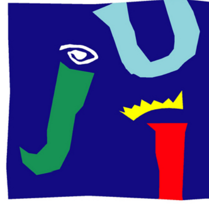


Application of Wave Function Collapse Algorithm for Procedural Content Generation in Video Games

Adrián Ramos Boira

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I
May 25, 2022
Course 2021/2022

Supervised by: Miguel Chover Selles

To my mum, Tali,
who was always ready to comfort me
in my lowest moments in the degree when everything seems
imposible. Finally, I follow her advice and it seems that
I have been able to finish, we will see what provides the real world.

***

To my dad, Jose,
who was always trying to help me when
I could not find the solution to a problem, despite that he have
no idea about anything related to programming
and he always tried to cheer me up during the degree. Lets see if I finally
could continue with this video games career and all the suffering was worth it.

***

To my sister, Laura,
who have always been a good example of achieving her goals,
I could not get anything that I got without her. And the most important
She always reminded me to make prints to debug a code,
best coding advice ever. Lets see if I could keep following her steps
and get a job where I could explode everything that I have learned.

ACKNOWLEDGEMENTS

First of all, I would like to thank my Final Degree Work supervisor, **Miguel Chover Selles** for his help during my years in the University first of all, and then for his help before the beginning of the development of this project, which was crucial in helping me focus on this topic which seemed unreachable when we started. As well as his advice to aim the project in the direction that finally I followed.

I also would like to thank **José Martinez Sotoca** for his helping hand and his support during the project since the early beginning, giving **Miguel** and me his complex and unique point of view. He was a source of wisdom and crazy ideas which allowed us to find easy ways to develop the algorithm that finally gave us the result that we wanted.

ABSTRACT

This document constitutes the project report of the Video Games Design and Development Degree final project by **Adrián Ramos Boira**. This work is related to the procedural generation of content using the Wave Function Collapse (WFC) algorithm. This algorithm is used to build an environment following a set of constraints that ensures features such as good playability, connectivity among different zones and an appealing style in the level construction without losing the randomness of the procedural creation. Furthermore, the environment built with the WFC algorithm will be included in a simple game, named *World Fighters Coliseum*, which we have used to demonstrate the functionality of all the generated content. On the other hand, in this project we are going to explore other functionalities of the algorithm applying and adapting it to AI, such as building an NPC's behaviour structure in a procedural way.

CONTENTS

Contents v

1 Introduction **1**
1.1. Work Motivation 1
1.2. Objectives 2
1.3. Environment and Initial State 2

2 Planning and Resources Evaluation **5**
2.1 Planning 5
2.2 Resource Evaluation 8

3 System Analysis and Design **9**
3.1 Requirement Analysis 10
3.2 System Design 14
3.3 System Architecture 21
3.4 Interface Design 21

4 Work Development and Results **25**
4.1 Modelling. 25
4.2. Procedural level generation 28
4.3. Procedural AI structures generation 37
4.4. Game Environment 57
4.5 Results 61

5 Conclusions and Future Work **64**
5.1 Conclusions 64
5.2 Future work 65

Bibliography **66**

List of figures **68**

INTRODUCTION

Contents

1.1. Work Motivation 1
1.2. Objectives 2
1.3. Environment and Initial State 2

This chapter shows what was the motivation of the project in the beginning and how it is going to be developed, putting the emphasis on which are the main objectives of it and why these objectives were chosen.

1.1. Work Motivation

The video games industry is one of the most profitable industries in the world nowadays, an industry which moves so much money needs to evolve each day so there is a lot of money invested in the investigation of new methods that boost the video games development to jump to a new generation. Some fields that are being investigated are related with the procedural generation of content for video games, an example of this procedural generation is *Minecraft* where the terrain building is a key part of the experience of replaying the game infinitely. Another example is *Bad North* which is an indie game where the procedural generation of islands is a key element in the gameplay because of its mechanics. And like these examples, there are many other video games which use algorithms to produce their content, making each player experience different and even each play.

Therefore, we investigated until we found the Wave Function Collapse (WFC for short) algorithm and we designed a project where this algorithm was applied to procedural generation.

However, this final degree project was conceived not to be a project about how an algorithm works, what we wanted is that this algorithm could be understood by the people, modify its working pieces and watch how it changes the results. So we decided to create a game where a key part of it would be modifying some parameters of the algorithm and trying the content to check if it is functional for the game.

1.2. Objectives

Based on the motivations of the work, there are some goals that we are interested in achieve:

- Understand how the Wave Function Collapse algorithm works.
- Implement the Wave Function Collapse algorithm in the Unity environment.
- Use the Wave Function Collapse algorithm to create procedural levels.
- Explore the possibilities that the Wave Function Collapse algorithm could have in the procedural generation of behaviours.
- Use the Wave Function Collapse algorithm to create behaviour structures for the NPCs
- Create a demo where we can test the correct functioning of all the content created by the WFC algorithm.

1.3. Environment and Initial State

As it was mentioned above, there are a lot of studios that are developing video games in which some of their content was procedural generated, a good example are those games whose terrain could be regenerated infinitely and it was different in each play. So, when we found that there is an investigation topic that is related to the procedural generation of content, this topic caught our attention and we decided to start investigating it. Then, we started to look deeper and deeper and we found some information about the projects that the people had been doing to understand how this kind of algorithm works.

Thus, we searched for some more information and examples of the Wave Function Collapse algorithm application and the investigation started to look clearer. We found two really important developers who have worked hard to explain and demonstrate the possibilities that this algorithm has and they are **Oskar Stålberg** and **Boris the Brave**. As we were gathering information about Wave Function Collapse, we came across more and more examples created by **Oskar Stålberg** [5, 6, 7, 8] which show really clearly how the algorithm works from the inside. His little projects were linked in the official github of the Wave Function Collapse algorithm [27], created by one of the first persons that apply the algorithm to a video games

field, **Maxim Gumin** [28], and where many project related with the algorithm are linked to make easier that the people can access to extra information about it. Those projects made by **Oskar Stålberg** were our first contact with some content created by the WFC. When we watched those examples and we understood how they were working, we were ready to start figuring out the entire project.

The best project made by **Oskar** to understand the functioning of the WFC, it is one where from a sixteen tiles tileset, the algorithm can build uncountable tilemaps all in a 2D world [1]. In this project, the user can slow down the execution and watch how the algorithm decides which tiles are the ones which finally are set in the tilemap, so it is really useful to understand the algorithm functioning. He also has some 3D projects which show the real power of the algorithm, so when we saw these projects, we thought that could be a good first step in the project to implement the Wave Function Collapse algorithm in a 3D environment. This initial idea, along with the 3D projects of **Oskar** as example, ended up becoming a video game where the player could watch how the algorithm works and test if the resulting content was good enough for allowing them to play following some rules and using different mechanics, demonstrating that the algorithm is good to create video games content.

We already had the first stones to build the foundation of the project, we had the idea of what we wanted to do, we had a bunch of examples to understand the algorithm and we had some articles and papers where we could search for information. So, with all these aspects we were sure that making a project about procedural level generation was possible, but we had the possibility to innovate if we tried to apply the WFC algorithm to another field where it had been less used, so we thought about applying the algorithm to the AI field. So in this project we have had the opportunity of creating a tool to make this algorithm useful for more video games development fields than just the terrain generation. Therefore, we started to implement the algorithm to create the level generation tool, so we needed more information about the WFC and how it can be translated to code, and here is where the next important person appears, **Boris the Brave**.

We started reading the **Boris the Brave** blog, it was a blog where he explain the functioning of the WFC algorithm for dummies [2], so it was perfect to start getting the key aspects of the algorithm and we could start thinking how we could implement such a complicated algorithm in a scripting language that we could know and make it work properly.

Once we were sure that we had a good point to start with the algorithm, we started to design the way that an algorithm for terrain generation could be applied to the AI field, so we investigated a little more about AI structures and behaviours' definition. So, we realised that if the algorithm can generate terrain as a matrix of tiles, it also could generate a matrix of behaviours and use this matrix to build a decision tree. Then, we realised that applying the algorithm to the AI field meant that in the final game this content had to be tested by the player too, so in that moment we had to rethink the game not only to let the player modify the WFC algorithm to generate the terrain but also to generate the NPCs behaviour and then try all this content in a entertaining game.

And this is how we ended up starting the development of the project. With clear ideas about how we wanted to introduce the Wave Function Collapse algorithm to generate the terrain and the enemies behaviours' structures, and also develop some gameplay mechanics that let the player check the correct functioning of this content in a real game. During the development of the project we would be able to find support in the incredible community of the Unity Asset Store, and thanks to the assets and models that they offer, we could focus our efforts on the programming of the algorithm and the game mechanics.

CHAPTER 2

PLANNING AND RESOURCES EVALUATION

Contents

2.1. Planning 5
2.2. Resources Evaluation 8

This is the chapter with the most technical information. In this chapter we are going to explain how we have planned the development of the project and the resources that we have needed to use to accomplish the project.

2.1. Planning

Planning is a key aspect of a project development if we want to finish the project without dying in the attempt. In this section it is detailed the tasks that form the entire project and how it was decided to divide the work time among them. The planning is the following:

- **Task 1 (20 hours):** Search information about how the WFC works and which kind of content can be generated.

- **Task 2 (10 hours):** Design the tiles which form the tileset and check if they are appropriate to be used with this algorithm.

- **Task 3 (20 hours):** Implement the WFC algorithm in Unity3D. It involves thinking which is the best way to express the ideas of the algorithm in C# and making it functional inside the Unity engine.

- **Task 4 (25 hours):** Design how to adapt the algorithm to the tiles generation process and use it to build procedural terrain which could be functional for a game.

- **Task 5 (25 hours):** Design how to adapt the algorithm to the behaviour generation process and use it to build the decision structure that allows the NPCs to demonstrate a functional behaviour in a game.

- **Task 6 (30 hours):** Design a real game where the player can modify some aspects of the WFC algorithm and test the content procedurally generated, bearing in mind that the gameplay experience has to be entertaining for the player.

- **Task 7 (20 hours):** Test if the joining together of the terrain generation, the behaviour generation and the actual game work fine and creates a playable experience.

- **Task 8 (40 hours):** Solve the bugs that would be found during the development of the project and in the testing task.

- **Task 9 (70 hours):** Write and correct the final report of the project and all the documents that are needed during the project.

- **Task 10 (40 hours):** Create the slides and prepare the final presentation of the entire project.

Total of hour: 300.

The summary of these tasks execution is represented by a Gantt's chart which shows the final order followed to work in the tasks and the final time that have been invested in each one (see Figure 2.1):

	Assigned	Progress
TFGProject		97%
Setting of the Project		100%
Search information about WFC algorithm	Adrián Ramos Boira	100%
Technical Proposal Delivery	Adrián Ramos Boira	☑
Design tiles of the tileSet	Adrián Ramos Boira	100%
Implement WFC algorithm in Unity3D	Adrián Ramos Boira	100%
Game Development		100%
Environment generation by WFC	Adrián Ramos Boira	100%
Behavior generation by WFC	Adrián Ramos Boira	100%
Gameplay mechanics, aesthetic, etc	Adrián Ramos Boira	100%
GDD Document Delivery	Adrián Ramos Boira	☑
Game Testing and Bug Solving		100%
Testing the game features	Adrián Ramos Boira	100%
Bug solving	Adrián Ramos Boira	100%
Document Writing		92%
Write the Technical Proposal	Adrián Ramos Boira	100%
Write the GDD document	Adrián Ramos Boira	100%
Write the Final Project Report	Adrián Ramos Boira	100%
Final Report Tutor's Delivery	Adrián Ramos Boira	☑
Final Report Delivery	Adrián Ramos Boira	☑
Write the Project Presentation	Adrián Ramos Boira	80%
Project Presentation Delivery		☐

JAN 2022 FEB 2022
9 16 23 30 6 13 20 27

MAR 2022 APR 2022 MAY 2022
27 6 13 20 27 3 10 17 24 1 8 15 22 29 5

Figure 2.1: DFP task planner (made with TeamGantt [3])

2.2. Resources Evaluation

This point contains a list of all the hardware and the software that have been used to accomplish the project, not all of them are essential, similar products can be efficient too. This is the list of tools:

- **MSI GF65 Thin 9SEXR** (i7 CPU, 16 GB RAM and GPU 2060 RTX 6GB): We have used this pc as our main work platform.

- **Unity 3D 2021.1.23f1**: The engine used to develop the game.

- **Visual Studio 2019:** We used it attached with the Unity engine. It is a really useful coding tool which integrates all the methods that are needed to handle all the actions in the game. It uses the C# language which is a big pro because it is the language that is more used during the degree.

- **GitHub (GitHub Desktop):** It is the tool through which we upload our project versions to an online repository. It is really useful to avoid losing progress of the project and it provides backup versions if something goes wrong.

- **Krita 4.2.9:** It is a free design tool that we used to create the tile meshes and all the GUI aspects and menus of the game.

- **Adobe Premiere Pro CC 2020:** We used it to edit the videos that are included in the game.

- **TeamGantt:** It is a tool to make Gantt charts in an easy way and we used it to organise our workflow and control when the delivery dates were close.

- **Unity Asset Store:** It is the official store site of Unity where we found the models that we used for some aspects in the game such as the character and the environment.

- **Mixamo:** It is a free website which provides the users with good looking animations that can be applied to 3D models and that are compatible with the Unity engine. The most interesting part of this tool is that everyone can upload their model, apply any animation and then download it for free.

- **Lucidchart:** It is an online free tool that provides the user with a lot of tools to design their diagrams, we have used it to design the case use and activity diagram for this project.

CHAPTER 3

SYSTEM ANALYSIS AND DESIGN

Contents

3.1. Requirement Analysis 10
3.2. System Design 14
3.3. System Architecture 21
3.4. Interface Design 21

This chapter deals with the description of the requirement analysis, the system design and the system architecture of the game, here we are going to explain all the main aspects related with the game mechanics, the game flow and the possibilities of the player inside the game. Moreover, this chapter shows how the game user interface is and the information that it provides to the player.

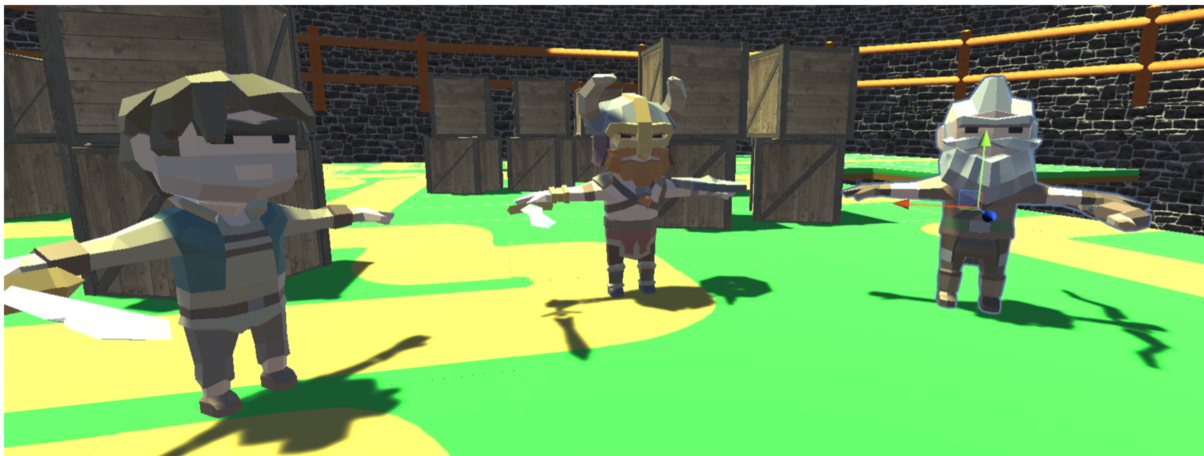


Figure 3.1: Playable and Non-Playable Characters

3.1. Requirement Analysis

Making a game from the beginning can be seen as a problem to solve. To start figuring out how the problem can be handled is really important to do a preliminary analysis of the requirements that the final product must have.

The game we are going to design to explode the power of the WFC algorithm is an arcade gladiator combat game. Maybe, this description is too generic, so it is difficult to deduce the game requirements from it. A deeper description surely helps us to work out better the functional and nonfunctional requirements for the game.

World Fighters Coliseum is, as said, an action combat game inside a gladiators' arena. Therefore, the player will have to face the enemies inside the arena until all the enemies are dead or the player fails and dies (see Figure 3.1).

As soon as the game is started, the player sees the main menu (Figure 3.2) where there are 3 options: *Start, Tutorial* and *Exit*. Selecting the last option the game will be quitted. Pressing the *Tutorial* button a new scene appears where the player can try the main mechanics of the game and practise with the character's controls. And, finally, selecting the *Start* button the game starts.

Figure 3.2: Main Menu of the game

In this game, the mechanics are a key factor and they have been thought to be simple and easy to understand. In addition, the mechanics are really important for trying the performance of the WFC algorithm in AI generation, which is one of the main contributions of the project.

Once inside the game, all the action is going to be performed in an arena where the player has to face all the difficulties that they find. Inside this arena there will be a random number of enemies which could be archers or swordsmen, so each type will behave differently to confront the actions of the player. Furthermore, the arena is going to be a kind of colosseum where there will be a moat limiting the fight zone and some features such as barrels and crates which can be used by the player to get some cover and hide from the enemies. Before entering the arena, the player will be in a separate room where they can select the characteristics they want that the enemies and terrain have. These characteristics will be used as parameters for the WFC algorithm, this allows the player to understand how the algorithm decides when it is generating new content.

The main actions that the player is able to do with the main character are represented by the following list, we have decided to differentiate two types of mechanics related to the player's actions (see Figure 3.3):

The first category is related to the actions that the player can do during the action phase of the game. So, the basic actions are the possibility of moving over the XZ plane, using the WASD keys, and rotating around the Y axis, using the mouse. These actions let the player move around the game map, explore the combat zone, seek or run away from the enemies. Moreover, the player can attack, they are able to fight with their sword, using the left mouse click, and block with the sword the enemies' attacks, using the right mouse click.

The second category is related to the actions that the player can do but these actions are focused on the functioning of the WFC algorithm to build the terrain and to generate new decision trees for the NPCs, all these actions are limited to the starting room before entering the arena. So, in the terrain generation field, the player has the possibility to choose which tiles are used to create the terrain. However the behaviour generation seems to be really different from the terrain building, the way that WFC face these two processes are really comparable, so their parameters selection is also similar. Thus, in the behaviour generation field, the player has the possibility to choose which actions are used to build the decision tree of the NPC. This selection, using the left mouse click, is going to be done through activating or deactivating buttons which the player is going to interact with to minimise the complexity of the parameter selection.

Figure 3.3: The main character model

On the other hand, we can say that it is a simple game but it lets us explore different combinations of behaviours for the NPCs inside a simple and controlled zone which makes the algorithm testing an easier task. As has been said, there are two types of enemies, archers and swordsmen, so, in the game, there are some shared mechanics among the both types, and some specialised mechanics for each one. To differentiate between both types of enemies, each one has a different model and a different weapon (see Figures 3.4 and 3.5). The most important mechanics are the following:

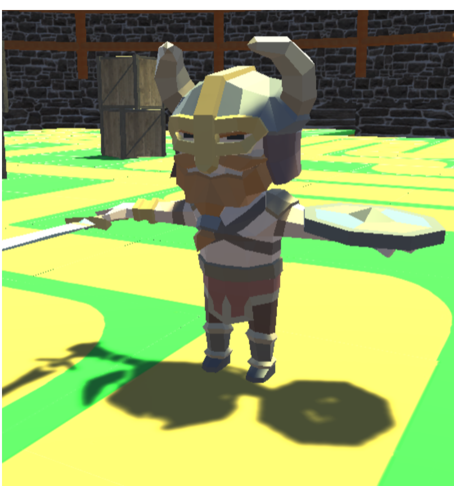


Figure 3.4: Face to face enemy

Figure 3.5: Long distance enemy

These mechanics are all actions which the enemies can do to face the player and try to create complex strategies. There are two types of enemies: long-distance enemies, archers, and

face-to-face enemies, swordsmen. The shared behaviours are a stand-still behaviour and a patrol behaviour needed to watch some zones in the arena. In addition, both types have the behaviour of retreat inside a more fortificated position and go to strategic locations. Moreover, face-to-face enemies have some different behaviour than long-distance ones. The first ones have the behaviour of getting close to the player and attack with the sword, the second ones have the behaviour of getting away from the player, aiming and shooting them.

3.1.1 Functional Requirements

A functional requirement represents a function of the system that some agent is going to accomplish during the play. This is the list of the functional requirements of the game:

R1: The player can quit the game by pressing the Quit button.
R2: The player can start the game by pressing the Start button.
R3: The player can enter the tutorial by pressing the Tutorial button.
R4: The player is capable of moving all around the map using WASD keys.
R5: The player can activate buttons, using the left mouse click, inside the initial room.
R6: The player can deactivate buttons, using the left mouse click, inside the initial room.
R7: The player can attack the enemies with the sword, using the left mouse click, inside the arena.
R8: The player can block the enemies attacks with the sword, using the right mouse click.
R9: The player can choose which tiles are used for building the tile map.
R10: The player can choose which basic behaviours constitute the decision tree of the NPCs.
R11: The enemies are able to retreat to a fortificate area.
R12: The enemies are able to hide in different strategic positions.
R13: The enemies can attack the player when they detect them.
R14: The enemies (only the swordsmen) can get close to the position of the player.
R15: The enemies (only the archers) can get away from the player position.

3.1.2. Non-functional Requirements

Non-functional requirements are requirements that impose restrictions in the game, such as the aesthetic, the GUI, the map limitations and so on. Commonly these restrictions are related with the world design, the interface and the possibilities inside the game.

R16: The aesthetic must be low-poly, simple and clean.
R17: The UI must be simple, easy to read and discreet, not obstructing the player's sight.
R18: The control must be practical, a bit slow but responsive.
R19: The enemies must have an individualist behaviour but coordinated with their partners.

3.2. System Design

This section presents the logical and operational design of the system. This design is represented with a case of use diagram (see Figure 3.6) and the game flow inside the game is represented with an activity diagram (see Figure 3.7). And the description of each case is listed in the following tables:

Requirements:	R1
Actor:	Player
Description:	At the main menu scene the player have the possibility to select the Quit button to exit the game.
Preconditions:	1. The player is at the main menu scene.
Steps normal sequence:	1. The player selects the quit button. 2. The game is closed.
Alternative sequence:	[1.1] The player selects any other button in the menu.

Table 3.1: Case of use “CU01. Quit Game”

Requirements:	R2
Actor:	Player
Description:	At the main menu scene the player have the possibility to select the Play button to enter the game.
Preconditions:	1. The player is at the main menu scene.
Steps normal sequence:	1. The player selects the play button. 2. The game world is loaded. 3. Control is returned to the player.
Alternative sequence:	[1.1] The player selects any other button in the menu.

Table 3.2: Case of use “CU02. Play Game”

Requirements:	R4
Actor:	Player
Description:	The player is capable of moving around the map using the WASD key.
Preconditions:	1. The player must have started the game.
Steps normal sequence:	1. The player presses one of the movement keys. 2. The game moves the player according to the key that is chosen.
Alternative sequence:	[1.1] The player selects any other key. [1.2] The player is trying to move toward an obstacle.

Table 3.3: Case of use “CU03. Move Character”

Requirements:	R5
Actor:	Player
Description:	Inside the game and in the initial room the player can interact with buttons to activate them, that let the player choose which tile and behaviours will be available for the WFC algorithm.
Preconditions:	The game has started and the player is inside the initial room.
Steps normal sequence:	1. The player aims for the button. 2. The player clicks the left mouse button. 3. The player activates the button.
Alternative sequence:	[1.1] The player decides not to aim for the button. [2.1] The player clicked any other button.

Table 3.4: Case of use “CU04. Activate buttons”

Requirements:	R6
Actor:	Player
Description:	Inside the game and in the initial room the player can interact with buttons to deactivate buttons, that let the player choose which tile and behaviours will not be available for the WFC algorithm.
Preconditions:	The game has started and the player is inside the initial room.
Steps normal sequence:	1. The player aims for the button. 2. The player clicks the left mouse button. 3. The player deactivates the button.
Alternative sequence:	[1.1] The player decides not to aim for the button. [2.1] The player clicked any other button.

Table 3.5: Case of use “CU05. Deactivate buttons”

Requirements:	R7
Actor:	Player
Description:	Inside the game and in the arena the player can attack to the enemies.
Preconditions:	The game has started and the player is inside the arena.
Steps normal sequence:	1. The player aims for an enemy. 2. The player clicks the left mouse button. 3. The player attacks the enemy.
Alternative sequence:	[1.1] The player is not aiming at the enemy. [2.1] The player clicked any other button.

Table 3.6: Case of use “CU06. Attack to the enemies”

Requirements:	R8
Actor:	Player
Description:	Inside the game and in the arena the player can block with the sword the enemies' attacks.
Preconditions:	The game has started, the player is in the arena and under an enemy's attack.
Steps normal sequence:	1. An enemy attacks the player. 2. The player clicks the right mouse button. 3. The player blocks the enemy's attack.
Alternative sequence:	[1.1] The enemy fails the attack. [2.1] The player clicked any other button.

Table 3.7: Case of use "CU07. Block the enemies' attack"

Requirements:	R3
Actor:	Player
Description:	Inside the main menu, the player can select the Tutorial tab.
Preconditions:	The player has to be in the main menu scene.
Steps normal sequence:	1. The player selects the Tutorial button. 2. The tutorial scene is loaded. 3. The control is returned to the player.
Alternative sequence:	[1.1] The player can select another tab of the menu.

Table 3.8: Case of use "CU08. Select Tutorial button"

Requirements:	R9
Actor:	Player
Description:	Inside the game and in the initial room, the player can select which tiles the WFC algorithm will use to generate the terrain.
Preconditions:	The game has started and the player is inside the initial room.
Steps normal sequence:	1. The player activates a certain button. 2. The related tile is considered by the algorithm. 3. The algorithm uses it to generate the terrain.
Alternative sequence:	[1.1] The player deactivates this button. [3.1] This tile is not used during the generation.

Table 3.9: Case of use “CU09. Select a tile for the tile set”

Requirements:	R10
Actor:	Player
Description:	Inside the game and in the initial room, the player can select which behaviour blocks are considered in the block set which the WFC algorithm used to generate the NPCs’ decision trees.
Preconditions:	The game has started and the player is inside the initial room.
Steps normal sequence:	1. The player activates a certain button. 2. The related block is considered by the algorithm. 3. The algorithm uses it to generate a NPC’s decision tree.
Alternative sequence:	[1.1] The player deactivates this button. [3.1] This block is not used during the generation.

Table 3.10: Case of use “CU10. Select a block for the block set”

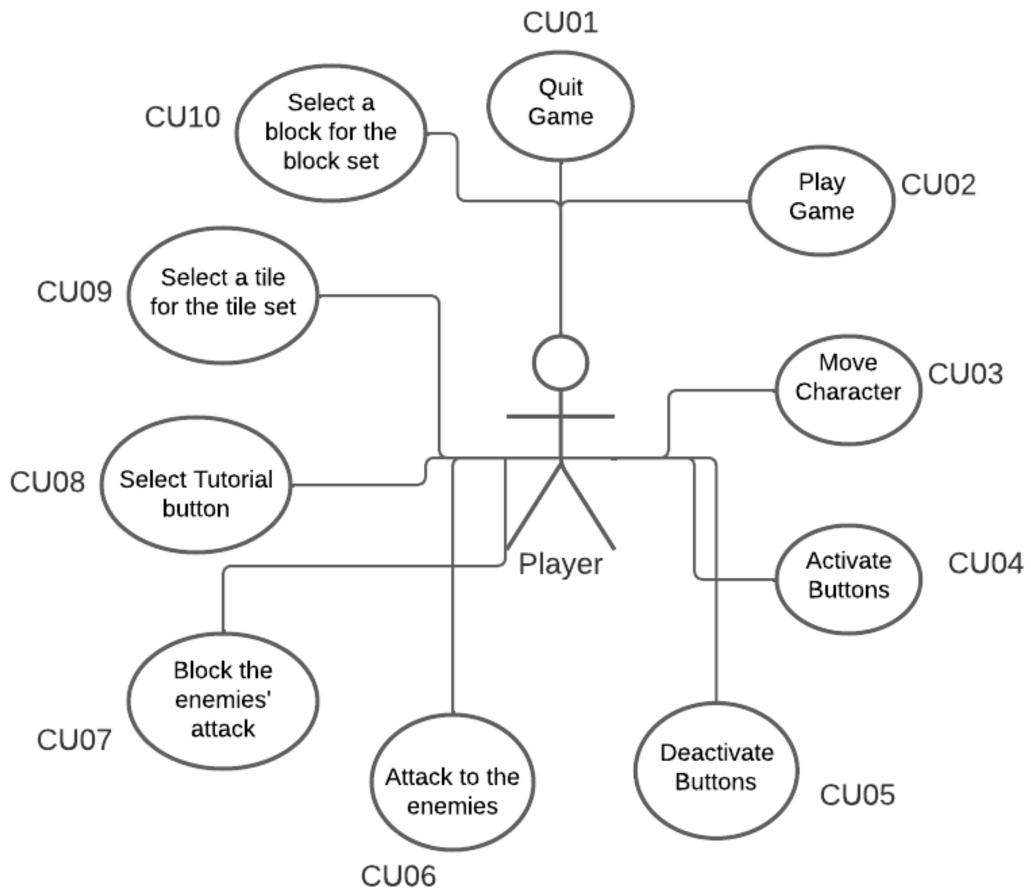
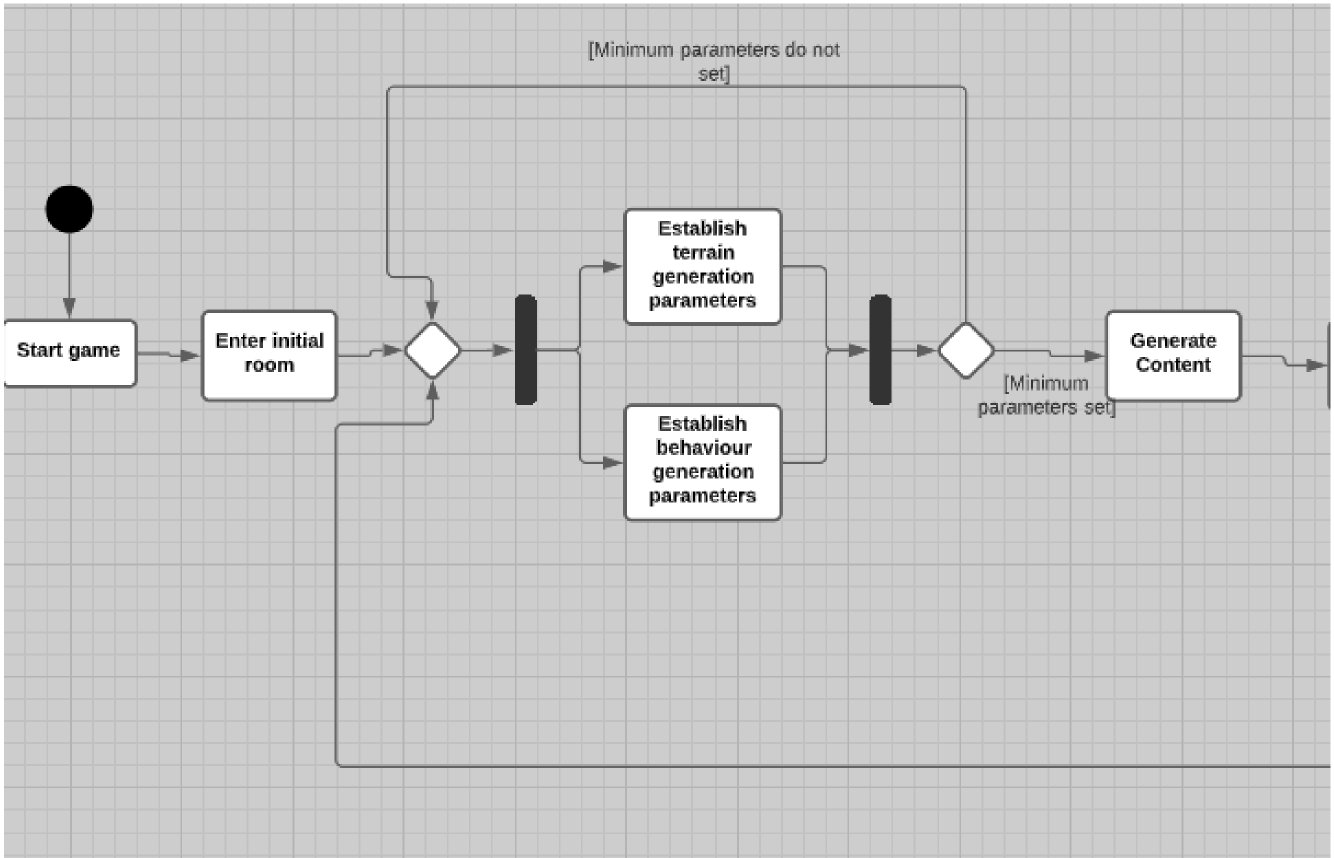


Figure 3.6: Case use diagram (made with https://www.lucidchart.com/pages/es)



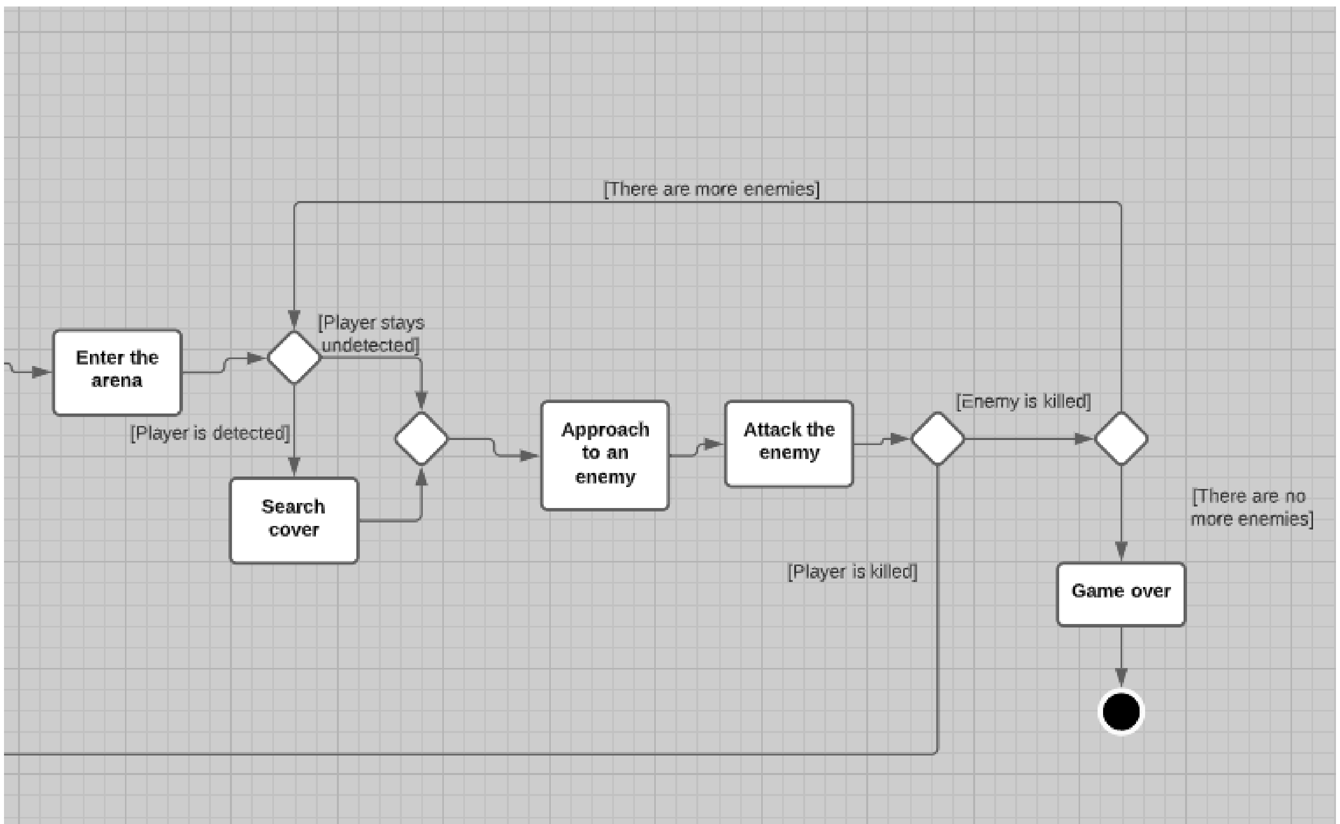


Figure 3.7: Activity diagram (made with https://www.lucidchart.com/pages/es)

In the activity diagram it shows the game flow that a regular game can have. The normal evolution of the game play is the following: Firstly, the player starts the game, and they appear inside the initial room. Here, the tiles selection is done by the player, concurrently to the blocks selection. Then, we have a possible alternative, if the minimum number of tiles, or blocks, needed is not chosen then the player can not continue and they repeat the selection. If the parameters' number is the correct one, the player can continue with the normal execution of the game. Secondly, the player enters the arena, once inside we have two possibilities, the player can stay undetected, so they proceed as normal; or the player is detected and they have to find some cover. In both cases, the following step is getting close to an enemy and attacking them. Finally, here we have two possible paths: the player is defeated, so the game is restarted and the player appears inside the initial room again; or the player kills the enemy and keeps going to the next step. The last bifurcation is determined by the number of enemies that are still alive, if there are no more enemies alive, the game is over and the player have won, otherwise the player has to come back to the approaching phase of the game.

3.3. System Architecture

This section describes the architecture of the projected system. Which includes the hardware and software requirement for accomplishing the entire project. The video game is made with Unity3D engine, specifically with the 2021.1.23f1 version. For running games made with this engine the minimal specifications are:

- Operating system:
    - Windows 7 sp+1(64-bit version)
    - Mac OS X 10.2
    - Ubuntu 16.04
    - CentOS 7

- CPU: SSE2 instruction set support.

- GPU: Graphic card with DX9 (shader model 3.0) capabilities

- RAM: 8 GB ram memory.

This information is taken from this web [4], but the minimum requirements can vary a lot depending on the scale of the project, so could be some systems that satisfy the minimum requirements of Unity that could not run this project. The project has been developed and tested on a system with the following specifications:

- Operating System: Windows 10 (64-bits)

- CPU: Intel Core i7-9750H

- GPU: NVIDIA GeForce 2060 RTX
- RAM: 16GB memory

It is recommended to play the game using a mouse if playing on a laptop, because the controls are more comfortable and accurate.

3.4. Interface Design

The game user interface is designed to be simple, easy to understand and unobstructive to the view. We want that the GUI would be appealing for the player but trying that the design does not steal too much attention from the important artistic aspects of the game as the tiles used for the terrain or the models.

The interface tells the player some information related to the current state of the game. The player can know how much life they have left, this information appears on the upper left corner of the screen. Then, the player can also know which weapon they are holding in their hand, this information is supported with some representative sounds about what actions the character does (see Figures 3.8, 3.9 and 3.10). This information appears in the lower right corner of the screen. Here we can see some images that help to understand the GUI distribution better.

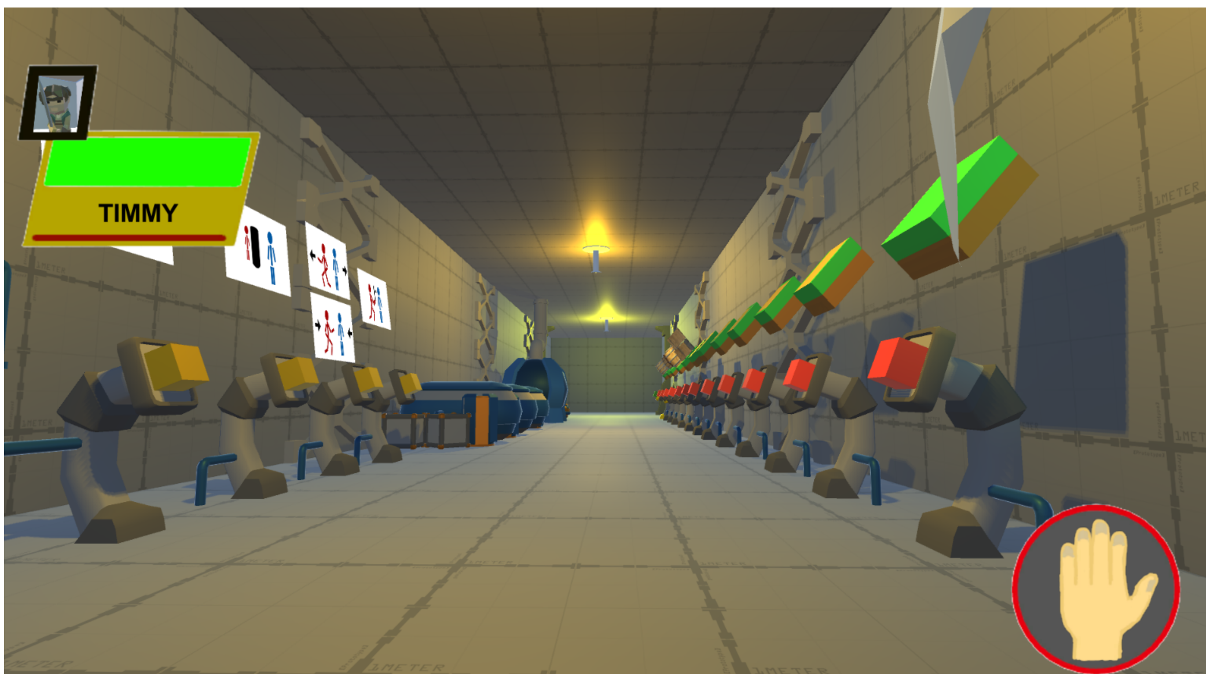


Figure 3.8: GUI in the initial room

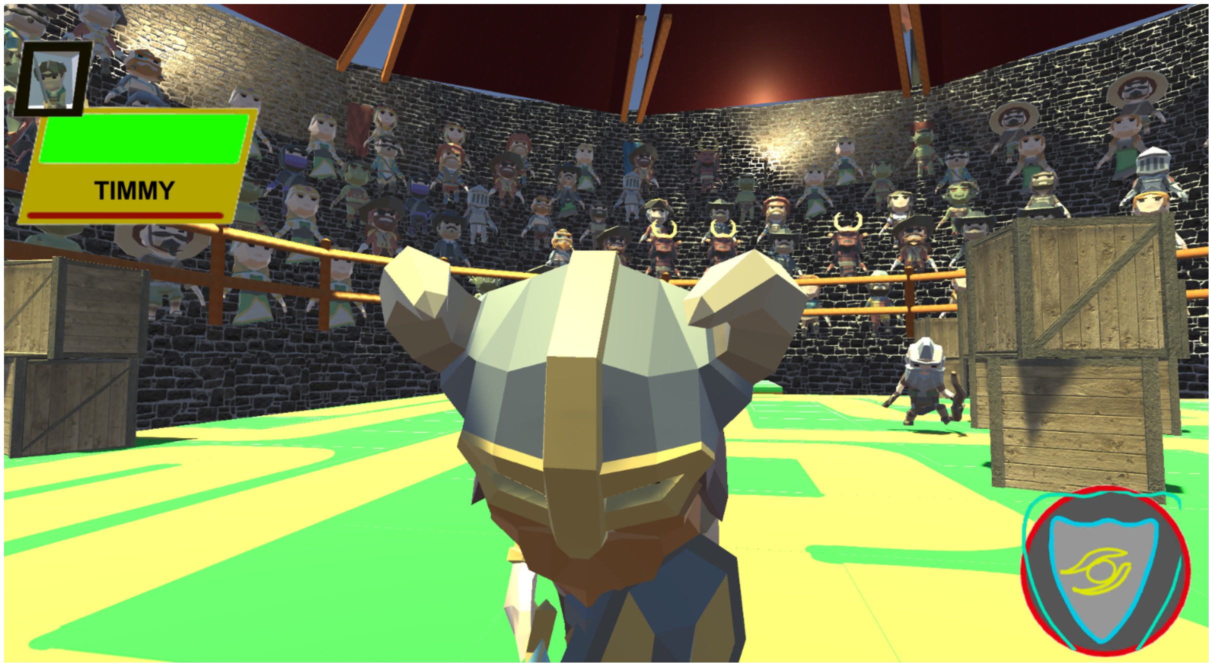


Figure 3.9: GUI in the middle of the arena (blocking action)

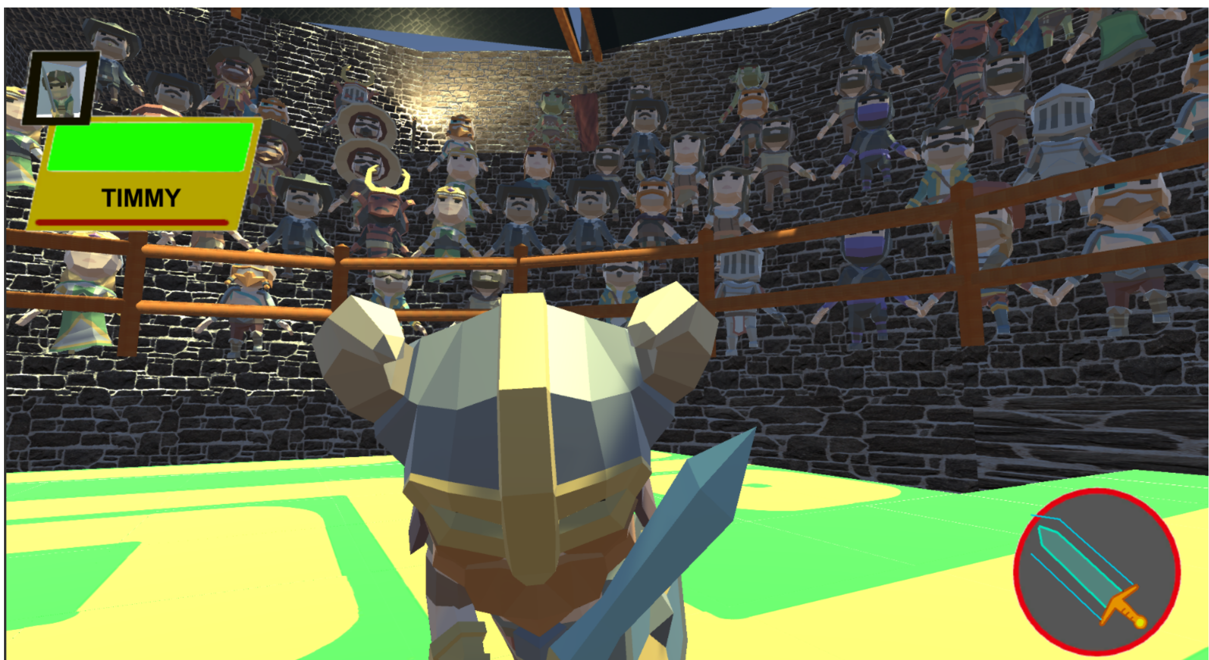


Figure 3.10: GUI in the middle of the arena (attack action)

CHAPTER 4

WORK DEVELOPMENT AND RESULTS

Contents

4.1. Modelling 25
4.2. Procedural level generation 28
4.3. Procedural AI structures generation 37
4.4. Game Environment 57
4.5. Results 61

After having a general idea about how the entire project has been developed, the requirements that we need to satisfy to carry on with the project, the main mechanics of the game and the main objectives of the project, we can start talking deeply about the most important aspects of the project development. So, we are talking about the actions that have been accomplished during the project development and why each decision was taken.

In this section, it will be discussed the decisions and the actions that have been taken during the development of the game. The information used to develop the work comes from scientific papers, blogs, experiences from other developers and Youtube videos. All these documents have been used to get a wide perspective of the WFC algorithm and how to implement it in a functional game. In the bibliography you would have the links to the articles, the examples and the videos used in this project.

4.1. Modelling

Before starting with all the code and implementation part, we have to differentiate the two ways that we are going to use the WFC algorithm, the first one is to generate procedural terrain, which were the first step on the implementation, the second one is to generate procedural behaviour structures for the NPCs, which implementation has been based on the terrain generator implementation. So, one of the main parts of the project is focused in the terrain building and the key pieces of an algorithm of terrain generation are the tiles that will

be the minimum unit of the tilemap. Therefore, it is needed that the designing of these tiles would be appropriate for the generation because it directly influences the functionality of the WFC algorithm. This algorithm works as good as the tiles we give it to work so it is a key factor in the project (see Figure 4.1).

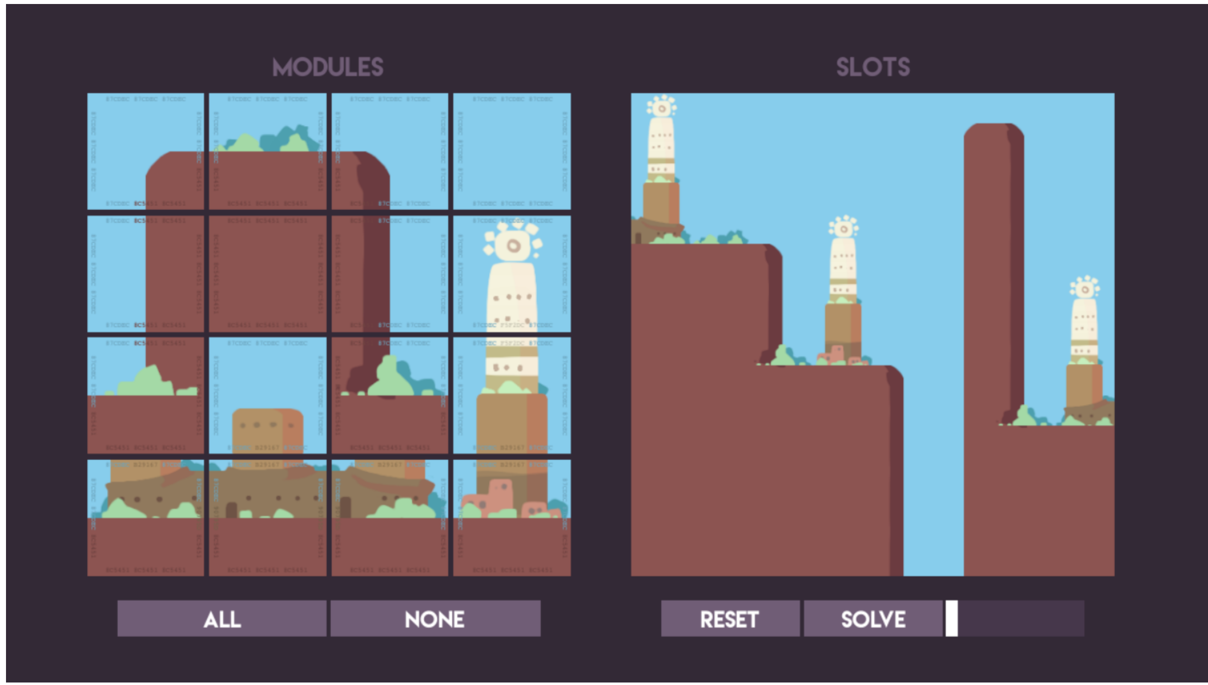


Figure 4.1: Image of 2D content created by WFC algorithm [1]

There are many ways to make a good tileset for getting a good efficiency from the algorithm, among all the strategies, using Marching Cubes is a good option because it checks if the corners are empty or solid and based on that information chooses which tiles can fit together. We are using a similar strategy to model and design our tileset, we are checking the sides of the tile leveraging that they have square shape because it is more visual and simple. So, we have created a tileset (see Figure 4.2) in which the tiles can fit together if they have in the touching sides the same material, for example, a tile with grass crossed by a straight road has to be touching a tile with grass in two of their opposite sides and a tile with road in the other two sides.

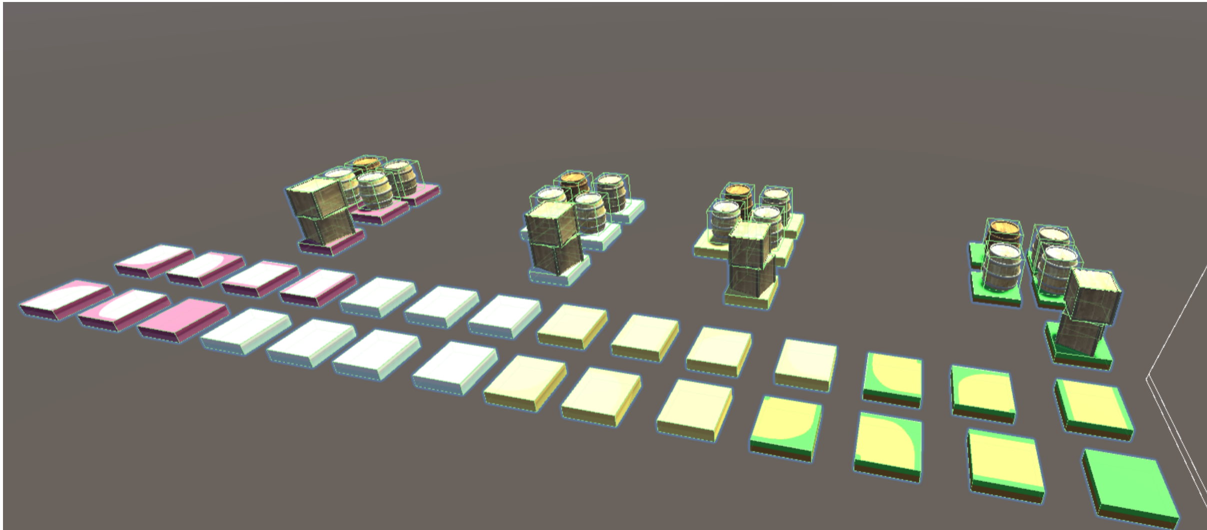


Figure 4.2: Appropriate tileset for terrain building with WFC

The terrain in this game has an important role not only because it is a key aspect for the mechanics and the strategy of the player but also because it is generated by WFC algorithm, which can be critical for the playability if the algorithm would not work well. In this game the terrain is built using tiles from our tileset, which has the characteristics that we have explained above. These tiles are set on a matrix that represents the entire arena environment, and they are taking up the positions of the matrix cells.

The tiles have to accomplish some constraints to be used in this algorithm. First of all, we need that the tiles have the same size as the cells of the matrix, they also must be the same size as the other tiles because if not, when they were set up, it will produce some critical problems and the environment will not be able to be used in a play. The most important characteristic in the tiles is that, bearing in mind the constraints, the algorithm always finds a combination of tiles that could be placed together to create the terrain. Despite that, the tiles can have any form or any stuff over as long as they have one of their sides compatible with a side of another tile in the tileset.

The rest of the art in the video game have less importance for the main objectives of the project, because it is related to the visual aspect of the minigame features and to the GUI of the game, so it has to be appealing but it can not interfere with the functionality of the algorithm or the game.

The artistic style of the game is influenced by the cartoon, we want to make the environment look simple and friendly, in addition, the colour palette is going to be a pastel palette with similar intensities and minimising the contrast in the environment. Because we want to achieve a children-playground atmosphere, we can make the game visually appealing and easy to understand which fits well with the way that we want to show the functionality of the WFC algorithm.

4.2. Procedural Level Generation

Once the tiles design is appropriated for the Wave Function Collapse algorithm, we had to start implementing the algorithm, so we based our work on the **Boris The Brave** [2] blog, and we had to start translating the text into C# structures. The translation started understanding which are the key elements in the algorithm and how they are related to each other because that will give us the key to represent it as classes and relationships among them. The key elements that WFC uses are: the variables which are the unknown elements that the algorithm has to establish, the domain that is all the possible elements that can take up the variable space and the constraints that are the rules that we want the final result to follow.

So, how can we implement the terrain generator? In our problem we are implementing the algorithm by creating a grid in which each cell represents a variable. Each variable has a domain which is formed by the tiles from the tileset which still respects the constraints that are applied to this variable, in the implementation the domain is a boolean array. Finally, we have to define what the constraints are. In our problem, we have decided that these rules are defined by the orientation of the roads that are represented in the tiles.

But we can not talk about the Wave Function Collapse algorithm without talking about entropy. In the original problem, the algorithm establishes an initial tile and propagates the restriction from this tile to the rest of variables, once the algorithm can not propagate the restrictions through more variables, then it uses the entropy to choose the next tile that could be established. So, in our problem the entropy has to be used in a similar way, because our first WFC implementation is used to build terrain.

Then, the entropy, in this algorithm, is a value that represents the probability to get to a stable situation, so taking a decision that have the lowest entropy value is more likely to end getting to this stable situation, and, we need to achieve this stability to build correctly a playable terrain.

Once we have more idea about what the entropy value really mean, it is time to get to the way that it could be calculated. After all, the entropy formula is $\sum_{i=0}^{n} p_i * log(p_i)$ being p_i the probability of each element in the set. So, it considers the probabilities of each element to be elected to determine the entropy value of the set. Therefore, in our problem the probability of each element is the weight of each tile divided by the total weight of the set, and the set is the tiles that are still available in a variable when the entropy is calculated, that is the domain.

The way that the WFC algorithm generates the terrain is a bit complicated but it is really visual. We establish one cell of the matrix as the starting point of the algorithm, there we set a tile randomly and that tile gives the first restrictions. Then, the algorithm moves through the neighbours of the current cell eliminating possibilities of them, and this process is repeated in each cell until all would be visited or a certain condition was accomplished. When the

algorithm finds that in some cell there is one possibility left then set the specific tile on this variable. This is the way that the complete arena is generated so when the algorithm finishes we are sure that all the tiles accomplish the constraints that the player has specified, so the arena is totally playable. We are able to generate really different scenarios using the WFC algorithm but changing the weights and the type of the tiles (see Figure 4.3).

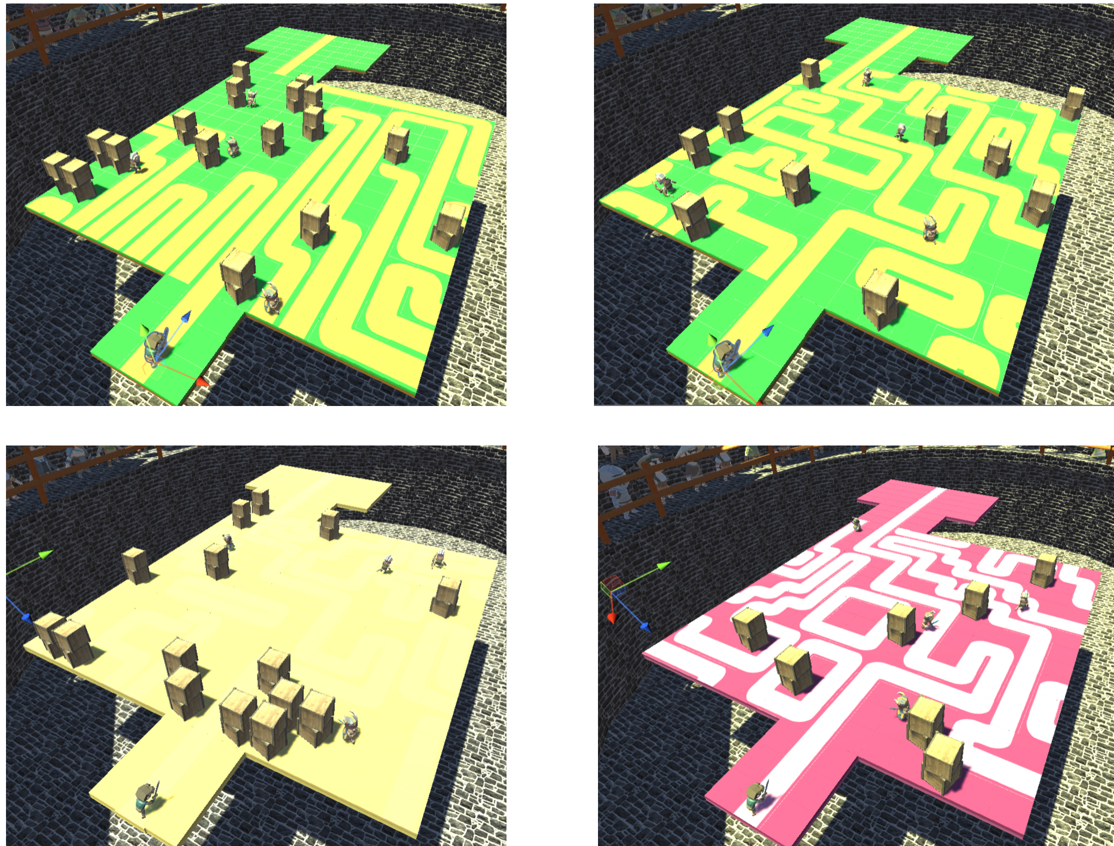


Figure 4.3: Different maps examples generated by the WFC algorithm

Now, that we have understand how the algorithm works and which are its key aspect for its well-functioning, we can start seeing how the basic elements can be implemented in C#. As I said, the minimum element in the WFC algorithm is the Variable, so this is how the Variable class is built (see Figure 4.4).

```
public class Variable : ScriptableObject
{
    public bool[] domain;
    public int domainCount;
    public int[] visited;
    public GridTile tileChosen;
    public float entropy;
    public GameObject tileReference;

    public void SetVariable(int size)
    {
        domain = new bool[size];
        entropy = 0f;
        for (int i = 0; i < size; i++)
        {
            domain[i] = true;
        }
        visited = new int[size];
        tileChosen = null;
        domainCount = size;
    }
```

Figure 4.4: Variable class and its constructor.

A little note before carrying on with the explanation. In the Variable class we need an integers array named visited, because when we are propagating the constraints it is possible that in the previous variable's domain there were more than one available tile, so we have to take the count about how many of those tiles can connect with the current variable's available tiles, and if one of the tiles can not, that tile have to be eliminated from the domain. However, if the previous variable's domain has only one available tile, if one tile from the current one domain can not connect, it automatically is out of the domain.

Now, we can talk about the other key element in the WFC algorithm, the entropy. All the variables need a method to calculate the entropy when the main process demands it (see Figure 4.5). The final entropy value only depends on the available tiles in the domain of the variable, so we have to check which tiles are still active and which ones are already eliminated.

At this point it is necessary to explain why we need weights for the tiles and why we choose a weight and not another. The weights are needed because we want that our terrain would be formed by all the tiles in the tileset but not all the tiles in the same proportion, because it would produce that the final terrain would be a meaningless mixture of tiles making it unappealing. So, the weights is a way to determine which tiles are more important than others and the algorithm is made to use the most important tiles more than the others. On the other hand, the weight chosen for a tile is not really important, what is important is the difference of weights among tiles. So, we need to differentiate really well the tiles by weight, when we want that in the final terrain some tiles would be clearly favoured. Besides, if we want that some tiles appear more or less with the same probability, we have to put similar weights independently which would be the number chosen.

```
public void CalculateEntropy(List<Tile> tileSet)
{
    float auxiliar = 0f;
    float maxWeight = 0f;

    for (int i = 0; i < tileSet.Count; i++)
    {
        if (domain[i])
            maxWeight += tileSet[i].weight;
    }

    for(int i = 0; i < domain.Length; i++)
    {
        if (domain[i])
        {
            auxiliar += (tileSet[i].weight/maxWeight) * Mathf.Log((tileSet[i].weight/maxWeight));
        }
    }

    entropy = auxiliar * (-1);
}
```

Figure 4.5: Variable's method to calculate its entropy.

The last method needed is SetTile, this method is used to choose which is going to be the final tile that fills this variable, the tile is passed as an argument and the real game object is created using the Instantiate() unity method, which lets the algorithm transform a tile reference to a real interactable object in the game world.

Following with the algorithm we can say that it has three well-defined parts: the initialization of the variables grid, the selection and setting of a tile in a variable and the constraint propagation from this tile. All these three parts are gathered in a class named *TileSetGenerator*. Here we are going to break down this class into four parts to make it simple to understand, first of all, the attributes declaration (see Figure 4.6), secondly the initialization of the generator, thirdly the tile election method and finally, the constraint propagation method.

```
public class TileSetGenerator : MonoBehaviour
{
    [Header("Terrain building variables")]
    public int numCol;
    public int numRow;
    public int maxSteps;
    public bool terrainGenerated = false;
    public Vector3 tileSize;
    public List<Tile> tileSet;
    public NavMeshSurface surface;
    public Transform predefinedPath;
    [HideInInspector] public Variable[,] grid;

    // Variables used to instantiate the Tiles
    Vector3 originalPos;
    Vector3 currentPos;
    bool gridCleared = true;

    // Variables used to handle with a predefined path inside the grid
    bool preDefinedPath = false;
    int[,] predefinedPathCoor;
```

Figure 4.6: TileSetGenerator attributes declaration.

Some clarifications about the attributes are needed. The *maxSteps* attribute is a kind of depth limit to avoid that the constraint propagation would be run through all the grid making the algorithm unaffordable in execution time. Moreover, the *surface* attribute is a reference to an object that lets the algorithm create a navMesh in execution time that could adapt to the terrain generated [9]. Then, the transform attribute named *predefinedPath* is a reference to an object that includes a pre-established path that is set in some of the cells of the grid. It represents some constraints that would change the terrain generation making that the final terrain would be adapted to this path. Finally, the *grid* attribute is the base of the algorithm, it is the object that the algorithm is going to run through deleting the tiles from the variables' domain, setting the final tiles in the appropriate positions and propagating the constraints to the neighbours.

Secondly, in the *TileSetGenerator* class we need an initialization method where the algorithm could initialise the grid to neutral values, creating the appropriate initial scene for the WFC. In addition, in this method we have added a way to adapt the initial state of the grid to the possibility that a predefined path already exists, so the WFC has to modify the initial state of some variables to adapt them to the constraints generated by this predefined path, making sure that the algorithm could build a functional terrain for the game (see Figure 4.7).

```
private void Initialize()
{
    originalPos = transform.position;
    currentPos = transform.position;
    grid = new Variable[numRow, numCol];

    for ( int i = 0; i < grid.GetLength(0); i++)
    {
        for (int j = 0; j < grid.GetLength(1); j++)
        {
            grid[i, j] = ScriptableObject.CreateInstance<Variable>();
            grid[i, j].SetVariable(tileSet.Count);
            grid[i, j].CalculateEntropy(tileSet);
        }
    }


    if (transform.childCount > 0 && predefinedPath.gameObject.activeSelf)
    {
        preDefinedPath = true;
        predefinedPathCoor = new int[predefinedPath.childCount, 2];
```

```
    for (int i = 0; i < predefinedPath.childCount; i++)
    {
        int[] coor = { Mathf.FloorToInt(predefinedPath.GetChild(i).position.z),
                Mathf.FloorToInt(predefinedPath.GetChild(i).position.x) };
        predefinedPathCoor[i, 0] = coor[0];
        predefinedPathCoor[i, 1] = coor[1];
        int choosenTile = -1;

        for (int k = 0; k < tileSet.Count; k++)
        {
            if (predefinedPath.GetChild(i).name.StartsWith(tileSet[k].tile.transform.name))
                choosenTile = k;
        }

        for (int j = 0; j < grid[coor[0], coor[1]].domain.Length; j++)
        {
            if (j != choosenTile)
                grid[coor[0], coor[1]].domain[j] = false;
        }

        grid[coor[0], coor[1]].visited[choosenTile]++;
        grid[coor[0], coor[1]].domainCount = 1;
        grid[coor[0], coor[1]].tileChosen = tileSet[choosenTile].tile;
        grid[coor[0], coor[1]].CalculateEntropy(tileSet);
        grid[coor[0], coor[1]].tileReference = predefinedPath.GetChild(i).gameObject;
    }
}
else
{
    preDefinedPath = false;
}
```

Figure 4.7: TileSetGenerator Initialize method.

To adapt the grid to the predefined path, this method, search which positions of the grid are taking up the tiles which form the predefined path, then in the corresponding variables it delete all the tiles from the domain except the tile which is set on the variable's position, then establish this tile as the final tile of the variable and recalculates the entropy of the variable.

The next important method for the initialization of the generator is the *Generate* method. This is a simple method because it is used to call the *Initialise* method and if there is a predefined path, this method carry on the first constraint propagation based on the tiles that form the path, this way the grid is adapted to the pre-existent path and all the errors that it could trigger are avoided, then the *TileElection* method is called, this method is going to be explained in the following section. However, if there is not a predefined path, the *Generate* method calls directly to the *TileElection* method and all the terrain building is carried out. After the generation the navMesh is created adapting its form to the terrain created (see Figure 4.8).

```
public void Generate()
{
    if (gridCleared)
    {
        terrainGenerated = true;
        gridCleared = false;
        Initialize(); // Initialize the grid to base values

        // If the predefined path is active, propagate the constraint generated by this tiles
        if (preDefinedPath)
        {
            for(int index = 0; index < predefinedPathCoor.GetLength(0); index++)
            {
                if (predefinedPathCoor[index, 1] > 0)
                    ConstraintPropagation(predefinedPathCoor[index, 0], predefinedPathCoor[index, 1] - 1, 3,
                     grid[predefinedPathCoor[index, 0], predefinedPathCoor[index, 1]], 0);
                if (predefinedPathCoor[index, 1] < (numCol - 1))
                    ConstraintPropagation(predefinedPathCoor[index, 0], predefinedPathCoor[index, 1] + 1, 1,
                    grid[predefinedPathCoor[index, 0], predefinedPathCoor[index, 1]], 0);
                if (predefinedPathCoor[index, 0] > 0)
                    ConstraintPropagation(predefinedPathCoor[index, 0] - 1, predefinedPathCoor[index, 1], 2,
                    grid[predefinedPathCoor[index, 0], predefinedPathCoor[index, 1]], 0);
                if (predefinedPathCoor[index, 0] < (numRow - 1))
                    ConstraintPropagation(predefinedPathCoor[index, 0] + 1, predefinedPathCoor[index, 1], 0,
                    grid[predefinedPathCoor[index, 0], predefinedPathCoor[index, 1]], 0);
            }

            int[] nextCel = SearchNextGridCell();
            if (nextCel[0] != 1 && nextCel[1] != 0)
                TileElection(nextCel[0], nextCel[1]); // Start  with the terrain generation
        }
        }
        else
        {
            var firstCell = SearchNextGridCell();
            TileElection(firstCell[0], firstCell[1]); // Start  with the terrain generation
        }

        if(surface != null)
            surface.BuildNavMesh();
    }
    else
    {
        Debug.Log("You need to clear the grid to generate a new landscape.");
    }
}
```

Figure 4.8: TileSetGenerator Generate method.

The next method that is going to be discussed is the *tileElection* method, this method needs as an argument the column index and the row index of the grid which represents the variable with the lower entropy value. Inside this method the algorithm make three different actions (see Figure 4.9): first of all, it determines the tile which is elected to take up the current variable’s position and instantiate this tile in the game world, secondly it propagates the constraints through the four edge-touching neighbours, finally, determines which is the next variable in the grid which is going to be analysed by the algorithm.

```
private void TileElection(int rowIndex, int colIndex)
{
    // Select the only tile available in the domain of the cell if there is only one
    // if not it makes a random election bearing in mind the weights of the tiles
    int chosenIndex = -1;
    if (grid[rowIndex, colIndex].tileChosen == null)
    {
        if (grid[rowIndex, colIndex].domainCount == 1)
        {
            for (int i = 0; i < grid[rowIndex, colIndex].domain.Length; i++)
            {
                if (grid[rowIndex, colIndex].domain[i])
                {
                    chosenIndex = i;
                    break;
                }
            }
        }
        else if (grid[rowIndex, colIndex].domainCount > 1)
        {
            Tile[] availableTiles = new Tile[grid[rowIndex, colIndex].domainCount];
            int j = 0;
            for (int i = 0; i < grid[rowIndex, colIndex].domain.Length; i++)
            {
                if (grid[rowIndex, colIndex].domain[i] && j < availableTiles.Length)
                {
                    availableTiles[j] = tileSet[i];
                    j++;
                }
            }
            chosenIndex = weightedRandom(availableTiles);
        }

        InstantiateTile(rowIndex, colIndex, chosenIndex);
    }

    // Propagates the constraints bearing in mind the current tile
    if(colIndex > 0)
        ConstraintPropagation(rowIndex, colIndex -1, 3, grid[rowIndex, colIndex], 0);
    if(colIndex < (numCol - 1))
        ConstraintPropagation(rowIndex, colIndex + 1, 1, grid[rowIndex, colIndex], 0);
    if(rowIndex > 0)
        ConstraintPropagation(rowIndex - 1, colIndex, 2, grid[rowIndex, colIndex], 0);
    if(rowIndex < (numRow - 1))
        ConstraintPropagation(rowIndex + 1, colIndex, 0, grid[rowIndex, colIndex], 0);

    // Search if any cel has one tile in its domain
    // If can't find any, then choose the one with minimal entropy
    int[] nextGridCell = SearchNextGridCell();

    if(nextGridCell[0] != -1 && nextGridCell[1] != -1)
        TileElection(nextGridCell[0], nextGridCell[1]);
}
```

Figure 4.9: TileSetGenerator TileElection method.

A little note to make the code simpler to be understood, inside the else statement where we check that the domain count would be greater than one, we are selecting one of the tiles that are available in the domain of this variable, because there are more than one tile that accomplish the constraints established by the neighbour variables. To make this selection, first of all we keep in an array the tiles that are still available and then we make a random selection among them biassed by its weights, the resulting tile will be the one chosen to be instantiated.

As we said before the tiles that are going to be analysed by the algorithm are chosen by its entropy value. So, inside the *SearchNextGridCell* method what the algorithm is doing is going through all the grid variables comparing its entropy value (see Figure 4.10). If it finds one that only has one tile in its domain, this variable would be the next one processed by WFC. But If it is not, the WFC algorithm will process the variable with the lowest entropy value.

```
public int[] SearchNextGridCell()
{
    float finalEntropy = Mathf.Infinity;
    int nextCol = -1;
    int nextRow = -1;

    for (int i = 0; i < grid.GetLength(0); i++)
    {
        for (int j = 0; j < grid.GetLength(1); j++)
        {
            if (grid[i, j].tileChosen == null && grid[i, j].domainCount == 1)
            {
                int[] nextGridCell = new int[2] { i, j };
                return nextGridCell;
            }
            else if (grid[i, j].domainCount > 1)
            {
                var entropy = grid[i, j].entropy;
                if (entropy < finalEntropy)
                {
                    finalEntropy = entropy;
                    nextRow = i;
                    nextCol = j;
                }
            }
        }
    }

    return new int[2] { nextRow, nextCol };
}
```

Figure 4.10: TileSetGenerator SearchNextGridCell method.

Finally, the last method that the algorithm needs to generate the terrain is the *ConstraintPropagation* method, which has a similar structure to the *TileElection* method, these both methods have the same parts and they are related in a similar way. However, the last one is a little more complex because it needs to delete the tiles that do not follow the constraints from the current variable's domain, so the algorithm needs to do more checks to determine this.

```
private void ConstraintPropagation(int rowIndex, int colIndex, int direction, Variable lastCell, int step)
{
    if (grid[rowIndex, colIndex].domainCount == 1 || step >= maxSteps || rowIndex < 0 || colIndex < 0 || rowIndex >= numRow || colIndex >= numCol)
        return;

    step++;
    if (lastCell.domainCount == 1)
    {
        for (int i = 0; i < grid[rowIndex, colIndex].domain.Length; i++)
        {
            if (grid[rowIndex, colIndex].domain[i] && grid[rowIndex, colIndex].domainCount > 1)
            {
                //Siempre que no sean iguales, como solo hay una posibilidad de conexi�n, se elimina del dominio
                if ((direction == 0 && tileSet[i].tile.sideIndex[2] != lastCell.tileChosen.sideIndex[direction])
                    || (direction == 1 && tileSet[i].tile.sideIndex[3] != lastCell.tileChosen.sideIndex[direction])
                    || (direction == 2 && tileSet[i].tile.sideIndex[0] != lastCell.tileChosen.sideIndex[direction])
                    || (direction == 3 && tileSet[i].tile.sideIndex[1] != lastCell.tileChosen.sideIndex[direction]))
                {
                    grid[rowIndex, colIndex].domain[i] = false;
                    grid[rowIndex, colIndex].domainCount--;
                }
            }
        }
    }
```

```
    else
    {
        // Buscar un metodo para eliminar las teselas que no se pueden
        for (int i = 0; i < lastCell.domain.Length; i++)
        {
            if (lastCell.domain[i])
            {
                for (int j = 0; j < grid[rowIndex, colIndex].domain.Length; j++)
                {
                    if (grid[rowIndex, colIndex].domain[j])
                    {
                        //Si las dos partes coinciden se cuenta como una visita (hay una manera de conectarlas)
                        if ((direction == 0 && tileSet[j].tile.sideIndex[2] == tileSet[i].tile.sideIndex[direction])
                        || (direction == 1 && tileSet[j].tile.sideIndex[3] == tileSet[i].tile.sideIndex[direction])
                        || (direction == 2 && tileSet[j].tile.sideIndex[0] == tileSet[i].tile.sideIndex[direction])
                        || (direction == 3 && tileSet[j].tile.sideIndex[1] == tileSet[i].tile.sideIndex[direction]))
                        {
                            grid[rowIndex, colIndex].visited[j]++;
                        }

                    }
                }
            }
        }

        for (int i = 0; i < grid[rowIndex, colIndex].visited.Length; i++)
        {
            if (grid[rowIndex, colIndex].domain[i] && grid[rowIndex, colIndex].visited[i] < 1) {
                grid[rowIndex, colIndex].domain[i] = false;
                grid[rowIndex, colIndex].domainCount--;
            }
            else
                grid[rowIndex, colIndex].visited[i] = 0;
        }
    }
```

```
    if(grid[rowIndex, colIndex].domainCount == 1)
    {
        int chosenIndex = -1;
        for (int i = 0; i < grid[rowIndex, colIndex].domain.Length; i++)
        {
            if (grid[rowIndex, colIndex].domain[i])
            {
                chosenIndex = i;
                break;
            }
        }

        InstantiateTile(rowIndex, colIndex, chosenIndex);
    }

    //Calculate entropy after the constraint propagation
    grid[rowIndex, colIndex].CalculateEntropy(tileSet);

    // Propagates the constraints bearing in mind the current tile
    if (colIndex > 0 && direction != 1)
        ConstraintPropagation(rowIndex, colIndex - 1, 3, grid[rowIndex, colIndex], step);
    if (colIndex < (numCol - 1) && direction != 3)
        ConstraintPropagation(rowIndex, colIndex + 1, 1, grid[rowIndex, colIndex], step);
    if (rowIndex > 0 && direction != 0)
        ConstraintPropagation(rowIndex - 1, colIndex, 2, grid[rowIndex, colIndex], step);
    if (rowIndex < (numRow - 1) && direction != 2)
        ConstraintPropagation(rowIndex + 1, colIndex, 0, grid[rowIndex, colIndex], step);
}
```

Figure 4.11: TileSetGenerator ConstraintPropagation method.

In this case, a little more explanation of some details in the code is necessary. First of all, (see Figure 4.11), there is a if statement that checks some conditions, if any of them is accomplished, the execution ends and the program starts to go up in the calls stack. Moreover, in this method we are checking if the previous analysed variable has one element in its domain or more than one. Depending on this, the algorithm does different actions, if there is only one tile available, then the algorithm searches in the current variable's domain and compares the sides of each tile. If this tile can connect with the remaining tile of the previous variable, this tile remains available, if not, this tile is deleted from the domain. On the other hand, if there are more than one tile available in the previous variable's domain, the algorithm goes through all the tiles available and compares them with all the tiles available in the current variable's domain, and if they can connect, then we add one to the count of tiles that can connect with the current tile (visited array). After going through the previous variables' domain, the algorithm checks which tiles from the current variable's domain have a count lower than one, and these tiles are deleted from the domain. Finally, the algorithm checks if there is one tile left in the domain of the current variable and if so, it instantiates this tile in the game world. Independently of the domain count of the current variable, the last thing that the algorithm does is propagate the constraints generated by the tiles that remain available in the current variable's domain.

Well, all these classes and methods are the elements needed to create procedural terrain for video games. But, was this actually functional? The answer was yes. Once we had finished writing all the code and modelling all the tiles, we tried to build terrain in a Unity scene. We tried the code with grids of different sizes and changed the depth of the constraint propagation calls and, although the loading time changed with these parameters, the generator

was always able to create a tile map. Finally, the functionality of the terrain was demonstrated and we achieved one of our goals in the project.

4.3. Procedural AI Structure Generation

The next step in the development of the project was to adapt the WFC algorithm to generate behaviour structures that let the NPCs behave intelligently. The strategy that we decided to use with this implementation of the algorithm was basing the classes in the classes that we had needed in the terrain building implementation. Following this strategy, we had a route map which was easy to follow and helped us not to lose focus during the development.

Therefore, following the same logic the first point that has to be explained is the basic key elements of the WFC algorithm applied to AI as we have done with terrain generator implementation. Here, we have some similar pieces of the algorithm: the variables, which are the elements that are unknown and have to be discovered to generate the structure, the domain, which is the set of behaviour blocks that can take up the place of the variables, and the constraints, which are the rules that the behaviour structure have to follow.

On the other hand, there are some differences between the AI generation algorithm and the terrain generation algorithm, for example, the basic elements used and the way how these elements connect to each other. The basic elements in this approach are: the variables, the behaviour blocks and the entropy. In addition, the way these elements are connected is using a grid of variables, the cells of this grid are the variables and they have two degrees of conexion, so they only can connect with the neighbours through the right edge and the down edge, and the connections are one-way. So, we had used this grid because the final structure is a kind of tree, where the root node is the top-left cell and each node has two children at most.

We are going to start talking about the variable, the variable that we are using in this implementation of WFC has two different arguments (see Figure 4.12): the behaviour block that takes up its position in the grid and its children, which are the variables connected to the current one. The algorithm needs the reference to the children because, once the WFC generates the grid, it has to transform the blocks inside into nodes of a tree, so it needs a fast way to go through all the structure in an ordered way. Then, we are going to expand the information about the behaviour blocks and how this class works.

```
public class IAVariable : ScriptableObject
{
    public bool[] domain;
    public int domainCount;
    public int[] visited;
    public BehaviourBlock blockChoosen;
    public float entropy;
    public List<IAVariable> children;

    public void SetVariable(int size)
    {
        children = new List<IAVariable>();
        domain = new bool[size];
        entropy = 0f;

        for (int i = 0; i < size; i++)
            domain[i] = true;

        visited = new int[size];
        blockChoosen = null;
        domainCount = size;
    }
```

Figure 4.12: AI Variable constructor.

Furthermore, inside this *AIVariable* class we need more methods but they are similar to the ones used in the terrain generation. For example, we need a method to calculate the entropy of each variable, so we have used the same *CalculateEntropy* method that we had used in the terrain generation variable. The last variable method that is needed is the *SetBlock* method, used when the variable's domain has only one block remaining or one block is chosen as the one which is going to take up the variable's position in the grid.

That is all that is important to know about the variable in the AI generation implementation, and, as we said, the entropy works equally as it works in the terrain generation implementation. So, now we are going to describe the functionality of the behaviour blocks and how they are used to represent the complex behaviours of the NPCs.

The behaviours block is the name that the instances of the program that executes behaviours on the NPCs have received. These instances have several important methods, the most importants are related with: establishing the connections inside the AIVariables grid where WFC is executed; establishing the connections with the block's children, which is used to create the behaviour structure for each NPC; and executing methods that call functions inside the NPCs that let the system represent these behaviours in the game world. Now, we are going to explain how the behaviour block's class is built and then all the classes that inherit from this main class, finally we are going to explain how all these behaviours are represented in the agent's script.

First of all, we are talking about the *BehaviourBlock* basic class, this class needs some arguments for each activity it develops in the algorithm (see Figure 4.13). The enterConnections and exitConnections arrays are used to keep the final connections that the WFC algorithm established once it decides that one behaviour block takes up an AIVariable position. So, these arrays are keeping a connection with the current behaviour block's neighbours, the enter ones are aiming for the left and up neighbour in the grid and the exit ones are aiming to the right and down neighbour in the grid. Then, the List of BehaviourBlocks named children is used to keep a kind of pointer to the BehaviourBlocks that are taking up the exitConnections' neighbours, these are used to construct in an easy way the tree structure that are going to represent the decision tree that use the NPC to decide.

```
public abstract class BehaviourBlock
{
    [Tooltip("Enter conections used to get in this behaviour")]
    public int[] enterConnections; // 0 -> up side of the block; 1 -> left side of the block

    [Tooltip("Exit conections used to get out this behaviour")]
    public int[] exitConnections; // 0 -> right side of the block; 1 -> down side of the block

    public List<BehaviourBlock> children;

    1 referencia
    public void SetChildren(List<IAVariable> childrenVariables)
    {
        children = new List<BehaviourBlock>();
        for (int i = 0; i < childrenVariables.Count; i++)
        {
            if(childrenVariables[i] != null)
                children.Add(childrenVariables[i].blockChoosen);
        }
    }

    2 referencias
    public void SetConnections(int[] Connections)
    {
        enterConnections = new int[Connections.Length/2];
        exitConnections = new int[Connections.Length / 2];

        for (int i = 0; i < Connections.Length; i++)
        {
            if (i < Connections.Length / 2)
                enterConnections[i] = Connections[i];
            else
                exitConnections[i - Connections.Length / 2] = Connections[i];
        }
    }
```

Figure 4.13: Behaviour Block's class constructors and arguments.

In addition, *BehaviourBlock*'s class needs two more methods that are used to execute the corresponding methods in the enemy script. These methods are *RunCondition* and *Run*, these methods are useful in the decision tree that is formed by the behaviours blocks and they are used to control the execution flow inside the tree. The way that these methods work is the following: when some NPC has to execute one block, first check the *RunCondition*, if it is accomplished, then this node is executed by calling the *Run* method.

Now that we know the basic functioning of the *BehaviourBlock*'s class we are going to view each of the specific classes that inherit from this main class. Each one of these classes represents one of the behaviours that we want the NPCs to carry out in the game world. This way of creating behaviours is inspired by the decision tree and behaviour tree techniques of the AI to construct intelligent agents. The shared parts of the techniques are that the three of them use a tree structure and the three of them use a kind of inheritance making nodes with similar characteristics but adapted to different actions. All we know about these techniques is extracted from the AI book that is recommended to read in the degree subject oriented to the AI techniques [10].

The first subclass that we are going to analyse is the *Patrol* class (see Figure 4.14). This is the base behaviour of every NPC in the game, so all the NPCs start doing a patrol through the game world. That means that this behaviour block is set always as root for the decision tree of any NPC in the game no matter what type would be the NPC.

```
public class Patrol : BehaviourBlock
{
    14 referencias
    public override bool RunCondition(Dictionary<string, float> gameData)
    {
        return gameData["playerDetected"] == 0f;
    }

    8 referencias
    public override BehaviourBlock Run(EnemyAgent enemyAgent, Dictionary<string, float> gameData)
    {
        enemyAgent.GoPatrolling();

        for (int i = 0; i < children.Count; i++)
        {
            if (children[i].RunCondition(gameData))
            {
                return children[i];
            }
        }

        return this;
    }
}
```

Figure 4.14: RunCondition and Run method for the Patrol class.

Here we are going to do a little note about the gameData dictionary, this is present in all the subclasses that inherit from *BehaviourBlock* class. This argument is a dictionary that each enemy fulfils with the information that it gathers from the environment and this info is used to decide if the appropriate conditions are set to accomplish the *RunCondition* of the corresponding block in the tree. In this case, this *RunCondition* is really simple, it always executes while the player remains undetected. Besides, inside the *Run* method we can see that it calls the *GoPatrolling* method of the *EnemyAgent* class which has inside the instructions to represent this behaviour. After each execution of the behaviour, the *RunConditions* of the current behaviour block's children are checked if one of them is accomplished, then the execution will change to the corresponding child, causing an evolution in the NPC external behaviour.

Now that we can understand how the behaviour blocks work internally, we can carry on analysing the rest of the classes that represent a behaviour and that have a little more complexity in their functioning. The next class that we are going to present is the *Retreat* class (see Figure 4.15). This class provides that the NPC carries out a defensive movement going back to a fortified position inside the game arena which lets the enemies coordinate their attacks to face the player.

```
public class Retreat : BehaviourBlock
{
    14 referencias
    public override bool RunCondition(Dictionary<string, float> gameData)
    {
        return gameData["playerDetected"] == 1f && (gameData["health"] < gameData["maxHealth"] / 3
            || gameData["allyNum"] < 4f || gameData["ammo"] < gameData["maxAmmo"] / 2);
    }

    8 referencias
    public override BehaviourBlock Run(EnemyAgent enemyAgent, Dictionary<string, float> gameData)
    {
        enemyAgent.RetreatToHome();

        for (int i = 0; i < children.Count; i++)
        {
            if (children[i].RunCondition(gameData))
            {
                return children[i];
            }
        }

        return this;
    }
}
```

Figure 4.15: RunCondition and Run methods of the Retreat class.

Here we can see that the *RunCondition* has grown in complexity because it keeps in mind more aspects of the environment that can trigger this behaviour. First of all, the player must have been detected by any NPC in the game world, at the same time the NPCs have to detect that their life is less than the half of their full life, that the number of allies is less than four or that the amount of ammunition, which the current NPC has, is less than the half of the maximum amount of ammunition that the NPC can have. On the other hand, inside the *Run* function the *RetreatToHome* method of the *EnemyAgent* class is called which has the instruction to make the NPCs represent this retreating behaviour.

We can continue talking about another *BehaviourBlock* subclass, this time it is named *StrategicPositioning* class (see Figure 4.16). This class is designed to make the NPCs carry out a defensive movement trying to get the back of the player. This movement consists in searching some covers in the environment and hiding after the cover waiting for the player to get close to them. This is a strategy that makes the player search for the NPC leading them to close combats where the player has less numbers than the NPCs.

```
public class StrategicPositioning : BehaviourBlock
{
    14 referencias
    public override bool RunCondition(Dictionary<string, float> gameData)
    {
        return gameData["playerDetected"] == 1f && (gameData["distanceToPlayer"] > gameData["safeDistance"] || gameData["playerVisible"] == 1.0f);
    }

    8 referencias
    public override BehaviourBlock Run(EnemyAgent enemyAgent, Dictionary<string, float> gameData)
    {
        enemyAgent.SearchStrategicPos();

        for (int i = 0; i < children.Count; i++)
        {
            if (children[i].RunCondition(gameData))
            {
                return children[i];
            }
        }

        return this;
    }
}
```

Figure 4.16: RunCondition and Run methods of the StrategicPositioning class.

Here we can see the *RunCondition* method of the *StrategicPositioning* class, this behaviour block uses less information than the retreat one, because we want that this behaviour will be a more common strategy inside the game arena. In this case, the condition considers that the player must be detected by any NPC in the arena and, at the same time, that the distance to the player must be greater than a safe distance to hide without being seen or that the NPCs have direct visibility with the player that means that the NPC is too exposed to be a real threat for the player. Furthermore, this *Run* method calls to the *SearchStrategicPos* function inside the *EnemyAgent* class, which is the function that contains the instructions to represent this behaviour in the game world.

Since now, all the behaviour blocks that have been explained are used for all the NPCs regardless of their type, but now we are going to explain two behaviour blocks that one is the opposite to the other, however, their way of working is really similar. We need this both classes because each one is oriented to one type of enemy, the first one, the *GetClose* class is oriented to face-to-face enemies, the swordsmen, and the second one, the *GetAway* class is oriented to long distance enemies, the archers (see Figures 4.17).

The *GetClose* class is the representation of an offensive movement that lets the NPCs make a direct attack to the player, so the coordination of this attack with the movements of the other NPCs can cause the appropriate distraction to let the enemies kill the player. On the other hand, the *GetAway* class is the representation of an evasion movement during the attack that lets the NPCs take some distance with the player and aim at them. This movement is a form to protect a little the long distance NPCs to let them shoot the player from more secure zones of the arena while the close-combat enemies make a distraction.

For the WFC these two behaviours blocks are the same, because one is a substitute of the other one, so only one of them can appear in the block set of the NPC. These behaviour blocks are usually the previous step to the attack behaviour during the combat, so when the NPC is representing these behaviours the combat is almost finished for one of the sides.

```
public class GetClose : BehaviourBlock
{
    14 referencias
    public override bool RunCondition(Dictionary<string, float> gameData)
    {
        return gameData["playerDetected"] == 1f && gameData["distanceToPlayer"] < gameData["AttackDistance"];
    }

    8 referencias
    public override BehaviourBlock Run(EnemyAgent enemyAgent, Dictionary<string, float> gameData)
    {
        enemyAgent.GetCloseToPlayer();

        for (int i = 0; i < children.Count; i++)
        {
            if (children[i].RunCondition(gameData))
            {
                return children[i];
            }
        }

        return this;
    }
}
```

```
public class GetAway : BehaviourBlock
{
    14 referencias
    public override bool RunCondition(Dictionary<string, float> gameData)
    {
        return gameData["playerDetected"] == 1f && gameData["distanceToPlayer"] < gameData["AttackDistance"];
    }

    8 referencias
    public override BehaviourBlock Run(EnemyAgent enemyAgent, Dictionary<string, float> gameData)
    {
        enemyAgent.GetAwayFromPlayer();

        for (int i = 0; i < children.Count; i++)
        {
            if (children[i].RunCondition(gameData))
            {
                return children[i];
            }
        }

        return this;
    }
}
```

Figure 4.17: RunCondition and Run methods of the GetClose and GetAway classes.

As we can see both classes are really similar, their *RunCondition* is equal, they check that the player has been detected by any NPC in the arena and that the distance between the current NPC and the player would be less than a certain distance appropriate for preparing the attack phase of the combat. Nevertheless, the major difference between both classes is the method which each one calls of the *EnemyAgent* class, the *GetClose* class calls the *GetCloseToPlayer* method which contains the instructions to let the close-combat enemies face the player, unlike the *GetAway* class which calls the *GetAwayFromPlayer* which contains the instructions to let the long combat enemies gain distance to the player and aim at them.

Now it is time to see the last two behaviour blocks designed for the NPCs of this game. The last two subclasses are named *Attack* class and *Shoot* class, and like the previous ones they are used only by one of the enemies' types (see Figure 4.18). The first one is used by the close-combat enemies and lets them swing their sword and protect themselves with the shield

trying to take advantage over the player. The second one is used by the long distance enemies and lets them aim at the player and shoot them, between shots, the enemies need to reload their arcs to shoot again. In this case, the WFC algorithm deals with these behaviour blocks similarly than the last one, so both of them are considered opposites and one is a substitute for the other.

```
public class Attack : BehaviourBlock
{
    14 referencias
    public override bool RunCondition(Dictionary<string, float> gameData)
    {
        return gameData["playerDetected"] == 1f && gameData["distanceToPlayer"] < gameData["hitDistance"];
    }

    8 referencias
    public override BehaviourBlock Run(EnemyAgent enemyAgent, Dictionary<string, float> gameData)
    {
        enemyAgent.Attack();

        for (int i = 0; i < children.Count; i++)
        {
            if (children[i].RunCondition(gameData))
            {
                return children[i];
            }
        }

        return this;
    }
}
```

```
public class Shoot : BehaviourBlock
{
    14 referencias
    public override bool RunCondition(Dictionary<string, float> gameData)
    {
        return gameData["playerDetected"] == 1f && (gameData["distanceToPlayer"] > gameData["ShootDistance"] || gameData["playerVisible"] == 1.0f);
    }

    - referencias
    public override BehaviourBlock Run(EnemyAgent enemyAgent, Dictionary<string, float> gameData)
    {
        enemyAgent.Shoot();

        for (int i = 0; i < children.Count; i++)
        {
            if (children[i].RunCondition(gameData))
            {
                return children[i];
            }
        }

        return this;
    }
}
```

Figure 4.18: RunCondition and Run methods of the Attack and Shoot classes.

In these two subclasses we can see that there are more differences between their *RunConditions* because the two actions that they represent have different needs to be carried on. The *Attack*'s *RunCondition* only needs that the player would be detected by any NPC in the arena and that the distance between the current NPC and the player would be less than the range distance of the enemy arm. On the other hand, the *Shoot*'s *RunCondition* needs that the player would be detected and that the distance between the player and the current NPC would be greater than the distance needed for loading the bow, aiming at the player and shooting or that the player would be directly visible for the current enemy. Besides, the both classes call a

really logical method inside the *EnemyAgent* class which are the *Attack* and *Shoot* methods correspondingly.

Now we need to make the game AI functional and capable of being a real threat for the player, so we have to see how to create a behaviour structure that the NPCa would be able to use. We can start explaining how these behaviours are used to generate the decision tree and, then, how all these behaviours are represented in the game world by the NPCs.

First of all, we want to explain how the behaviour blocks are transformed from a grid to a decision tree. So, to explain this, we have to come back a little to the point that we have explained above. We are talking about the AI variables' grid, this grid where the cells were the variable whose position is taken up by the behaviour blocks. So, once the WFC algorithm has processed all the cells determining which behaviour blocks are going to take the final positions up, we can go through this grid following the exit connections of the AI variables and creating the decision tree. This is possible thanks to keeping in each variable not only which variables are the neighbours but also whose are considered children.

```
private void GenerateBehaviourBlocksChildren(IAVariable currentVariable)
{
    if (currentVariable.blockChoosen == null)
        return;

    currentVariable.blockChoosen.SetChildren(currentVariable.children);

    for (int i = 0; i < currentVariable.children.Count; i++)
    {
        if (currentVariable.children[i] != null)
        {
            GenerateBehaviourBlocksChildren(currentVariable.children[i]);
        }
    }
}
```

Figure 4.19: Generation of the decision tree.

In this method (see Figure 4.19), we can see how the AI variable's children are set as the behaviour blocks' children, connecting directly each behaviour block. So, we can visit the behaviour blocks as nodes, starting with the root node, the Patrol behaviour block, and going through all the hierarchy, using them to change the behaviour that each NPC is representing in the game world.

This decision tree construction must be done after the WFC processing of the AI variable grid, as we said, all the process is really similar to the process explained with the terrain generation. Both of them share the same phases: the initialization of the variables, the generation of the grid, the final blocks establishment and the constraint propagation (see Figure 4.20).

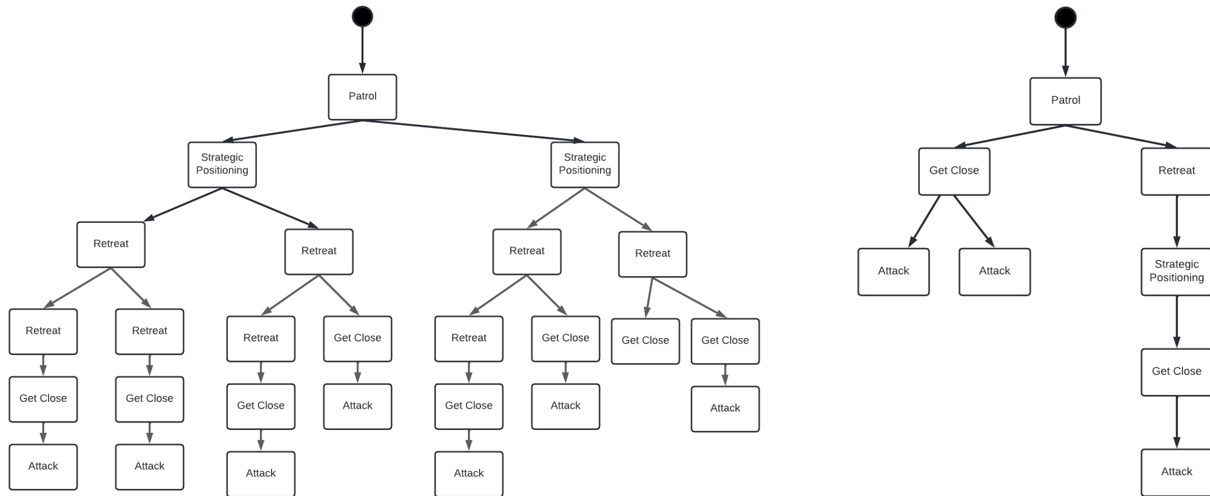


Figure 4.20: Decision Trees generated by the WFC algorithm.

After explaining how the decision tree is built, we think that is a good point to explain how all the behaviour blocks are represented in the game by the NPCs, so here we are going to make a review of the main points in the *EnemyAgent* class to finally understand how we can transform the behaviours from blocks to actual actions in the game world.

```
public class EnemyAgent : MonoBehaviour
{
    [Header("Select Enemy Type")]
    public EnemyType type;

    [Header("Game Manager")]
    public GameManager GM;

    [Header("Making Decisions")]
    // Constants used to decide in the tree
    public float maxHealth;
    public float maxAmmo;
    public float safeDistance;
    public float AttackDistance;
    public float ShootDistance;
    public float hitDistance;

    // Behaviour block Set
    [Header("Block Set")]
    public List<Block> enemyBlockSet;

    // Enemy Functional stuff
    [Header("Enemy Functionality stuff")]
    public float shootForce;
    public Transform tileMap;
    public NavMeshAgent agent;
    public Transform shootPos;
    public GameObject arrow;
    public GameObject sword;
    public Transform currentWaypoint;
    public Transform[] waypoints;
    public List<Transform> homePositions;
    public List<Vector3> strategicalPosition;

    [Header("Animation Stuff")]
    public Animator anim;

    [Header("Sound Stuff")]
    public AudioSource stepSound;
    public AudioSource effectSound;
    public AudioClip enemyHurt;
    public AudioClip enemyAttack;
    public AudioClip enemyBlock;
    public AudioClip bowCharge;
    public AudioClip[] stepsSoundEffect;
```

```
// Game data container to use the BehviourBlocks
Dictionary<string, float> gameData; // keys: {playerD
                                     // safeDistance,

// Variables used to check the state of the game
public bool playerDetected;
float ammo;
float health;
EnemyAgent[] allies;
int allyNum;
GameObject player;
bool isBlocking;
bool isAttacking;

// Enemy Functional variables
int waypointIndex;
float waitTimer = 5.0f;
Transform homeWaypoint;
bool retreating = false;
bool strategicallyHide = false;
float shootCoolDown = 8.0f;
bool reloading = false;
bool gettingAway = true;
bool headingPlayer = false;

// Variables that check the gameData Update
float gameDataUpdateTimer = 2.0f;
float gameDataUpdateTime = 0.0f;

// BehaviourGenerator and BehaviourTree
BehaviourBlockGeneration treeGenerator;
BehaviourBlock rootBlock;
BehaviourBlock currentBlock;

// Tiles Grid to search the strategical positions
Transform predefinedPath;
GameObject[,] gameObjectGrid;

// Damage control variables
bool damaged = false;
float damagedCoolDown = 2.0f;
float damagedTime = 0f;
```

Figure 4.21: Attributes of the EnemyAgent class.

As we can see (see Figure 4.21), the EnemyAgent class needs a lot of attributes to control the functioning of all the features related with the NPCs, in this list not all the attributes has something to do with the decision tree or the behaviours, a lot of them are simple functional stuff related with the sound or some mechanical aspects. What is key for the functioning of the behaviour translation from nodes to actual actions are the constructor of the NPC, because inside the constructor is where the decision tree is assembled and initialised.

Inside this constructor we are establishing the block set that is going to be used for the decision tree and it is determined by the election that the player can do using the buttons in the initial room inside the game world. Besides, we are generating the entire decision tree and using a variable to keep the reference to the root node, which is the entrance door to access to the rest of the behaviours blocks. Finally we determine that the first block being executed has to be the root node, that is the patrol behaviour (see Figure 4.22).

```
void Start()
{
    // Creating the Decision Tree
    treeGenerator = new BehaviourBlockGeneration();
    treeGenerator.blockSet = enemyBlockSet.ToArray();
    if (rootBlock != null)
    {
        ClearBlockTree(rootBlock);
        treeGenerator.ClearTree();
    }

    while (rootBlock == null)
    {
        rootBlock = treeGenerator.Generate();
        if (rootBlock == null)
            treeGenerator.ClearTree();
    }
    currentBlock = rootBlock;

    DebugArbol(); // Method that shows the tree blocks
}
```

Figure 4.22: EnemyAgent constructor.

In addition to the constructor, we have another method to initialise some important aspects for the correct working of the agent (see Figure 4.23). This method is the OnEnable method provided by Unity which lets us execute some actions each time the NPCs are activated in this initial room, just before the player enters the fighting arena. Inside this method, all the functional attributes are initialised such as the life, the ammo and so on. Moreover, each NPC searches which other NPCs have been activated and where it can find some covers inside the arena which are information needed to trigger some specific behaviours.

```
private void OnEnable()
{
    // Initializing the Game Variables
    playerDetected = false;
    ammo = maxAmmo;
    health = maxHealth;
    allies = FindObjectsOfType<EnemyAgent>();
    homeWaypoint = GameObject.Find("homeWaypoint").transform;
    player = GameObject.Find("Player");
    detector = transform.Find("Detector").gameObject;

    for(int i = 0; i < tileMap.childCount; i++)
    {
        if(tileMap.GetChild(i).name.Contains("Predefined") && tileMap.GetChild(i).gameObject.activeSelf)
            predefinedPath = tileMap.GetChild(i);
    }
```

```
    gameObjectGrid = new GameObject[tileMap.GetComponent<TileSetGenerator>().numRow,
        tileMap.GetComponent<TileSetGenerator>().numCol];

    // Initializing the game data structure
    gameData = new Dictionary<string, float>();
    gameDataUpdateTime = gameDataUpdateTimer;

    // Searching for covers in the arena
    if (!GM.isInTutorial)
    {
        GetGameObjectGrid();
        GetStrategicalPositions();
    }

    stepSound.clip = stepsSoundEffect[GM.tileSetChoosen];
}
```

Figure 4.23: EnemyAgent OnEnable method.

After the construction and initialization of the EnemyAgents we are going to start overviewing each one of the methods related with the behaviours of the NPC. We are going to start with the basic behaviour, that is the patrol behaviour that each enemy agent has to do while the player stays undetected inside the arena. All the movement is carried out by the *NavMeshAgent* class, so we only need to determine which are the waypoints of the patrol. Besides, we have determined that this behaviour alternates between two actions, patrolling and staying still (see Figure 4.24), which allows the NPCs to check some zones of the arena and gives the player the chance to get rid of some enemies without being seen.

As we said before, this is the base method of the enemy agents so it is the root node in every decision tree. This method can be connected following the restrictions with three more behaviour blocks which are going to be explained in the next paragraphs. The behaviour blocks that can be the patrol children are the retreat block, the strategic positioning block and the get away/get close block. That means that these three blocks can be connected among them, making the decision tree more complex and varied.

```
agent.SetDestination(currentWaypoint.position);

// Moving among the patrol waypoints
if (!agent.isStopped && Vector3.Distance(currentWaypoint.position, transform.position) < agent.stoppingDistance)
{
    if (UnityEngine.Random.value > 0.2f)
    {
        waypointIndex++;
        if (waypointIndex >= waypoints.Length)
            waypointIndex = 0;

        currentWaypoint = waypoints[waypointIndex];
        anim.SetBool("IsMoving", true);

        if (!stepSound.isPlaying)
            stepSound.Play();
    }
    else
    {
        agent.isStopped = true;
        anim.SetBool("IsMoving", false);

        if (stepSound.isPlaying)
            stepSound.Stop();
    }
}
else if (agent.isStopped)// Waiting time until start patrolling again
{
    if(waitTimer == 5.0f)
    {
        Vector3 pos = transform.position + transform.forward * 20;
        RotateEnemy(pos);
    }

    waitTimer -= Time.deltaTime;
    if (waitTimer <= 0f)
    {
        waitTimer = 5.0f;
        agent.isStopped = false;
        anim.SetBool("IsMoving", true);

        if (!stepSound.isPlaying)
            stepSound.Play();
    }
```

Figure 4.24: GoPatrolling method of EnemyAgent class.

The next enemy agent behaviour that we need to show is the retreat behaviour. This is a simple behaviour but this is separated in different phases. First of all, when this action is started, the agent simply heads into the waypoint that we have named home, which is in the centre of a fortificated position. Once the NPCs have got to this point, they start searching for a strategic position inside this fortificated area, at this moment, depending on the type of the NPC a certain position is favoured for the election, so the close-combat enemies would take up the front positions and the long distance ones the back positions. Then, when the agents have taken up their corresponding positions, starts the last phase of the behaviour where the agents wait for the player (see Figure 4.25). This whole behaviour is trying to take advantage of the numbers of the agents that is why the agents get to a position where the player would have to face different enemies at the same time.

The idea in this behaviour is that different NPCs decide to carry on these actions at the same time or separated by a little time space so they arrive at the same time to the fortified position. This is the way that the both sides in the face off are balanced, designing situations where the player must fight against a bigger number of enemies.

```
internal void RetreatToHome()
{
    // Going to the Fortificate position in the arena
    if(currentWaypoint != homeWaypoint && !retreating)
    {
        if (agent.isStopped)
            agent.isStopped = false;

        currentWaypoint = homeWaypoint;
        anim.SetBool("IsMoving", true);
        retreating = true;

        if (!stepSound.isPlaying)
            stepSound.Play();
    }

    // Selecting an Strategical position in the fortificate area
    if (!homePositions.Contains(currentWaypoint)
        && Vector3.Distance(transform.position, homeWaypoint.position) < agent.stoppingDistance)
    {
        if (type == EnemyType.Archer)
        {
            var furthestPosition = -Mathf.Infinity;
            for (int i = 0; i < homePositions.Count; i++)
            {
                var selected = false;
                foreach (EnemyAgent ally in allies)
                {
                    if (homePositions[i] == ally.currentWaypoint)
                    {
                        selected = true;
                        break;
                    }
                }

                if (!selected)
                {
                    var distance = Vector3.Distance(homePositions[i].position, player.transform.position);
                    if (distance > furthestPosition)
                    {
                        currentWaypoint = homePositions[i];
                        furthestPosition = distance;
                    }
                }
            }
```

```
        else
        {
            var closestPosition = Mathf.Infinity;
            for (int i = 0; i < homePositions.Count; i++)
            {
                var selected = false;
                foreach (EnemyAgent ally in allies)
                {
                    if (homePositions[i] == ally.currentWaypoint)
                    {
                        selected = true;
                        break;
                    }
                }

                if (!selected)
                {
                    var distance = Vector3.Distance(homePositions[i].position, player.transform.position);
                    if (distance < closestPosition)
                    {
                        currentWaypoint = homePositions[i];
                        closestPosition = distance;
                    }
                }
            }
        }

        anim.SetBool("IsMoving", true);
        if (!stepSound.isPlaying)
            stepSound.Play();
    }

    agent.SetDestination(currentWaypoint.position);
```

```
// Waiting for the player to arrive
if (currentWaypoint != homeWaypoint
    && Vector3.Distance(transform.position, currentWaypoint.position) < agent.stoppingDistance)
{
    agent.isStopped = true;
    anim.SetBool("IsMoving", false);
    if (stepSound.isPlaying)
        stepSound.Stop();
    RotateEnemy(player.transform.position);
}
```

Figure 4.25: Retreat method of enemyAgent class.

As we have exposed aboved, there is one more global behaviour represented by the agents, we are talking about the strategic positioning behaviour that is carried out by all the NPCs independently of its type. This behaviour needs a previous method that has been already seen but we have omitted to talk about it until this point. In the *OnEnable* method, we call two functions that are related with this behaviour and are needed for the well functioning of it. These two methods are used to find all the possible covers generated when the NPCs are activated just after the terrain building by the WFC algorithm. The first one is the *GetGameObjectGrid* method, this method is really simple, it keeps the reference of all the tiles that form the terrain in a GameObjects grid, so it could be accessed from the agent at any moment. The second one is the *GetStrategicPositions* method and this is a little more complex than the last one (see Figure 4.26). Inside this method, we are going through all the gameObjects in the grid and we check which tiles have a cover onto them, then we keep inside an array which positions can be used to protect the NPC with this cover, that is the neighbours with no cover on it.

```
// Search trhough all the tiles grid
for (int i = 0; i < gameObjectGrid.GetLength(0); i++)
{
    for (int j = 0; j < gameObjectGrid.GetLength(1); j++)
    {
        if (gameObjectGrid[i, j].name.Contains("Cover")) // Checking that the current tile is a cover
        {
            // Checking that the neighbour tiles are not covers, and they are inside the grid
            if ((i - 1) > 0 && !gameObjectGrid[i - 1, j].name.Contains("Cover"))
            {
                var pos = new Vector3(j * tileMap.GetComponent<TileSetGenerator>().tileSize.x + tileMap.GetComponent<TileSetGenerator>().tileSize.x / 2,
                    tileMap.GetComponent<TileSetGenerator>().tileSize.y / 2,
                    (i - 1) * tileMap.GetComponent<TileSetGenerator>().tileSize.z + tileMap.GetComponent<TileSetGenerator>().tileSize.z / 2);
                if (!strategicalPosition.Contains(pos))
                    strategicalPosition.Add(pos);
            }
            if ((i + 1) < (gameObjectGrid.GetLength(0) - 1) && !gameObjectGrid[i + 1, j].name.Contains("Cover"))
            {
                var pos = new Vector3(j * tileMap.GetComponent<TileSetGenerator>().tileSize.x + tileMap.GetComponent<TileSetGenerator>().tileSize.x / 2,
                    tileMap.GetComponent<TileSetGenerator>().tileSize.y / 2,
                    (i + 1) * tileMap.GetComponent<TileSetGenerator>().tileSize.z + tileMap.GetComponent<TileSetGenerator>().tileSize.z / 2);
                if (!strategicalPosition.Contains(pos))
                    strategicalPosition.Add(pos);
            }
            if ((j - 1) > 0 && !gameObjectGrid[i, j - 1].name.Contains("Cover"))
            {
                var pos = new Vector3((j - 1) * tileMap.GetComponent<TileSetGenerator>().tileSize.x + tileMap.GetComponent<TileSetGenerator>().tileSize.x / 2,
                    tileMap.GetComponent<TileSetGenerator>().tileSize.y / 2,
                    i * tileMap.GetComponent<TileSetGenerator>().tileSize.z + tileMap.GetComponent<TileSetGenerator>().tileSize.z / 2);
                if (!strategicalPosition.Contains(pos))
                    strategicalPosition.Add(pos);
            }
            if ((j + 1) < (gameObjectGrid.GetLength(1) - 1) && !gameObjectGrid[i, j + 1].name.Contains("Cover"))
            {
                var pos = new Vector3((j + 1) * tileMap.GetComponent<TileSetGenerator>().tileSize.x + tileMap.GetComponent<TileSetGenerator>().tileSize.x / 2,
                    tileMap.GetComponent<TileSetGenerator>().tileSize.y / 2,
                    i * tileMap.GetComponent<TileSetGenerator>().tileSize.z + tileMap.GetComponent<TileSetGenerator>().tileSize.z / 2);
                if (!strategicalPosition.Contains(pos))
                    strategicalPosition.Add(pos);
            }
        }
    }
}
```

Figure 4.26: Get Strategic Position method of EnemyAgent class.

Now that we know about the supportive methods of the StrategicPositioning behaviour, we can dive into this method to understand how the agent carries out the corresponding actions. The *StrategicPositioning* method is divided into two phases, the first one where the agents search among all the strategic positions that are available in the arena and go to the resulting position. The final position is chosen using a really simple heuristic: all the positions are analysed and a first selection is made, which gathers all the strategic positions that are further than a certain safe distance from the player, among them it is chosen the position which would be closest to the player. The second phase starts once the NPCs have arrived at the strategic position, then they have to wait for the player to come closer than a certain distance to start the combat (see Figure 4.27). This behaviour is motivated by the idea of hiding from the player forcing them to search all the enemies around the arena. This situation is good for the NPCs because they can make a surprise attack having the initiative in the combat, and starting it causing damage to the player.

```
var nextPos = currentWaypoint.position;
// Searching a strategical position in the arena
if (!strategicalPosition.Contains(agent.destination))
{
    var bestDistance = Mathf.Infinity;
    for (int i = 0; i < strategicalPosition.Count; i++)
    {
        var selected = false;
        foreach (EnemyAgent ally in allies)
        {
            if (strategicalPosition[i] == ally.currentWaypoint.position)
            {
                selected = true;
                break;
            }
        }

        if (!selected)
        {
            var distance = Vector3.Distance(strategicalPosition[i], player.transform.position);
            if (distance > safeDistance && distance < bestDistance)
            {
                nextPos = strategicalPosition[i];
                bestDistance = distance;
            }
        }
    }

    anim.SetBool("IsMoving", true);
    if (!stepSound.isPlaying)
        stepSound.Play();
}

agent.SetDestination(nextPos);

// Waiting the player to arrive
if (Vector3.Distance(transform.position, nextPos) < 1f)
{
    anim.SetBool("IsMoving", false);
    agent.isStopped = true;
    strategicallyHide = true;

    if (stepSound.isPlaying)
        stepSound.Stop();
}
```

Figure 4.27: StrategicPositioning method of the EnemyAgent class.

As we said above, apart from the global behaviours there are four behaviour blocks that can only be held in the decision tree of a certain type of enemy. Firstly we are going to explain

the functioning of the get close and get away behaviour. As we have said in the behaviour blocks review, the get close behaviour is carried out by close-combat enemies and the get away behaviour is carried out by long distance enemies (see Figure 4.28). Internally these two methods work similarly to some steering behaviours, concretely to the pursuit and flee behaviours.

Starting with the get close behaviour, we can say that it is really similar to the pursuit steering behaviour [10], this behaviour is specialised in making the agent chase a moving target, first of all, the agent must calculate the time that it can take to get to the target. Then, the agent calculates which is going to be the future position of the target at this time. Finally, the agent chases this future position and when it is closer to the player than to the future positions, it changes its target to the current position of the player.

As we said before, the get away behaviour, which we have used, is really similar to the flee steering behaviour [10], this behaviour is specialised in making the agent run away from the current position of a target. We have chosen this behaviour because it is faster to calculate and reduce the possibilities of failure. Besides, We have tried using the evade behaviour but the result was that the agent selected too far positions for the current arena size. Inside this method, the agent calculates the direction to get closer to the player position and then invert the direction of the movement, the target position is limited to the arena borders trying to avoid crashing into a wall or falling from the game world. Finally, the agent waits in this final position and aims at the player.

```
internal void GetCloseToPlayer()
{
    if(!headingPlayer)
    {
        if (agent.isStopped)
            agent.isStopped = false;

        float time = gameData["distanceToPlayer"] / agent.speed; // Time that las the enemy to arrive to the player

        Vector3 futurePlayerPosition = player.transform.position +
            player.GetComponent<Player>().movement * player.GetComponent<Player>().movementSpeed * time; // Future position of the player in that time

        if (!anim.GetBool("IsMoving"))
            anim.SetBool("IsMoving", true);

        if (!stepSound.isPlaying)
            stepSound.Play();

        agent.SetDestination(futurePlayerPosition);

        // If the enemy is close enought go for the player
        if (Vector3.Distance(transform.position, futurePlayerPosition) >= gameData["distanceToPlayer"])
        {
            agent.SetDestination(player.transform.position);
            headingPlayer = true;
        }
    }
    else
    {
        if (!anim.GetBool("IsMoving"))
            anim.SetBool("IsMoving", true);

        agent.SetDestination(player.transform.position);
        RotateEnemy(player.transform.position);
    }
}
```

```
internal void GetAwayFromPlayer()
{
    if (gettingAway)
    {
        if (anim.GetBool("IsAiming"))
            anim.SetBool("IsAiming", false);

        Vector3 direction = (player.transform.position - transform.position).normalized; // Get the opposite direction to the player
        Vector3 targetPos = -direction * ShootDistance; // Calculate an appropriate position to shoot to the player

        targetPos.x = Mathf.Clamp(targetPos.x, 1f, gameObjectGrid.GetLength(1) - 2);
        targetPos.z = Mathf.Clamp(targetPos.z, 1f, gameObjectGrid.GetLength(0) - 2);

        if (!anim.GetBool("IsMoving"))
            anim.SetBool("IsMoving", true);
        if (!stepSound.isPlaying)
            stepSound.Play();

        agent.SetDestination(targetPos);

        if (Vector3.Distance(transform.position, targetPos) < agent.stoppingDistance)
        {
            gettingAway = false;
            agent.isStopped = true;

            if (anim.GetBool("IsMoving"))
                anim.SetBool("IsMoving", false);
            if (stepSound.isPlaying)
                stepSound.Stop();
        }
    }
}
```

Figure 4.28: GetClose and GetAway methods of EnemyAgent class.

There are two more behaviours that can only be performed by a certain type of agent, these are the attack and shoot behaviours, as we said the attack behaviour corresponds to a close-combat enemy and the shoot behaviour corresponds to a long distance one (see Figure 4.29).

Firstly we are going to explain the attack behaviour, this method has two well different phases, the first phase involves getting close to the player position always because between the fighting actions the player could have moved back making impossible for the agent to impact the player. Once the agent is at an appropriate distance to the player, a random value is generated, this value is used to select which fighting action is going to do the agent, it could perform an attack with the sword, it could block with the shield or it could do nothing. This probability makes the combat unpredictable and creates different opportunities for the player and the agent.

On the other hand, we are going to explain the shoot behaviour, this method has two different phases too, the first phase involves calculating the player position and the direction that the agent needs to look at to shoot the player. The second phase involves checking if the bow is loaded with an arrow and if the agent has direct visibility to the player making it possible to shoot the arrow. Once an arrow is shooted the second phase is changed by reloading the arrow, so after each shot the player has the reload time to finish with the archer.

```
internal void Attack()
{
    if (gameData["distanceToPlayer"] > 1.7f) // Get to hit distance of the player
    {
        if (agent.isStopped)
            agent.isStopped = false;

        currentWaypoint = player.transform;
        anim.SetBool("IsMoving", true);
        if (!stepSound.isPlaying)
            stepSound.Play();

        agent.SetDestination(currentWaypoint.position);
    }
    else if(!isAttacking && !isBlocking)
    {
        if (!agent.isStopped)
            agent.isStopped = true;

        anim.SetBool("IsMoving", false);
        if (stepSound.isPlaying)
            stepSound.Stop();

        var randomValue = UnityEngine.Random.value;
        if ( randomValue > 0.7) // Attack the player
        {
            StartCoroutine("attackAnimation");
        }
        else if(randomValue < 0.3) // Block the player attack
        {
            StartCoroutine("blockAnimation");
        }
        var pos = new Vector3(player.transform.position.x, player.transform.position.y + .5f, player.transform.position.z);
        RotateEnemy(pos);
    }
}
```

```
internal void Shoot()
{
    // The enemy stops moving
    if (anim.GetBool("IsMoving"))
        anim.SetBool("IsMoving", false);
    if (stepSound.isPlaying)
        stepSound.Stop();

    if (!anim.GetBool("IsAiming"))
        anim.SetBool("IsAiming", true);

    // Calculate the rotation to face the player
    var pos = new Vector3(player.transform.position.x, player.transform.position.y + .5f, player.transform.position.z);
    RotateEnemy(pos);

    Vector3 playerDirection = player.transform.position - transform.position;
    playerDirection.y += .5f;
    playerDirection = playerDirection.normalized;

    // Check if there is enought ammo to shoot and the player is visible to shoot them
    RaycastHit hit;
    if (!reloading && ammo > 0 && Physics.Raycast(shootPos.position, playerDirection, out hit, ShootDistance) && hit.transform.tag == "Player")
    {
        StartCoroutine(shootAnimation(playerDirection));
    }
    else if (reloading)
    {
        shootCoolDown -= Time.deltaTime;
        if (shootCoolDown < 0)
        {
            reloading = false;
            shootCoolDown = 8.0f;
        }
    }
}
```

Figure 4.29: Attack and Shoot method of the EnemyAgent class.

The last thing that we are going to talk about the *EnemyAgent* class is how the agents can interactuate and gather information about the environment. In this game we have decided that the NPCs have the capability of “seeing” their environment and “listening” to the alarm. Therefore, each NPC has a range of vision and if the player gets inside this range, the agent makes a ray casting, if the ray impacts with the player, then automatically the NPC changes

its state to the alarm state. Every NPC that has entered the alarmed phase spreads the alarm state inside a certain range, all the agents inside this range change their state to the alarm state too and this process is repeated since every agent in the arena is in the alarm state.

4.4. Game Environment

After developing all the functional parts of the game which was the main effort in the project, we decided to finish the environment of the game, creating the initial room and the arena, needed to make the game interactable. To develop all this designing part we had to use some tools and assets that let us make all this visual aspect easier because it is not our strong point in games development. First of all, we design the tutorial space and the initial room as futuristic spaces because these are the places where the player is going to be able to manipulate the functioning of the WFC algorithm (see Figure 4.30). To create this space we have used an asset which provides structures and objects that let us create a futuristic industrial-style space which fits really well with the purpose of these both rooms [15].

Figure 4.30: Tutorial and initial room style.

On the other hand, we had to design all the arena which we had decided to create it as a kind of Roman Coliseum, we had searched in the asset store some arenas that can fit in the idea that we had about a Coliseum and that could be functional for the correct use of the WFC generation, because the arena needs to be adapted to the size and the shape of the terrain generated. After all, we had not found any arena that solves all our problems, so we decided to design it by hand. We had searched for some tools and tutorials that could make all this creation process easier, and we had found two tools that are really useful and easy-to-use, so we chose them to create the Coliseum. These two tools are created by Unity and work really well with the current version of the engine, these are the ProBuilder tool and the PolyBrush tool, [22, 24]. Firstly, we had no idea about how to use these tools, so we had searched some help on tutorials which it is always a good idea and we found two tutorial really understandable and that cover all the basic features of the tools so they gave us all the info that we need to design the arena [23, 25]. Finally, when the arena was finished we needed some textures to dress the polygons that form the structure, these textures are the ones used by the PolyBrush (see Figure 4.31). All the textures are downloaded from Share Textures a website specialised in copyright free textures, the textures that we needed were a stone bricks texture for the arena walls [18], a wood texture for the wood details as fences and beams [21] and some fabric textures for the sunshades that cover the arena grandstand [19, 20].

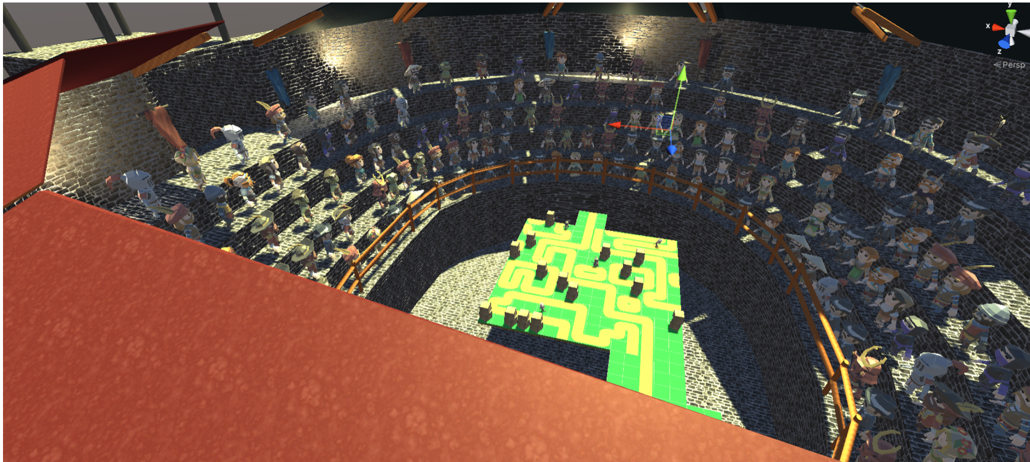


Figure 4.31: Arena structure and textures style.

In addition, we need some objects that give the arena some realism and that all these space could not look so artificial and empty, so to populate the arena we decided to use some assets from the asset store, that keep the low poly aspect that we used in the terrain but that could be design with more skill than we have. Firstly, we have used some asset packages to create the covers inside the arena, where we have used crates and barrels, these were chosen by their textures that respect the low poly aspect but have detail in them and most importantly, they use LOD [11, 12]. Secondly, we used some assets to make the grandstand space more accurate for a imaginable Coliseum not to get too much attention for the player and let them concentrate on the game action, this space have been populated with some stuff provided by a mediaeval fortress asset and all the object fit really well with the general style of the structure [13]. Finally, we have used an asset to provide the game with appealing characters not only inside the arena, as the main character or the enemies, but also as part of the crowd that is spectating all the combat inside the arena (see Figure 4.32). All the characters in the game are part of the same package which has a really high visual quality bearing in mind that they keep the low poly style of the entire game [14].

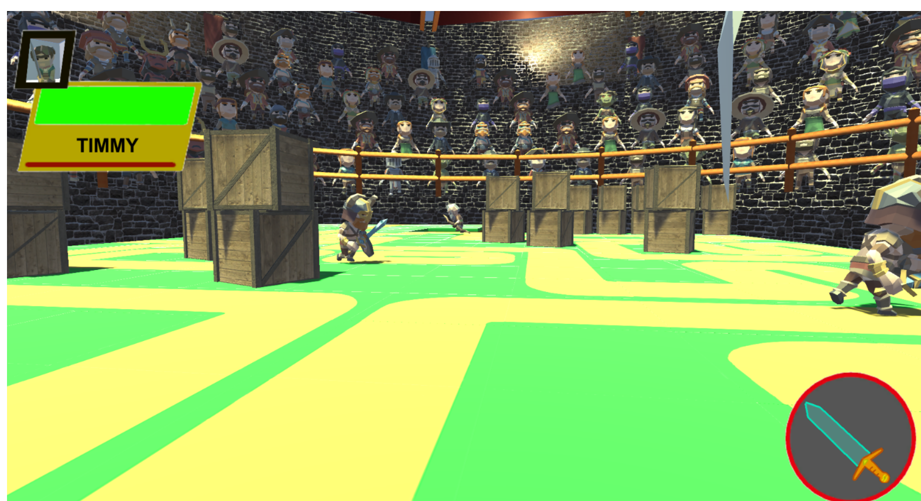


Figure 4.32: Arena objects style.

4.1.1. Problems

In this section we would like to mention some problems that appeared during the development of the project.

The main problem in the project was the optimization of the WFC algorithm working in the grid. The problem was produced because we have used several loops to implement the constraint propagation in the grid and the tile election. However, the problem was notable when we realised that all the loops were going to be needed to make the algorithm work the way that we wanted. So, when we started making the grid bigger from a 5x5 grid to a 10x10 grid and even to a 15x15, it was obvious that the optimization was a problem, the system took several minutes to establish all the tiles. But, that was not all, when we started to change the number of tiles in the tile set, which is a main mechanic in the game, we realised that this affected the processing time of the algorithm. Here, we started to think that it was going to be impossible to generate a terrain with the enough size in execution time, because it was taking too much time for the terrain generation, and to this time we had to add the time of all the NPCs' decision tree generation which was going to use WFC too. Finally, we found a solution, implementing correctly the entropy formula to select the new cells where the algorithm is going to work, limiting the number of steps that the algorithm can do during the propagation and limiting the minimum tiles and blocks from the tile set for the terrain and the block set for the decision tree, were some of the solutions that better results give us in terms of optimization of the algorithm.

I had some problems that are related with the experimentation base of the project, because sometimes, the terrain generation and the decision tree generation end with final results that are appealing for understanding the power of the algorithm but they are unappealing for the normal development of a game. This problem is easily fixable, adding to the algorithm a checking method that lets us clear the terrain and rebuild it if we find some strange combination of tiles or blocks. This could be a good point to investigate in future projects.

Another problem that we found, now related with the main camera, was that the camera that represents the main character view has problems with viewing through some objects as walls, decorating stuff or the enemies the player is fighting against. We tried to solve this problem changing the position of the camera and limiting the movement but we found that moving the camera the only that we achieved was looking the inside of the main character head what difficult a lot the playability. So, what we decided was to move this camera to a midpoint, however there are some objects and some combat situations where this problem remains.

The last problem we found is related with the *RunConditions* of some behaviour blocks, there some specific situations where one behaviour is not activated because one of their conditions is not accomplished, what cause that the agent do random actions, however this condition is necessary for the well-functioning on the majority of the situations in the game. So, the only

solution that we found was making a fine tune of these conditions and trying to adapt it to most of the situations that could be found during the game.

4.2 Results

Based on the objectives that have been exposed in section 2.1, we can ensure that all the objectives have been completed.

Thanks to this project we have understood how the WFC algorithm works and what are the main ideas behind this kind of algorithm. We have understood the importance and the power in the procedural generation methods and how these algorithms could be the future of the video games industry. A part of this, we have learnt how to implement a complex algorithm since the beginning. Besides, we have demonstrated that we are capable of adapting an entire algorithm design for terrain generation to a totally different work such as the AI structures generation, which was the actual big challenge of the project, so it is the biggest victory of the project. Finally, we have developed a demo where the player can prove the functioning of the algorithm and it is easy to understand how the algorithm works and how powerful it is.

We are really proud of this demo because it is good entertainment and it lets us put the people who want to know more about algorithms of procedural content generation closer to a really good example of a tool that they can use for their projects. The demo is available for Windows in the following link:

https://drive.google.com/file/d/14uV2sg9zdhmh7qd1sBKdrtjTKA82dMXj/view?usp=sharing

You can also access to the gitHub repository where all the project code is available in the next link:

https://github.com/AdrianRamosBoira/WFC.git

As we said before and during the whole report, adapting the WFC algorithm from the terrain generation to the AI structure generation is the biggest objective that we have got in this project. First of all, it sounded crazy to start thinking about adapting an algorithm to a field not even explored when we knew nothing about the application of the algorithm. To close this chapter, we are going to add some images of the final game and of all the ways that the player can interact with the algorithm to let you understand better how is the final result of the project that you have read and we encourage you to give a chance to the demo if you are interested in the generation of content for video games.

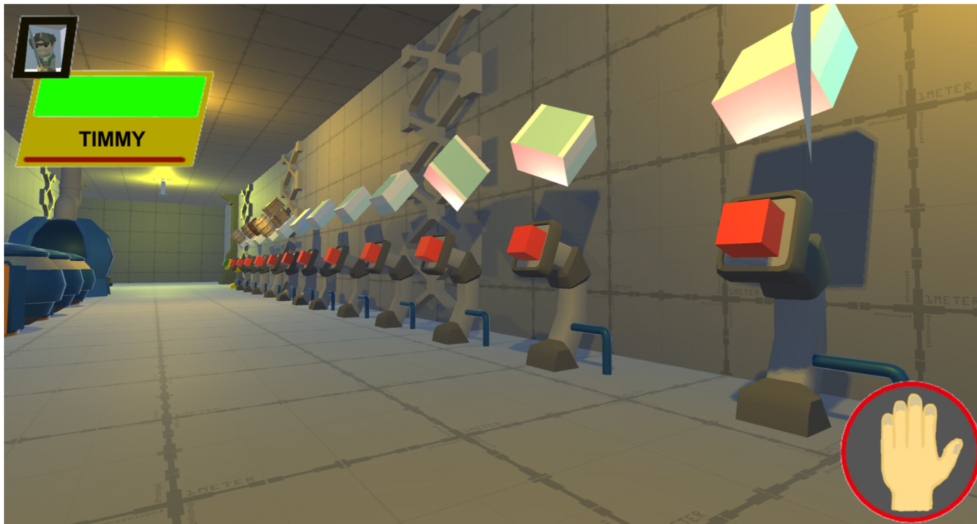


Figure 4.33: Choosing the tile set for the terrain generation.

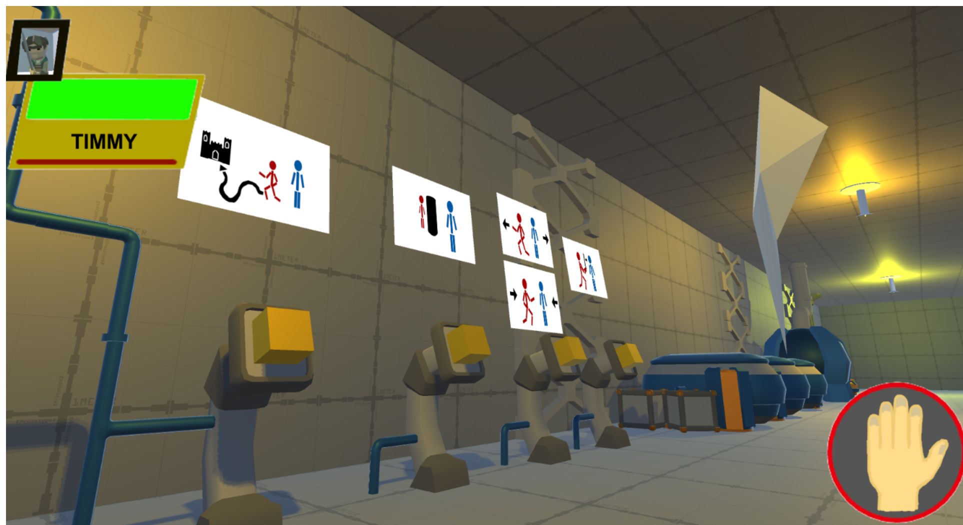


Figure 4.34: Choosing the block set for the behaviour generation.

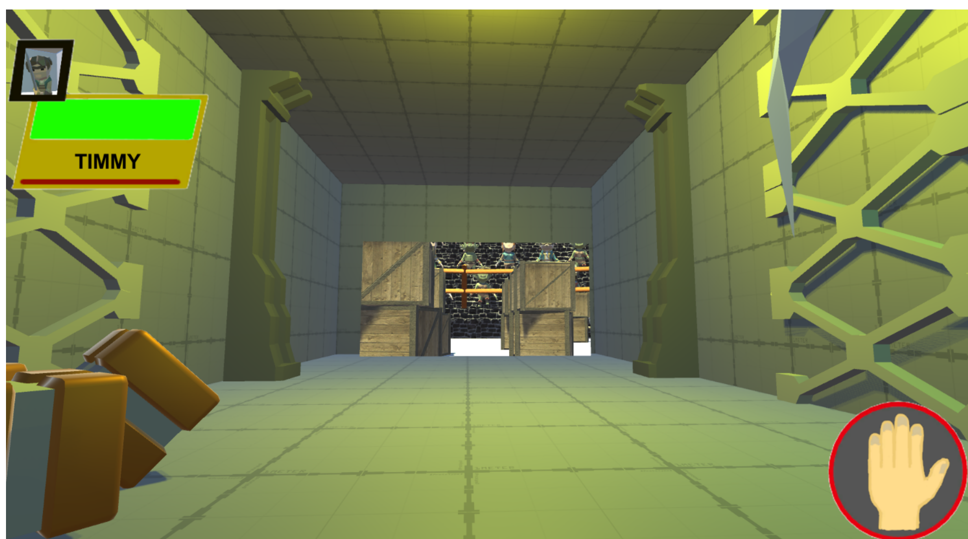


Figure 4.35: Entering the arena through the portal.

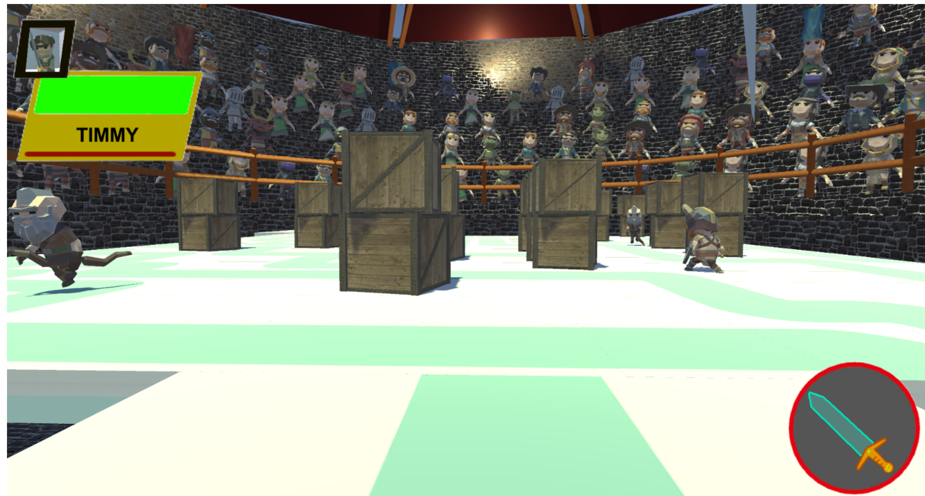


Figure 4.36: Player inside the arena.

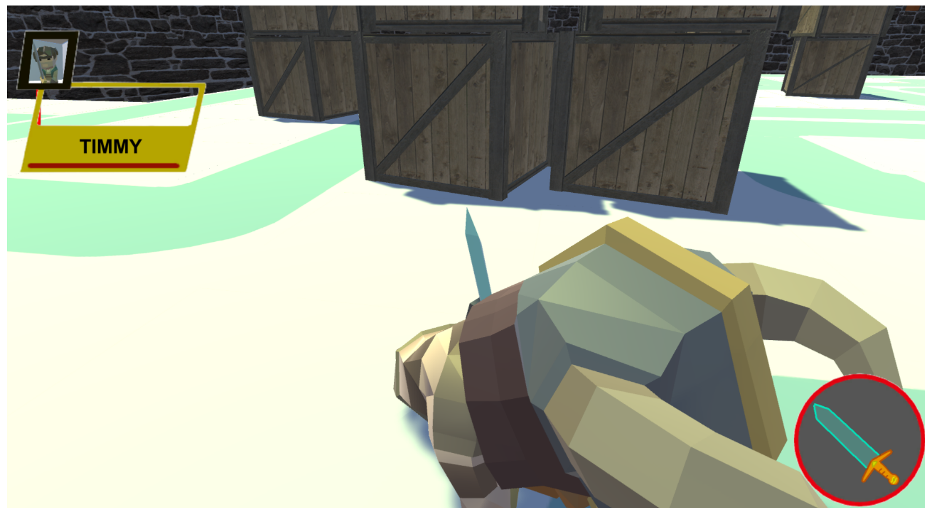


Figure 4.37: Fighting the face to face enemy.

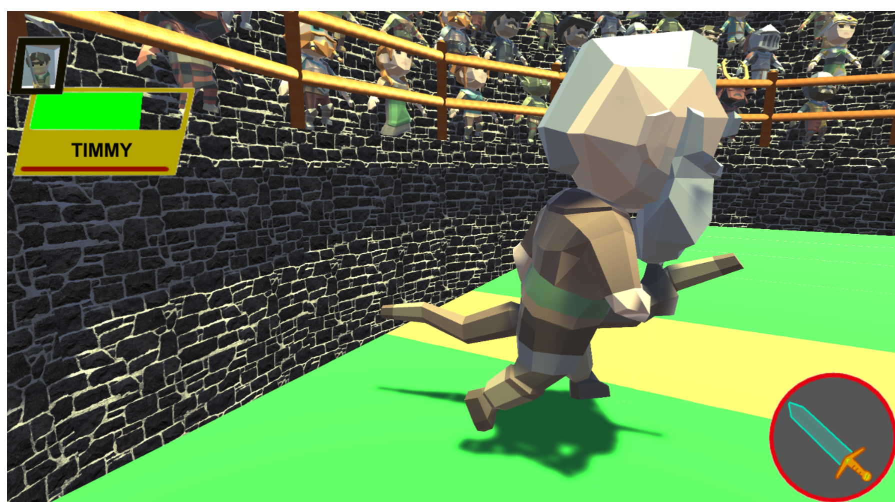


Figure 4.38: Fighting the long distance enemy.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

Contents

5.1. Conclusions 64
5.2. Future work 65

In this chapter the conclusions of the project are shown, as well as future works that can extend the contents of this one.

5.1. Conclusions

We think that the DFP is a great opportunity to do whatever you really want and take your capabilities to its maximum, always helped by tutors that can help you with their experience and their point of view. This project is a good chance to experience how it is a real project development and how difficult it could be to carry out a game from the blank and without any expecification what is unique in the degree.

During this project development we have understood how important procedural content generation is in video games and how it could represent a turning point in the industry. There are a lot of games that use techniques for content generation and a lot of enterprises are investing tons of money to investigate new techniques or techniques application which can accelerate the games development. So, this project is a first step into a really huge field that can be used as a simple example for the power of this technology and the possibilities that these kinds of technologies have in fields such as modelling, terrain generation, AI, narrative and so on. For now, there are game examples where the environment or some visual aspects are procedural generated but, as we have done in this project, methods as WFC can be applied to a lot of fields that can add variety to a game and that are practically virgin in the industry. So, that means that this is a huge opportunity to innovate in the games industry.

5.2. Future work

There is a lot to do in terms of developing a tool that can use the WFC algorithm for any kind of games and making the content totally functional for gameplay. In this project we have developed a tool to generate procedural content in AI fields and terrain, but this tool is thought to prove the possibilities of the algorithm. The next step would be creating a kind of test that lets us ensure that the final result is appropriate for a game. When this kind of tools are used for actual games in the industry they always check the functionality before letting the player use the content generated, so the next step will be to create this checking technique for the AI structure generation. After developing the tool for creating the AI structure in any kind of game, another good point would be to apply the WFC to another field in the games development, opening new fields for the content generation. For example, the WFC algorithm could be used for the creation of procedural narratives, in a similar way that we have applied the algorithm to the AI structures generation.

On the other hand, there is a lot of work to do with the demo that we have created for testing the algorithm. It could be transformed into an actual game that could be used not only to show the possibilities of the WFC algorithm but also to be an entertaining experience for itself. We can add a kind of levels, a narrative that could lead the game flow through different phases and a kind of points or objective that could be used to develop features for the main character making the gameplay more deep and interesting. We are thinking about trying to finish the game with the help of some friends and trying to give the game more personality, even making it a complete game, fixing some bugs and polishing these gameplay aspects.

BIBLIOGRAPHY

1. Stalberg, Oskar. *Wave by Oskar Stalberg.* *https://oskarstalberg.com/game/wave/wave.html*
2. Boris The Brave (13th April 2020). Wave Function Collapse Explained. https://www.boristhebrave.com/2020/04/13/wave-function-collapse-explained/
3. Alexandre Thomas & Dimitry Barashev. TeamGantt: Online Gantt Chart Maker Software - Free Forever. (2022). https://www.teamgantt.com/
4. MediaWiki (13th May 2020). UNITY Requisitos previos - MediaWiki. https://wiki.cifprodolfoucha.es/index.php?title=UNITY_Requisitos_previos
5. Stalberg, Oskar (8th October 2016). World generation. Twitter. https://twitter.com/OskSta/status/784847588893814785
6. Stalberg, Oskar (1st November 2016). Building generation. Twitter. https://twitter.com/OskSta/status/793545297376972801
7. Stalberg, Oskar (26th November 2016). Building generation 2.0. Twitter. https://twitter.com/OskSta/status/793545297376972801
8. Stalberg, Oskar (12th May 2017). Island generation. Twitter. https://twitter.com/OskSta/status/863019585162932224
9. Brackeys (14th March 2018). GitHub - Brackeys/NavMesh-Tutorial: Tutorial project files on using NavMesh in Unity. Github. https://github.com/Brackeys/NavMesh-Tutorial
10. J. Millington, I. & Funge. Artificial Intelligence for Games. Taylor & Francis, 2009.
11. Ferocious Industries. (2018, October 23). FREE Medieval Props Asset Pack. Unity Asset Store. https://assetstore.unity.com/packages/3d/props/free-medieval-props-asset-pack-131420
12. JN 3D. (2019, January 25). Medieval barrels and boxes. Unity Asset Store https://assetstore.unity.com/packages/3d/props/exterior/medieval-barrels-and-boxes-137474
13. Lylek Games. (2019, May 14). Medieval Stone Keep. Unity Asset Store. https://assetstore.unity.com/packages/3d/environments/medieval-stone-keep-56596
14. Synty Studios. (2019. July 17). POLYGON MINI - Fantasy Character Pack. Unity Asset Store. https://assetstore.unity.com/packages/3d/characters/humanoids/fantasy/polygon-mini-fantasy-character-pack-122084
15. Asset Store Originals. (2020, June 10). Snaps Prototype | Sci-Fi / Industrial. Unity Asset Store. https://assetstore.unity.com/packages/3d/environments/sci-fi/snaps-prototype-sci-fi-industrial-136759
16. Secession Studios. Dark Synthwave Music - Devour. Youtube. https://www.youtube.com/watch?v=nUtAWeSK28g

17. Efectos de Sonido Sin Copyright. Musica Epica - Sin Copyright . Youtube. https://www.youtube.com/watch?v=ud23ua10ZEI
18. Sharetextures. (2018, November 2). Natural cut stone wall. Share Textures. https://www.sharetextures.com/textures/wall/natural-cut-stone-wall/
19. Sharetextures. (2020, March, 3). Blue leather 18. Share Textures. https://www.sharetextures.com/textures/fabric/blue-leather-18/
20. Share Textures. (2021, June, 8). Red Leather. Share Textures. https://www.sharetextures.com/textures/fabric/red_leather/
21. Sharetextures. (2019, March 19). Wood fine 16. Share Textures. https://www.sharetextures.com/textures/wood/wood-fine-16/
22. Unity. ProBuilder. https://unity.com/es/features/probuilder
23. Brackeys. (2020). MAKING YOUR FIRST LEVEL in Unity with ProBuilder!. Youtube. https://www.youtube.com/watch?v=YtzIXCKr8Wo
24. Unity. PolyBrush. https://unity.com/es/features/polybrush
25. Brackeys. (2018). SCULPT, PAINT & TEXTURE in Unity. Youtube. https://www.youtube.com/watch?v=JWAnQiN9Zkw
26. CO.AG Music. (2018). Futuristic Sci-fi Background Music Copyright and Royalty Free - Orion. Youtube. https://www.youtube.com/watch?v=wrJ0itsyVR8
27. Mxgmn. (2021, December). WaveFunctionCollapse. Github. https://github.com/mxgmn/WaveFunctionCollapse
28. Hamid R. Arabnia, Ken Ferens, David de la Fuente, Elena B. Kozerenko, José Angel Olivas Varela, Fernando G. Tinetti. Advances in artificial intelligence and applied cognitive computing. 2021. https://books.google.es/books?id=h3JIEAAAQBAJ&pg=PA526&lpg=PA526&dq=Maxim+Gumin&source=bl&ots=YOZnzLvmwq&sig=ACfU3U01ySrej5QhXUFyjs6bZy7Vse0osQ&hl=es&sa=X&ved=2ahUKEwil3K-Iwu33AhXNzYUKHQrDDlkQ6AF6BAgZEAM#v=onepage&q=Maxim%20Gumin&f=false

LIST OF FIGURES

2.1. DFP task planner (made with TeamGantt [3]) 7

3.1. Playable and Non-Playable Characters 9
3.2. Main Menu of the game 10
3.3. The main character model 11
3.4. Face to face enemy 12
3.5. Long distance enemy 12
3.6. Case use diagram 19
3.7. Activity diagram 20
3.8. GUI in the initial room 22
3.9. GUI in the middle of the arena (blocking action) 22
3.10. GUI in the middle of the arena (attack action) 23

4.1. Image of 2D content created by WFC algorithm [1] 26
4.2. Appropriate tileset for terrain building with WFC 27
4.3. Different maps examples generated by the WFC algorithm 29
4.4. Variable class and its constructor 30
4.5. Variable's method to calculate its entropy 31
4.6. TileSetGenerator attributes declaration 31
4.7. TileSetGenerator Initialize method 32
4.8. TileSetGenerator Generate method 33
4.9. TileSetGenerator TileElection method 34
4.10. TileSetGenerator SearchNextGridCell method 35
4.11. TileSetGenerator ConstraintPropagation method 36
4.12. AI Variable constructor 38
4.13. Behaviour Block's class constructors and arguments 40
4.14. RunCondition and Run method for the Patrol class 41
4.15. RunCondition and Run methods of the Retreat class 42
4.16. RunCondition and Run methods of the StrategicPositioning class 43
4.17. RunCondition and Run methods of the GetClose and GetAway classes 44
4.18. RunCondition and Run methods of the Attack and Shoot classes 45
4.19. Generation of the decision tree 46
4.20. Decision Trees generated by the WFC algorithm 47
4.21. Attributes of the EnemyAgent class 47
4.22. EnemyAgent constructor 48
4.23. EnemyAgent OnEnable method 49
4.24. GoPatrolling method of EnemyAgent class 50
4.25. Retreat method of enemyAgent class 51
4.26. Get Strategic Position method of EnemyAgent class 52
4.27. StrategicPositioning method of the EnemyAgent class 53
4.28. GetClose and GetAway methods of EnemyAgent class 55

4.29. Attack and Shoot method of the EnemyAgent class 56
4.30. Tutorial and initial room style 57
4.31. Arena structure and textures style 58
4.32. Arena objects style 58
4.33. Choosing the tile set for the terrain generation 62
4.34. Choosing the block set for the behaviour generation 62
4.35. Entering the arena through the portal 62
4.36. Player inside the arena 63
4.37. Fighting the face to face enemy 63
4.38. Fighting the long distance enemy 63