = (go.transform.position - agent.transform.position).magnitude;

                if (dist < nearDist)
                {
                    nearDist = dist;
                    closestIndex = i;
                }
            }
        }
    }

    if (targetFound)
    {
        path.destination = (Vector3)AstarPath.active.data.gridGraph.GetNearest(detected[closestIndex].transform.position).node.position;
    }
}
```

Figure 5.17: Code snippet for the collector behaviour

- **Socializer**: when an ally is fighting, the agent will support the fight.

These basic behaviours are a sample of a myriad of possible functionalities, within the scope of the main *Bartle Types*, although the reason to employ them is that they can be combined with the following concept: weapons.

According to the agent's weapon (rifle, shotgun, sniper rifle), the agent will behave differently:

- **Rifler**: searches for cover between him and the enemy. That is to say, the code scans all nodes and tries to find a node closer to the AI than to the enemy which provides a certain amount of cover.

```
bool PositionSuitable(GraphNode node, GameObject enemy)
{
    // Not Walkable
    if (node.Walkable == false)
    {
        return false;
    }

    Vector3 enemyPos = enemy.transform.position;
    Vector3 diff = agent.transform.position - enemyPos;
    Vector3 dir = diff.normalized;
    Vector3 nodePos = (Vector3)node.position;

    // I am too close to enemy
    float distToEnemy = (agent.transform.position - enemyPos).magnitude;

    if (distToEnemy < aIParameters._rifleCoverMinTriggerDist())
    {
        return false;
    }

    float nodeToPos = (nodePos - agent.transform.position).magnitude;
    float nodeToEnemy = (nodePos - enemyPos).magnitude;


    // Too far
    if (nodeToPos > aIParameters._rifleCoverMaxDist())
    {
        return false;
    }

    // Too close
    float marginalDistance = 3f;
    if (nodeToPos < marginalDistance)
    {
        return false;
    }

    // Too far from enemy
    if (nodeToEnemy > distToEnemy)
    {
        return false;
    }

    // Too far from player in comparison to enemy
    float percentage = 0.2f;

    if (nodeToPos / nodeToEnemy > percentage)
    {
        return false;
    }


    int nHits = aIPerception.LOF_FromNodePos((Vector3)node.position, enemy);
    int maxHits = aIPerception._raycastTargetOffsets().Length;

    // No cover
    if (nHits == 0)
    {
        return false;
    }

    // Total cover (no Line Of Fire)
    if (nHits == maxHits)
    {
        return false;
```

Figure 5.18: Full list of checks for the rifler “search cover” behaviour

- **Shotgunner:** if the enemy is at medium range, runs towards it to shorten the distance and then fight.
- **Sniper:** prioritizes staying at a distance from all cover to only engage on longer range fights *(see figures 5.15 and 5.16).*

The playstyle and weapon of choice can be combined, resulting in 9 AI types. As will be explained later in the Behaviour Tree section, the playstyle logic and the weapon logic do not interfere with each other.

Prefab[23]

Each type of AI is stored in a prefab, for instance the “killer rifler” or the “collector shotgunner”.

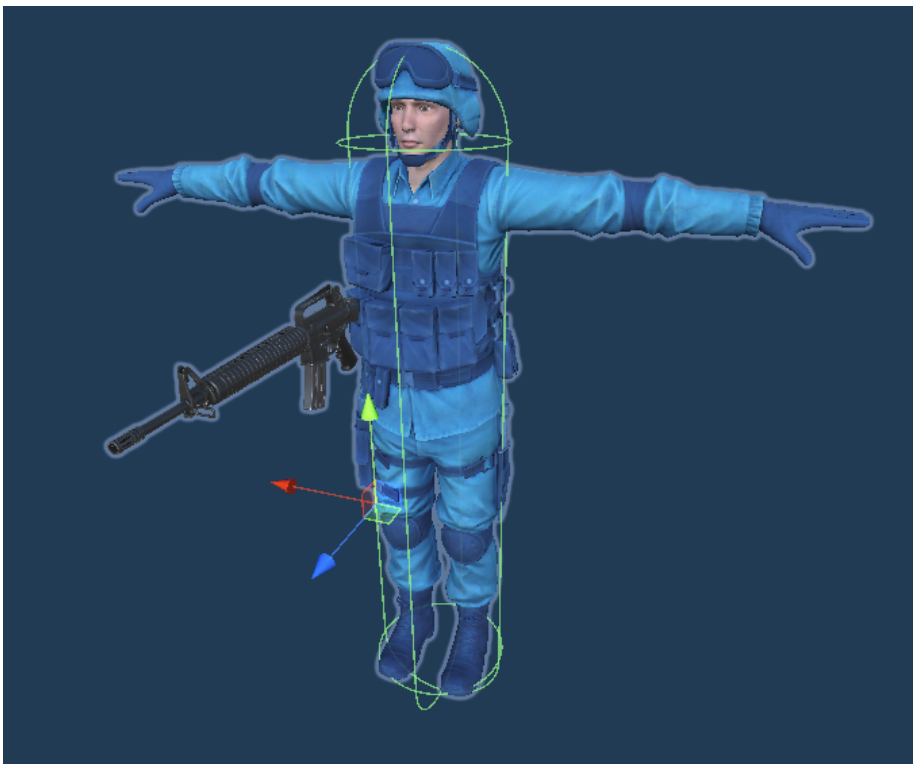

Figure 5.19: Rifler

This prefab is conformed of a series of components, that can be categorized in:

- **Pathfinding:** *“AIPath”* and *“AIDestinationSetter”* (handles current destination).
- **Animator:** in *Unity,* the “*Animator”* component has an *“Animation Controller”* reference. Said controller is made of nodes, with an animation for each node, and transitions, that are triggered by certain variable changes in the animator.

The ultimate goal of the *"Animation Controller"* is to associate every AI movement state with the corresponding animation, which has been previously imported to the project.

Note: both the character and the animations from this project have been exported from https://www.mixamo.com/

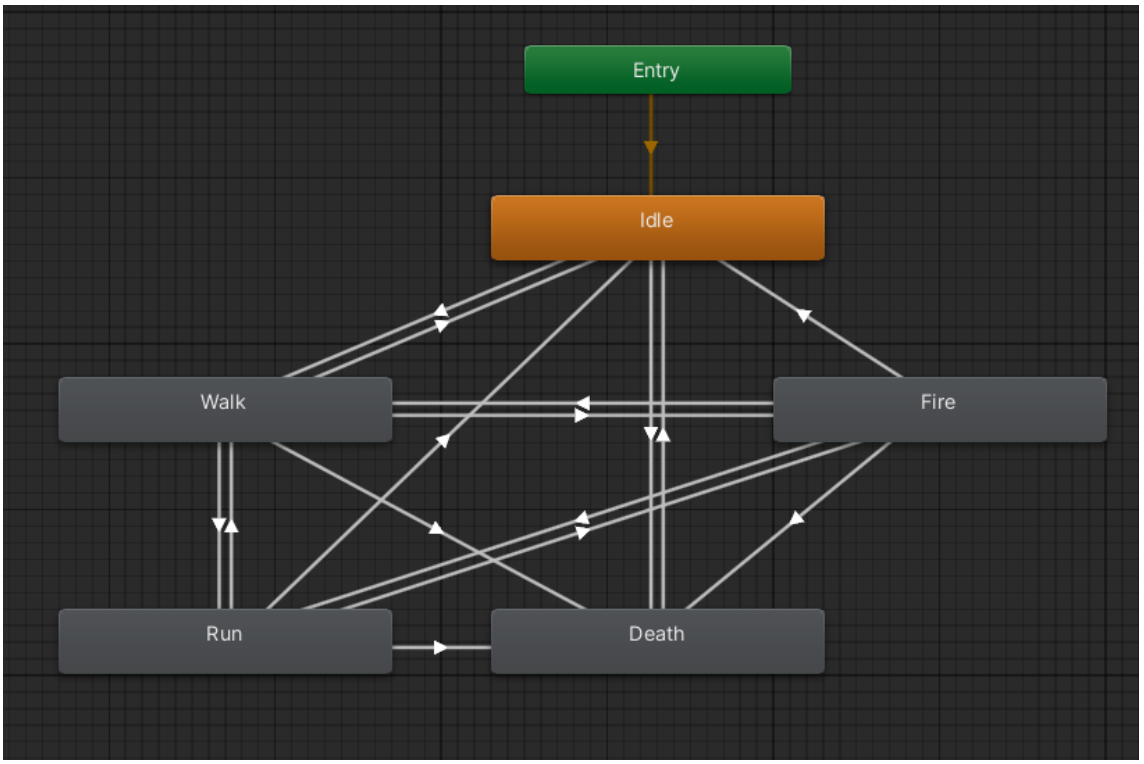

Figure 5.20: AI's animator

- **Behaviour Tree:** *"Behaviour Tree Owner"*, which links the prefab with a Behaviour Tree *asset*[25], and *"Blackboard"*, a complement for the Behaviour Tree that stores variables which can alter the flow of execution.

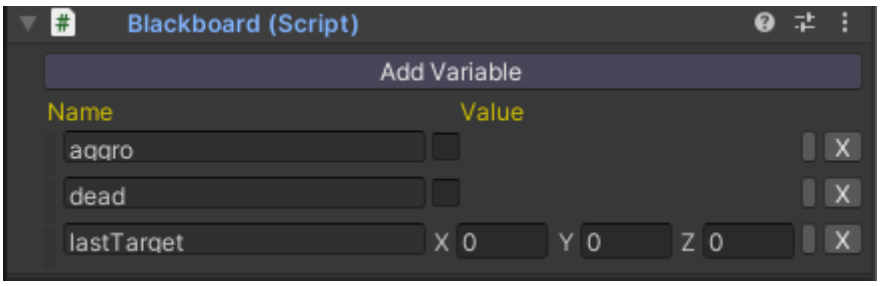

Figure 5.21: Blackboard

- **Logic:** I have coded these scripts to both parameterize the AI and to make the agents aware of their surroundings.

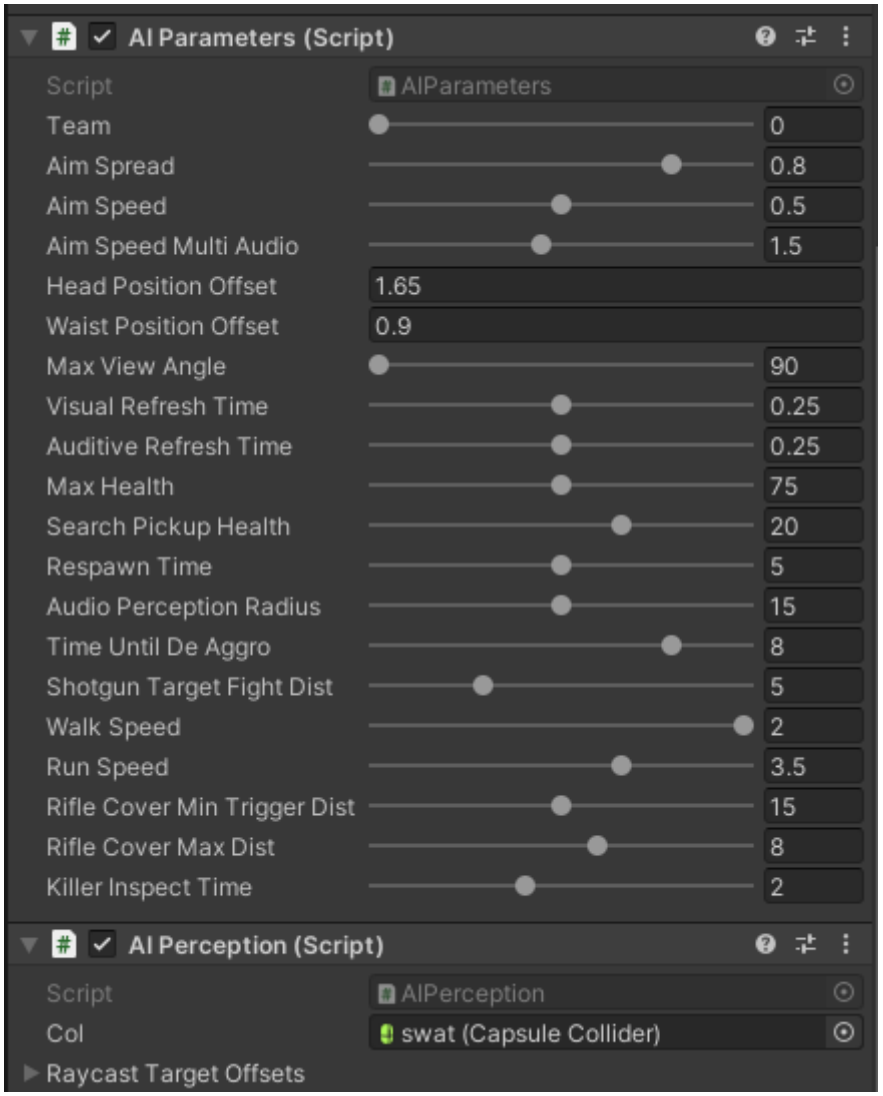

Figure 5.22: AI Parameters and AI Perception

"AIParameters" holds variables that mainly account for the AI's mechanical skills, the most relevant ones being:

- **Team:** the AI only fights against the other teams' AIs.
- **Aim Spread, Speed**: they handle how skilled the AI is in regards to aiming accurately and fast. If an enemy is detected via audio and then visually, the aim speed increases, by *"Aim Speed Multi Audio"* times.
- **Visual/Auditive Refresh Time**: amount of time until the next perception logic frame is executed. The purpose of these parameters was to simulate a human's brain response time
- **Max View Angle:** as humans, the AI will only detect enemies in front of them, within a certain angle.

As for "AIPerception", the script checks every "*Visual/Auditive Refresh TIme"* for visual and auditory cues.

Visually, enemies are detected via multiple raycasts (lines), that range from the AI's head to multiple body parts (hence *"Raycast Target Offsets"*).

If one of those rays collides with an enemy (and thus it does not collide with an obstacle or an ally), a target is found.

Apparently, another possibility within Unity was to employ a camera for each agent to detect when an enemy starts to be rendered.

The caveat for this option is that the method in question: **MonoBehaviour.OnBecameVisible()** is called when *any* camera detects an object and thus there is no way to tell which AI detected the object.

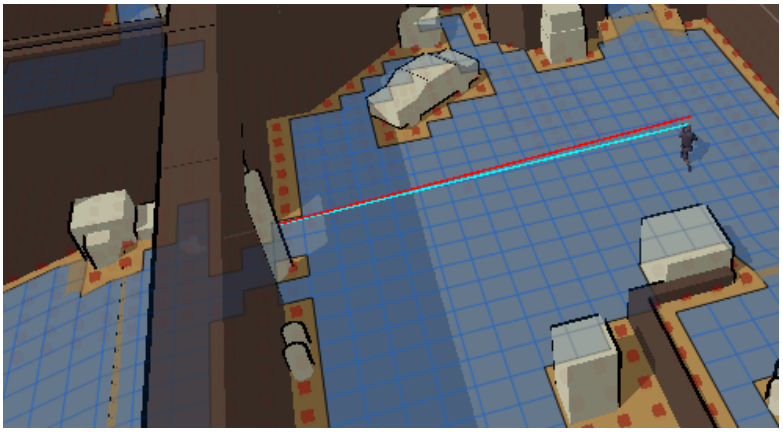

Figure 5.23: Sample raycasts to enemy

On the other hand, audio cures are detected via a sphere surrounding the AI.

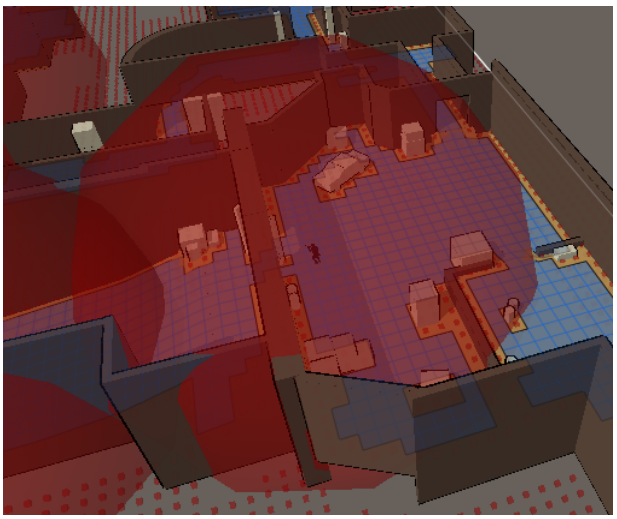

Figure 5.24: Audio sphere

The code snippet for auditive detection is the following:

```
public void AuditiveDetection(bool detectAllies = false)
{
    if ((auditiveTime += Time.deltaTime) >= parameters._auditiveRefreshTime())
    {
        auditiveTime = 0f;
        audioDetected.Clear();

        // Search new enemies inside radius
        var colliders = Physics.OverlapSphere(transform.position, parameters._audioPerceptionRadius(), 1<<9); // AI layer

        foreach (Collider col in colliders)
        {
            GameObject go = col.transform.parent.gameObject;

            if (go.CompareTag("Player"))
            {
                if (go != gameObject)
                {
                    AIParameters aIParameters = go.GetComponent<AIParameters>();

                    if (aIParameters._team() != parameters._team() || detectAllies)
                    {
                        Animator animator = go.GetComponent<Animator>();

                        if (animator.GetInteger("Moving") == 2)
                        {
                            audioDetected.Add(go);
                        }
                    }
                }
            }
        }
    }
}
```

Figure 5.25: Audio detection

Briefly, the method searches within the AI layer (which encompasses only objects inside Unity editor which belong to said layer) if any enemy is within the sphere's radius and, at the same time, if the enemy is running, because only running enemies emit sound.

Apart from components, the prefab also encompasses other objects, as childs:

Figure 5.26: AI's hierarchy

- **"swat":** the 3D model which will be animated
- **"Line Of Sight:** a camera now used for debugging purposes
- **"Weapon Slot":** contains the agent's weapon

Behaviour Tree Logic

There are many options in terms of the actions and logic that can define a shooter AI and at least two options in regards to how to structure them in Behaviour Trees: one BT (Behaviour Tree) for each type, or a unique BT with functions that execute a logic fragment or another within the code.

Following the main paradigm of BTs, that is to say, to compartmentalize them as much as possible, I have decided to create one generic Behaviour Tree to then duplicate it and substitute the key methods for each other BT. For every BT, and In order to simplify and split the logic, the agent can perform 3 main actions:

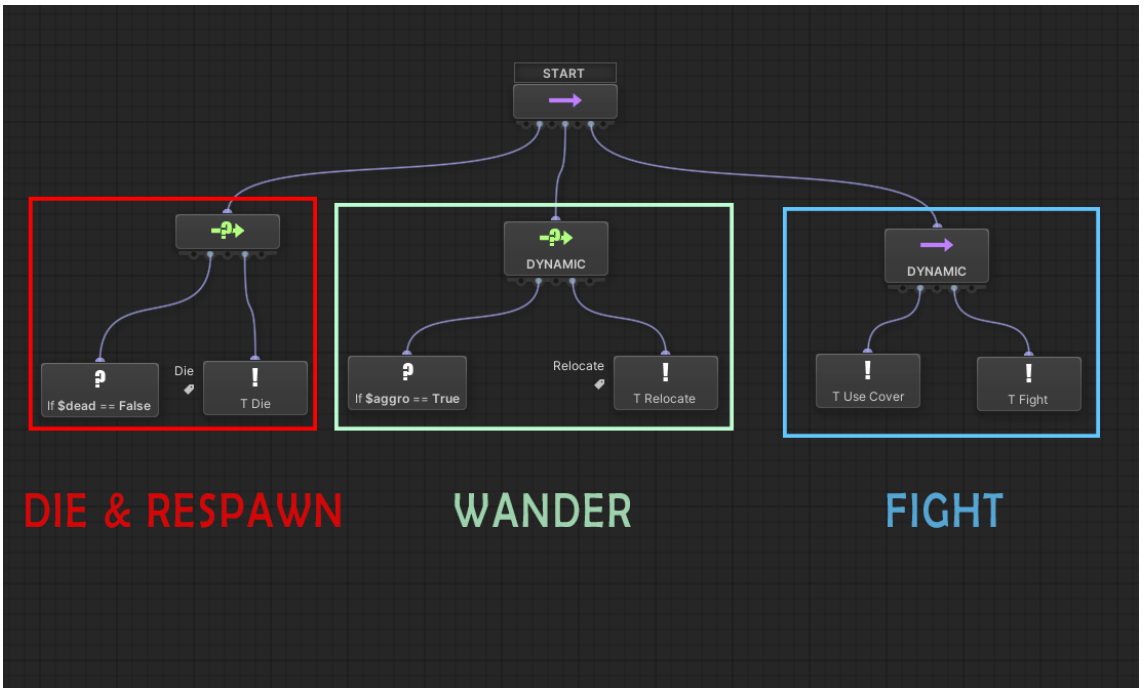

Figure 5.27: Sample AI Behaviour Tree phases

- **Die & Respawn:** handles the logic of being killed and calculating a new spawn position; in order to respawn, the code searches for a navigable node where there is no line of fire to an enemy.
- **Wander:** This phase represents the notion of navigating through the map without fighting. For collectors, the wander target position is that of a points pickup. If, at any moment in time, there is a nearer visible pickup, the target is updated to said pickup.

  In the case of the other types, instead of searching for pickups, they navigate to a random position in the map.

  In all cases, health pickups are available if the current health is below a certain percentage.

- **Fight:** the fight phase is triggered if either the visual or auditory perception detects an enemy during the wander phase.

  As mentioned in the “Types” section, each AI behaves differently according to weapon and playstyle.

  On top of that, this phase dictates how the AI aims and fires.

  For the aiming, I have programmed a simple functionality: move the forward vector towards the vector that connects the AI’s head to the target. Once close, the AI is ready to fire.

```
bool Aim()
{
    for (int i = 0; i < aILogic._aggrodEnemiesIndexes().Length; ++i)
    {

        if(aILogic._aggrodEnemiesIndexes()[i] == false)
        {
            continue;
        }

        GameObject enemy = PlayerManager.instance.transform.GetChild(i).gameObject;
        AIParameters aIParameters = enemy.GetComponent<AIParameters>();
        GameObject weapon = aILogic._weaponSlot().transform.GetChild(0).gameObject;

        Vector3 origin = weapon.transform.Find("Weapon Tip Position").position;
        Vector3 destination = enemy.transform.position + enemy.transform.up * aIParameters._headPositionOffset();
        Vector3 targetDir = destination - origin;

        float aimSpeed = AIParameters._aimSpeed() *
            ((aIPerception.IsAudioDetected(enemy)) ? AIParameters._aimSpeedMultiAudio() : 1f)
            * weapon.GetComponent<WeaponParameters>().GetAimMulti()
            * Time.deltaTime;

        Vector3 newAimDir = Vector3.RotateTowards(currentAimDir, targetDir, aimSpeed, 0.0f);
        currentAimDir = newAimDir;

        Vector3 newForwadVector = agent.transform.forward;
        newForwadVector.z = newAimDir.z;
        agent.transform.rotation = Quaternion.LookRotation(newForwadVector);

        currentAimVector = currentAimDir * targetDir.magnitude;

        currentDestination = origin + currentAimVector;
        if ((currentDestination - destination).magnitude <= aimThreshold)
        {
            return true;
        }

    }

    return false;
}
```

Figure 5.28: Aim code snippet

As for firing, the code emulates a fire rate (depending on the weapon) and also a random spread (in *AIParameters*). If the enemy is killed, the *“aggro”* (aggressive, fight) state stops and the AI goes back to wander.

```
void Fire()
{
    if ((currentFireTime += Time.deltaTime) >= weaponParameters._fireRate() / 100f)
    {
        currentFireTime = 0f;

        float signedAimSpread = AIParameters._aimSpread() / 2f;
        float xOffset = Random.Range(-signedAimSpread, signedAimSpread);
        float yOffset = Random.Range(-signedAimSpread, signedAimSpread);
        float zOffset = Random.Range(-signedAimSpread, signedAimSpread);

        Vector3 offset = new Vector3(xOffset, yOffset, zOffset);

        RaycastHit hit;
        Vector3 bulletDestination = currentDestination + offset;
        Vector3 origin = aILogic._weaponSlot().transform.GetChild(0).Find("Weapon Tip Position").position;
        Vector3 direction = bulletDestination - origin;

        if (Physics.Raycast(origin, direction.normalized, out hit, Mathf.Infinity))
        {

            if (hit.transform.parent.gameObject.CompareTag("Player"))
            {

                AIParameters aIParameters = hit.transform.parent.gameObject.GetComponent<AIParameters>();
                if (aIParameters._team() == AIParameters._team())
                {
                    return;
                }

                if (PlayerManager.instance.DamageAI(weaponParameters.GetDamageAtDistance(direction.magnitude), hit.transform.parent.gameObject, agent.gameObject))
                {
                    EndAction(true);
                };
            }
        }
    }
}
```

Figure 5.29: Fire code snippet

Each phase is represented with different nodes. For instance, the exclamation mark indicates an action node, which executes an attached script. Both the selector (with an interrogation mark and a left-to-right arrow) and the sequencer (marked by a left-to-right arrow) execute the action until the task returns via “EndAction(true)” or when the condition is not met, in the case of the condition sequencer.

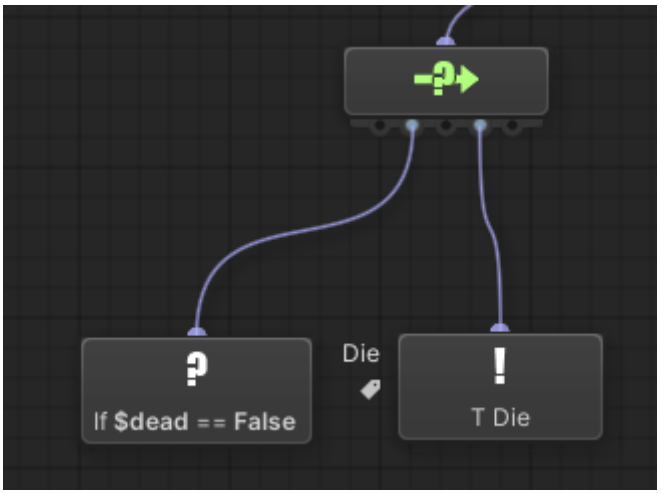

Figure 5.30: A selector with a condition and an action

Weapons

The three weapons in this project have common functionalities. First of all, each of them has a different fire rate and damage, with the sniper having the highest damage and the rifle having the highest fire rate.

On top of that, there is a damage curve that decreases with distance. Nonetheless, the sniper rifle is exempt from this distance penalty.

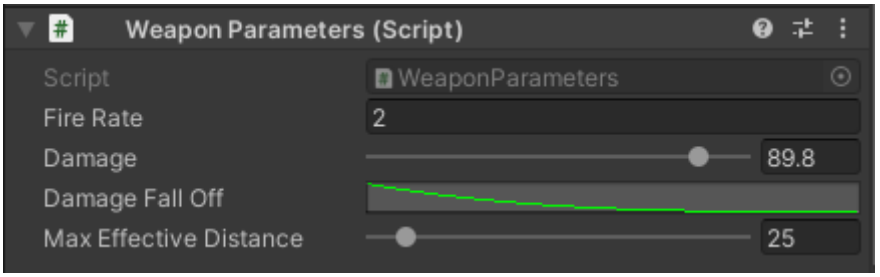

Figure 5.31: Weapon script inspector

The ultimate purpose of these weapons is to provide for different playstyles and, subsequently, the shooter level must offer opportunities for all of them.

Pickups

Spread throughout the map, there are both health and point pickups. Said pickups attribute a certain amount of either health to the individual or points to the team.

As for the concept of points, the motive behind them is to let AIs gain points both by killing and by collecting these pickups and, at the end of the playtesting, compare the performance of both approaches.

For each simulation, the placing and spread of these pickups will be altered, so as to provide for different case analysis and then compare the outcome.

A future improvement will be for the AI to not know with precision where these pickups are, in order to simulate players that do not know the level with 100% certainty.

5.2.4 Simulations

When importing *Unity Game Simulation*, a “Game Simulation” window is now available.

This window lets the user set-up parameters of different types: *bool, float, string*… So,thereafter, one can proceed to capture these variables via the code, so that they take effect.

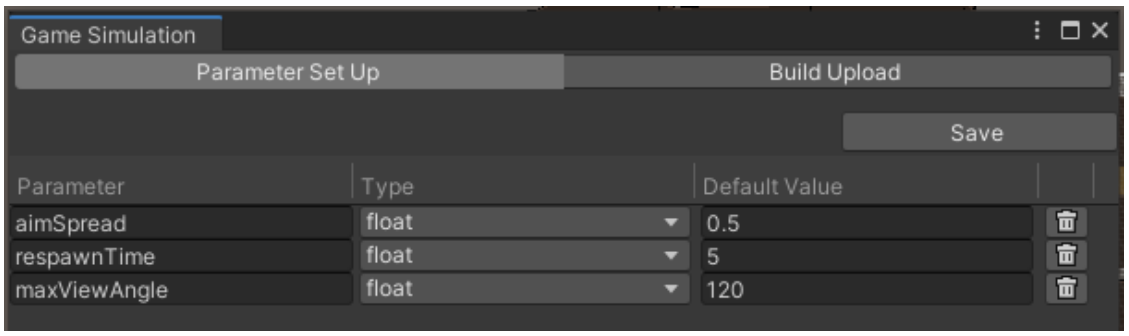

Figure 5.32: Game Simulation window

Following the documentation, one can achieve this behaviour by accessing the *"GameSimConfigResponse"* object.

This is done at the beginning of the agent's logic to set-up their parameters.

```
    GameSimManager.Instance.FetchConfig(OnConfigFetched);
}

1 referencia
void OnConfigFetched(GameSimConfigResponse config)
{
    aimSpread = config.GetFloat("aimSpread");
    respawnTime = config.GetFloat("respawnTime");
    maxViewAngle = config.GetFloat("maxViewAngle");
}
```

Figure 5.33: Game Simulation config

Not only that, but in order to track metrics, such as kills for each team, one can do so by modifying "counters", as the docs mention:

```
AIParameters e_aIParameters = emitter.GetComponent<AIParameters>();
string killCounter = "T" + e_aIParameters._team().ToString() + " kills";
string deathCounter = "T" + r_aIParameters._team().ToString() + " deaths";
GameSimManager.Instance.IncrementCounter(killCounter, (long)1);
GameSimManager.Instance.IncrementCounter(deathCounter, (long)1);
```

Figure 5.34: Game Simulation counter increment

Parameters and metrics have previously been discussed in 2.1 and, to elaborate on that, the current simulation dashboard for the project looks like this:

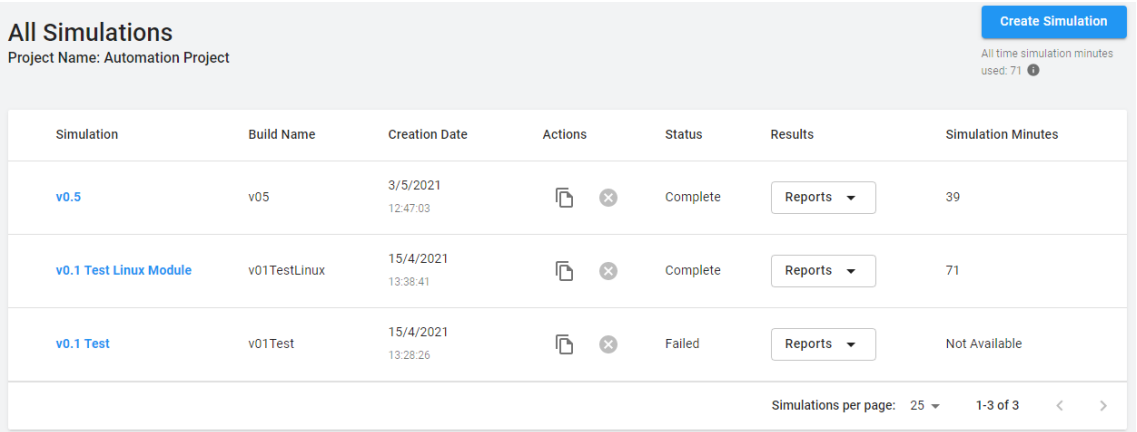

Figure 5.35: Game Simulation dashboard

As can be deduce from the image above, the “Create Simulation” button opens the menu mentioned in 2.1, which lets the user set-up different metrics and parameters, the latter ones within a range of possibilities.

In order to run a simulation, there must be a build, previously generated inside Unity and uploaded in the second tab of the *figure 5.32*.

In the image above, one can notice that there are a total of three simulations, one of them failed, due to an error in the code (Linux build required but Windows one provided), and the other ones succeeded. In any case, *Unity Game Simulation* outputs a “log” file with errors and warnings.

Apart from the log file, there is a “Results” file that shows the input parameters for each simulation and the resulting metrics.

_name	run_id	name	value	score	game_sim_settings	exit_status
5	6	T0 kills	4	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.25}	
5	6	T1 deaths	8	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.25}	
5	6	T2 deaths	3	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.25}	
5	6	T2 kills	7	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.25}	
5	6	T0 deaths	1	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.25}	
5	6	T1 kills	1	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.25}	
5	7	T2 kills	4	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.5}	
5	7	T1 deaths	7	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.5}	
5	7	T0 kills	9	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.5}	
5	7	T2 deaths	4	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.5}	
5	7	T1 kills	6	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.5}	
5	7	T0 deaths	8	NaN	{"respawnTime":10,"maxViewAngle":130,"aimSpread":0.5}	
5	1	T2 kills	9	NaN	{"respawnTime":5,"maxViewAngle":90,"aimSpread":0.5}	Ok
5	1	T1 deaths	5	NaN	{"respawnTime":5,"maxViewAngle":90,"aimSpread":0.5}	Ok
5	1	T0 deaths	7	NaN	{"respawnTime":5,"maxViewAngle":90,"aimSpread":0.5}	Ok
5	1	T0 kills	4	NaN	{"respawnTime":5,"maxViewAngle":90,"aimSpread":0.5}	Ok
5	1	T2 deaths	3	NaN	{"respawnTime":5,"maxViewAngle":90,"aimSpread":0.5}	Ok
5	1	T1 kills	2	NaN	{"respawnTime":5,"maxViewAngle":90,"aimSpread":0.5}	Ok
5	2	T2 kills	9	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.25}	
5	2	T1 deaths	4	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.25}	
5	2	T0 deaths	7	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.25}	
5	2	T1 kills	3	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.25}	
5	2	T0 kills	4	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.25}	
5	2	T2 deaths	5	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.25}	
5	3	T0 kills	3	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.5}	
5	3	T2 deaths	5	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.5}	
5	3	T1 kills	7	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.5}	
5	3	T0 deaths	4	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.5}	
5	3	T2 kills	1	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.5}	
5	3	T1 deaths	2	NaN	{"respawnTime":10,"maxViewAngle":90,"aimSpread":0.5}	
5	5	T0 kills	4	NaN	{"respawnTime":5,"maxViewAngle":130,"aimSpread":0.5}	
5	5	T1 deaths	7	NaN	{"respawnTime":5,"maxViewAngle":130,"aimSpread":0.5}	
5	5	T1 kills	4	NaN	{"respawnTime":5,"maxViewAngle":130,"aimSpread":0.5}	
5	5	T0 deaths	7	NaN	{"respawnTime":5,"maxViewAngle":130,"aimSpread":0.5}	
5	5	T2 kills	6	NaN	{"respawnTime":5,"maxViewAngle":130,"aimSpread":0.5}	

Figure 5.36: Sample simulation report snapshot

5.2.4 Level Design Parameterization

In regards to parameterization, there has been a first attempt at enumerating level design variables and establishing the relation between, for future evaluation.

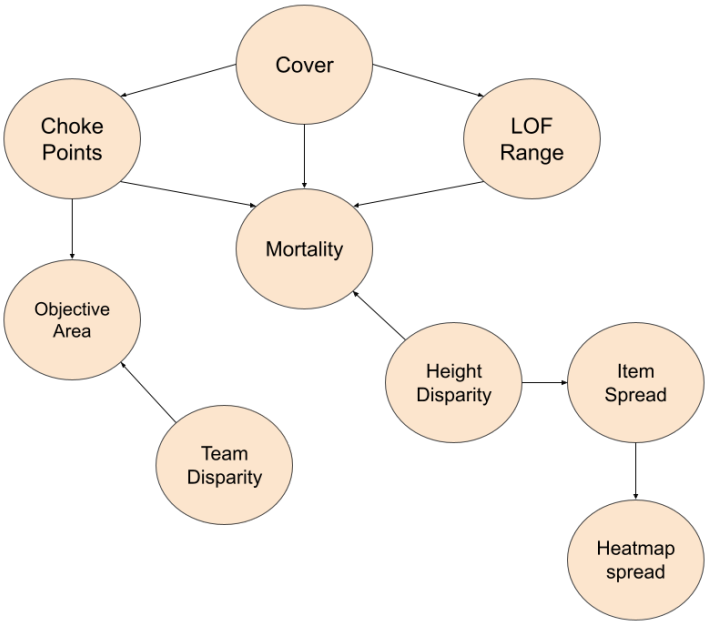

Figure 5.37: Shooter level evaluation

In the image above, each node represents one design variable. The most common throughout the shooter genre is cover, which at the same time influences which is the LOF range, or distance to other enemies, mortality and choke points, which are sections of the level that precede an objective, are straightforward and lack cover.

At the same time, choke points tend to lead to objective areas, that is to say “hot zones” where a team needs to stay or items like pickups or a flag which need to be captured.

Following the objective area branch there is the concept of team disparity, which, commonly, refers to one team defending said objective area from a better position than the one attacking it.

As for mortality, usually higher areas provide for a better angle to hit the enemies’ head, normally associated with higher damage.

At the same time, having multiple heights, or floors, affects item spread and the subsequent AI navigation spread.

I have found that many tactical shooters such as *CSGO or Valorant, etc* as well as other types of shooters can be summarized in these variables, and thus I conclude that the model is a proper base for the evaluations to come.

5.3 State of Development (Rubric 2)

The main focus of development until this point, as mentioned previously, the AI, is around 90% completed, with minor tasks remaining such as the socializer behaviour and code cleaning.

As for the blockout, the main layout is completed. Furthermore, small tweaks had to be made *(see figure 5.11)*, which coincidentally paved the way for actual map variations in the playtesting process.

At the time of releasing the build v0.5, there are multiple agents, 9 in total, divided in three teams, which navigate the map and fight each other.

Not present in the build though are the simulation capabilities, which exist in the *Unity Game Simulation* dashboard. Said package is already capable of running multiple simulations with different values each, which alter the AI’s variables inside the code.

Lastly, in terms of parameterization, a first theoretical model has been established. In the following weeks, this model ought to become a reference and ought to be translated into a numerical model for map evaluation.

5.4 State of Development (Rubric 3)

Succeeding the AI, the focus has been switched to analytics and manual playtesting. As for analytics, currently the code is able to generate:

- Heatmap: shows the most navigated positions. Per-agent basis
- Deathmap: shows death positions by agent
- “Pickupmap”: shows the number of times a pickup has been picked (globally)
- TTK (Time To Kill): shows, per team, the average time between kills

Along with the next playtestings, my intention is to expand the analytics module as I introduce elements such as risk-reward and more importance to height disparity.

Ultimately, this module will be capable of referring to the original theoretical model and give the levels some sort of score.

As for the human playtesting, I have coded a controller that can move, aim, shoot and collect pickups.

This functionality is the last code structure needed for the whole process. From this point onwards the development will orbit around iterating the analytics module, the playtesting process and, ultimately, the level that is being tested.

6. Conclusions and Future Projects

7. Bibliography

1. **Unity Game Simulation**
   https://blogs.unity3d.com/2020/12/11/automate-your-playtesting-create-virtual-players-for-game-simulation/?utm_source=linkedin&utm_medium=social&utm_campaign=ml_global_generalpromo_2020-12-11_virtual-player-game-simulation-blog

2. **Creating perfectly balanced gameplay with Unity Game Simulation**
   https://unity.com/case-study/death-carnival#balancing-complex-weaponry-takes-substantial-resources

3. **Learning the Patterns of Balance in a Multi-Player Shooter Game**
   http://antoniosliapis.com/papers/learning_the_patterns_of_balance_in_a_multi-player_shooter_game.pdf

4. **Evolving Interesting Maps for a First Person Shooter**
   http://julian.togelius.com/Cardamone2011Evolving.pdf

5. **Automated Playtesting with Procedural Personas through MCTS with Evolved Heuristics**
   https://arxiv.org/pdf/1802.06881.pdf

6. **The Rational Design Handbook: An Intro to RLD**
   https://www.gamasutra.com/blogs/LukeMcMillan/20130806/197147/The_Rational_Design_Handbook_An_Intro_to_RLD.php

7. **Automated Testing at Scale in Sea of Thieves | Unreal Fest Europe 2019 | Unreal Engine**
   https://www.unrealengine.com/en-US/events/unreal-fest-europe-2019/automated-testing-at-scale-in-sea-of-thieves

8. **Automation System Overview**
   https://docs.unrealengine.com/en-US/TestingAndOptimization/Automation/index.html

9. **Automated Testing For League Of Legends**
   https://technology.riotgames.com/news/automated-testing-league-legends

10. **Level Design Help Files**
    https://www.mapcore.org/topic/3598-level-design-help-files/

11. **LEVEL-DESIGN.org Knowledge Base**
    http://level-design.org/?page_id=2468

12. **Single-Player Mapping Tips**
    https://developer.valvesoftware.com/wiki/Single-Player_Mapping_Tips

13. **A* Pathfinding Project**
    https://arongranberg.com/astar/

14. **NodeCanvas Documentation**
    https://nodecanvas.paradoxnotion.com/documentation/

15. **Unity Machine Learning Agents**
    https://unity.com/products/machine-learning-agents