Design and development of a Roguelike game with procedurally generated dungeons and Neural Network based AI.

Video game Design and Development Degree

Technical Report of the Final Degree Project

Author: Andoni Pinilla Bermejo
Tutor: Raúl Montoliu Colás

Castellón de la Plana, July 2018

Abstract

In games, other aspects have always been preferred over artificial intelligence. The graphic part is usually the most important one and it alone can use a huge amount of resources, leaving little space for other features such as AI.

Machine Learning techniques and Neural Networks have re-emerged in the last decade and are very popular at the moment. Every big company is using Machine Learning for different purposes and disciplines, such as computer vision, engineering and finances among many others.

Thanks to the progress of technology, computers are more powerful than ever and these artificial intelligence techniques can be applied to videogames. A clear example is the recent addition of the Machine Learning Agents to the Unity3D game engine.

This project consists of the development of a Roguelike game called Aia in which procedural generation techniques are applied to create procedural levels, and Machine Learning and Neural Networks are the main core of the artificial intelligence. Also, all the necessary mechanics and gameplay are implemented.

Contents

List of Figures 5

List of Tables 6

1 Technical proposal 7

1. 1 Summary 7

1.2 Key Words 7

1. 3 Introduction and motivation 7

1.4 Related Subjects 8

1.5 Objectives 8

1.6 Planning 8

1.7 Expected results 9

1.8 Tools 9

2 Design and investigation 10

2.1 Summary 10

2.1.1 Concept and context 10

2.1.2 Genre and characteristics 10

2.1.3 Target audience 11

2.1.4 Art style 11

2.2 Gameplay 12

2.2.1 Objectives 12

2.2.2 Mechanics and economy 12

2.2.3 Power-ups 13

2.3 Narrative design 13

2.3.1 Story 13

2.3.2 Characters 13

2.3.3 Spiders 13

2.4 Levels and procedural generation 14

2.5 Interfaces 15

2.6 Artificial intelligence in games: state of the art 16

2.6.1 Movement 16

2.6.2 Decision Making 16

2.6.3 Machine Learning and Neural Networks 17
2.7 Platform 18
3 Development **20**
3.1 Procedural generation 20
3.1.1 Cellular Automata 20
3.1.2 Creating the mesh and applying the Marching Squares algorithm 22
3.1.3 Flood fill 24
3.1.4 Connecting rooms 26
3.1.5 Start and finish 27
3.1.6 Setting power-ups, eggs and crystals 29
3.1.7 Perlin Noise 29
3.1.8 Colliders 30
3.1.9 Rocks 30
3.1.10 Procedural Navmesh 30
3.1.11 Results 31
3.2 Art 33
3.2.1 Aia 33
3.2.2 Assets 33
3.2.3 Unifying assets 35
3.2.4 Post-processing 36
3.2.5 Interfaces and menu 37
3.3 Mechanics 38
3.4 Game flow 39
3.5 Artificial Intelligence 40
3.5.1 State Machines 40
3.5.2 Machine Learning and Neural Networks 43
3.5.2.1 Setting up the training 44
3.5.2.2 Training 47
3.5.2.3 Results 49
4 Final results **5**3
5 Conclusions and future plans **5**6
6 References and bibliography **5**7

List of Figures

Figure 1: The Binding of Isaac, one of the most acclaimed Roguelike games 11

Figure 2: Rime and Inside screenshots 12

Figure 3: Far Cry 3: Blood Dragon art style 12

Figure 4: Aia references. In-game reference (left) . Actual face reference (right**)** 14

Figure 5: Resulting level of A. Adonaac's way of generating dungeons 15

Figure 6: Resulting level of Sebastian Lague's way of generating dungeons 15

Figure 7: AI model 16

Figure 8: A simple state machine in a game 17

Figure 9: A simple scheme of a Neural Network layers 18

Figure 10: An example of how the Cellular Automata model works 20

Figure 11: Cellular Automata algorithm 22

Figure 12: Map generated with the Cellular Automata algorithm repeated 9 times 22

Figure 13: Squared room (left). Desired result (right) 22

Figure 14: Pre-built table with all the possible cases of the Marching Squares algorithm 23

Figure 15: Marching square algorithm implemented in the game 24

Figure 16: Result of the Marching Squares algorithm applied to the generated caves 24

Figure 17: Code for the flood fill algorithm and the region detection 26

Figure 18: Different areas found by the flood fill algorithm 26

Figure 19: Corridor line 27

Figure 20: Rooms connected 27

Figure 21: Another example of a procedurally generated level 28

Figure 22: BFS implementation in the game 28

Figure 23: Start and finish of the level marked in orange and cyan 29

Figure 24: Power-ups, eggs and crystals added to the level 29

Figure 25: Perlin noise texture 30

Figure 26: Perlin Noise applied to the wall terrain of the game 30

Figure 27: Colliders 30

Figure 28: Rocks 30

Figure 29: Navmesh surface generated in runtime 31

Figure 30: Different results of the techniques applied 32

Figure 31: Aia model in 3DS Max 33

Figure 32: Aia with the rigging skeleton (left). Aia rigged in position (right) 34
Figure 33: Models before (left) and after (right) applying MK Toon Shader 36
Figure 34: Before (left) and after (right) of the post-processing effect being applied 36
Figure 35: In-game interfaces 37
Figure 36: Main menu 37
Figure 37: Controls menu 38
Figure 38: Animator State Machine of the animations of Aia 38
Figure 39: Melee spiders state machine 40
Figure 40: Shooting spiders state machine 41
Figure 41: State Machine implementation 43
Figure 42: Unity's ML-Agents diagram 43
Figure 43: Basic code for the Action function of the agent 46
Figure 44: Training environment 47
Figure 45: Melee spider NN mean reward over time (steps) 48
Figure 46: Range spider NN mean reward over time (steps) 49
Figure 47: Melee spiders mean damage dealt over time 50
Figure 48: Melee spiders mean damage received over time 50
Figure 49: Range spiders mean damage dealt over time 51
Figure 50: Range spiders mean damage received over time 51
Figure 51: Final screenshot 1 53
Figure 52: Final screenshot 2 54
Figure 53: Final screenshot 3 54
Figure 54: Final screenshot 4 55
Figure 55: Final screenshot 5 55

List of Tables

Table 1: Planning 9
Table 2: Assets used in the game 34

1 Technical proposal

1. 1 Summary

This section is the technical proposal of my Final Degree Project in Video game Design and Development. The project consists of the development of a 3D Roguelike game using Unity3D. The main features are the design of a procedural dungeon generation system and the use of Machine Learning and Artificial Neural Networks for the NPCs (non-player characters) behavior. This artificial intelligence will be implemented with the new Machine Learning agents of Unity3D.

1.2 Key Words

Roguelike, Procedural generation, Neural Networks, Machine Learning.

1. 3 Introduction and motivation

The main goal of this Final Degree Project is the development of a roguelike game using techniques like procedural generation for the dungeons, and Machine Learning and Neural Networks for the artificial intelligence. Roguelike is a video game genre based on dungeon exploration where the game levels are usually procedurally generated. Other features of this genre are difficult to beat enemies, permadeath (the player has to start all over again when he or she dies) and a simple narrative, being the playability the main aspect of the game.

The procedural generation will be in charge of creating simple but unique dungeons, with different shapes and size rooms, different kind of enemies, items and power-ups. In order to achieve that, different kind of pseudo-random algorithms will be implemented.

For the artificial intelligence, the new Machine Learning agents of Unity3D will be used [1] [2]. They use the TensorFlow library of Google which gives you the chance of using neural networks with ease. Using them, a smart behavior will be tried to be simulated in order to make the game difficult. This means running away from preset behaviors, making unpredictable decisions based on the player's playstyle. This also prevents the use of tricks and exploits.

In addition, basic core gameplay features such as mechanics, items, game flow, etc., will be implemented. The aesthetic part of the game won't be left out, but it isn't the main purpose of the project. The main mechanics that will be implemented are running, dodging, shooting, sword fighting, ammo, stamina…

But why Machine Learning and Artificial Neural Networks (or just Neural Networks when in context)? After years of having been put aside, with the progress of technology, Machine Learning techniques have re-emerged and are very popular at the moment. Every big company is using Machine Learning for different purposes and disciplines, such as computer vision, engineering, finances... These techniques allow computer systems to learn autonomously without the need for their decisions to be programmed, even when dealing with large amounts of data. In games, other aspects have always been preferred over artificial intelligence. The graphic part is usually the most important and it alone can use a huge amount of resources, leaving little space for other features such as AI. In fact, Darryl Charles and Stephen McGlinchey discuss that only from one to five percent of resource allocation is

used for artificial intelligence in most games [3]. But now computers are more powerful than ever and Machine Learning techniques such as Reinforcement Learning [4] are starting to be used in the video game industry. A clear example is the recent addition of the Machine Learning Agents to the Unity3D game engine.

The main advantage of using Machine Learning over other AI techniques used in games (such as behavior trees) is that the player won't always expect the enemy decisions. It is not something that is already preset, it can adapt to the player's playstyle, avoiding possible tricks and exploits.

1.4 Related Subjects

- VJ1215 Algorithms and Data Structures
- VJ1216 3D Design
- VJ1227 Game Engines
- VJ1231 Artificial Intelligence
- VJ1234 Advanced Interaction Techniques

1.5 Objectives

- Developing a fun but difficult roguelike game with the main characteristics of the genre, giving primary importance to the gameplay over any aesthetic component.
- Implementing a strong system for procedural dungeon generation that can generate completely unique levels every time you play.
- Implementing a smart artificial intelligence for the NPCs using Neural Network techniques, capable of simulating smart behaviors.

1.6 Planning

The main tasks to be done are represented in Table 1.

Task	Time
Main game implementation. This includes the first steps of the game, the core mechanics, the flow of the game and any implementation referring to it.	70 hours
3D basic asset modelling and searching for free assets to be used.	30 hours
Procedural dungeon generation system, capable of generating different and unique levels every time.	70 hours
Create the basic NPCs with common AI techniques like Finite State Machines, Decision Trees or Behavior Trees.	20 hours
Connect Unity with TensorFlow using the new Unity3D ML agents, and prepare the neural network and the training environment.	20 hours

Make the Neural Network work and training.	40 hours
Documentation and investigation.	40 hours

Table 1: Planning.

1.7 Expected results

The expected result is to have a fully functional game with a robust system for the procedural dungeon generation, which shouldn not use too much computer time to generate them. Also, a smart neural network capable of simulating a smart behavior of the NPCs, typical of roguelike games. The final hope is to have a fun game with smooth mechanics, efficient frame rate and a really well-made and thought difficulty (a result of a well-made neural network).

1.8 Tools

- Unity3D
- Visual Studio 2017
- 3DS Max
- Anaconda
- Trello
- GitHub
- Google Docs / Microsoft Word

2 Design and investigation

2.1 Summary

2.1.1 Concept and context

Aia is the name that the game will receive. It refers to the name of the main character of the game: Aia. She is one of the few people that still live on the Earth. In 2148 a chain nuclear explosion made the Earth almost uninhabitable. Those who had the chance escaped to Mars, where the first outer space human colony was established. Those who didn't died, except for the ones who developed rare mutations that allowed them to survive. But mutations did not only happen to humans, other species suffered from them too, producing the most horrendous creatures the human eye could have ever seen.

Now, in 2374, giant spiders have taken over the Earth. No other species has been able to face them. Not even humans. Today, the human population is below 1 million people and our protagonist is one of them. They survive in little colonies spread around the world trying to hide from all the monstrosities that inhabit the world. But the colony in which Aia lived was recently attacked. She managed to survive but her family and all her friends were massacred in a bloody fight. Now, she is a nomad traveling the world looking for any sign of human existence.

This concept is represented in a roguelike game where the player takes control of Aia trying to survive in the new Earth. Guns, sci-fi swords, frenetic mechanics, countless spiders and tons of blood are the main components of the game.

2.1.2 Genre and characteristics

The roguelike genre's name comes from the famous old game Rogue [5], so roguelike means a game *like Rogue*. It was a procedurally generated dungeon crawler represented with ASCII characters in which the player had to find treasures and kill monsters. Then roguelike games developed into more complex kind of games with characteristics such as:

- One player game.
- Procedurally generated dungeons or rooms, with different sizes, different enemies, items, power-ups…
- Permanent death. Once you die, you have to start all over again.
- Different items and resources, such as health points or ammo.
- ASCII graphics.
- Turn-based combat.
- High difficulty game.
- Little narrative.

But recently, a wave of new roguelike games has risen. They take the main characteristics of the genre such as permadeath and procedural levels, running away from any turn-based combat and are being represented in 2D, 2.5D and 3D, thanks to the progress of technology. They also implement mechanics from other genres such as top-down shooters or hack'n slash games. But the main component that every game of this kind have in common is

the high difficulty of it. The game is difficult from the beginning to the end, there is no room for casual players. Some examples are The Binding of Isaac [6] (Figure 1) and Nuclear Throne [7], two of the most famous new roguelike games. Some people don't consider these games as roguelikes, considering them just as top-down shooters, dungeon crawlers or *Roguelites*.

Figure 1: *The Binding of Isaac, one of the most acclaimed Roguelike games [8].*

In fact, at the International roguelike Development Conference 2008, a definition of roguelike was created (called the Berlin interpretation). The general principle of the definition is that "roguelike" refers to a genre, not merely "like-Rogue" [9]. A list of factors is also defined in order to know if a game is a traditional roguelike, such as permadeath, procedural level generation, being turn-based or being grid-based. So, as Carter Dotson says in Lifewire [10], many roguelikes aren't roguelikes per se, at least by the Berlin interpretation. But a lot of people disagree and the majority of the gaming community automatically thinks about The Binding of Isaac or Nuclear Throne when hearing the word "roguelike".

Aia would fit in this new wave of roguelikes, with real time combat, top-down shooting, 3D perspective, permadeath, procedurally generated levels and high difficulty. This high difficulty of the game makes the use of Neural Networks and Machine Learning very promising. If you want to make a difficult game, using Neural Networks could be better than any other technique, being able to represent unpredictable behaviors. This is what is going to be implemented and tested in the game.

2.1.3 Target audience

Aia is a hard game, where the difficulty is a key feature of it. So the target audience is obviously hardcore players that are willing to die again and again.

2.1.4 Art style

The art of the game won't be the primary aspect of it. This does not mean that it will be left out. So, how can that be achieved? By using simple but effective art, such as low poly. Games like *Rime* [11] or *Inside* [12] (Figure 2) use this kind of style being able to obtain high-quality results without putting too much detail into the models.

Figure 2: *Rime (left) [13] and Inside (right) [14] screenshots.*

But this game has to be a lot more obscure, creepy and futuristic than the others, using dark colors with neon lights (following a retro and synthwave style) for the terrain, with blue, pink and green neon lights dominating the scene. Figure 3 shows a perfect example of the art trying to be achieved.

Also, not all the art is going to be done by the creator of the project. Some free assets are going to be searched in order to save time for other parts of the game. The only problem with this is the potential lack of connectivity between the art of the different assets, but with the use of shaders in order to standardize the art, this problem can be overcome.

2.2 Gameplay

2.2.1 Objectives

The main objective of the game is to kill every spider of the level without dying. There is almost no other objective than to kill and survive.

Figure 3: *Far Cry 3: Blood Dragon art style [15].*

2.2.2 Mechanics and economy

The player has two kinds of weapons always available: swords and guns. The swords are melee weapons while the guns are distance shoot ones. Using the swords consumes stamina and using the guns consumes ammo. These are the two kinds of usable resources of the game. Health points will be the maximum damage that Aia can take. The player has to find a balance

of using melee and ranged weapons in order to beat the spiders. Changing from melee mode to shooting mode will take a charging time. Stamina will recharge with time, while the player will have to find ammo in order to keep shooting. The maximum amount of stamina is 100. Ammo can be unlimited. The main actions that the player can do are:

- Running: Move rapidly. Does not consume stamina.
- Attacking with swords: Consumes 7 of stamina.
- Area attack: Move and attack at the same time. Consumes 40 of stamina.
- Block: Block an attack. Consumes 5 of stamina per attack received.
- Jump: Consumes 8 of stamina per jump.
- Dash to any direction: Consumes 15 of stamina.
- Range shooting: Consumes 5 bullets every second.

2.2.3 Power-ups

In each level can appear power-ups that help the player to beat the game, like health, ammo or stamina power-ups.

2.3 Narrative design

2.3.1 Story

The narrative of the game is not as important as the game itself with its mechanics, and it is not further developed than in the summary and context.

2.3.2 Characters

Aia is the main and only character of the game. She was born after the catastrophe, so she does not even know how the world was before. Giant spiders are the main threat of the human existence, but they are also a good source of food with a lot of proteins. She lost all her family and friends of her village in a massive spider attack that no one could expect. Aia was the only survivor and now she lives as a nomad, defeating spiders for food and escaping from them too. She has short green hair (natural green hair, not dyed, mutations changed a lot of things) and pink eyes. She's a 21-year-old, tall, strong independent woman surviving in a messed up world. In Figure 4 you can find some references on how Aia would be. The left image shows the kind of style in which Aia would be represented in the game. The right image shows how her actual face would be.

2.3.3 Spiders

There are three kinds of spiders:

1. Melee spiders. This type of spiders has to be near you in order to be able to attack you. They are green, fast and produce high pitch sounds.
2. Range spiders. These spiders can attack you from distance. With a pink neon body, they can spit large amounts of radioactive substances in order to defeat enemies.
3. Explosive spiders: Mad spiders that chase the enemies and explode right in front of them. They are characterized by their significative yellow cover.

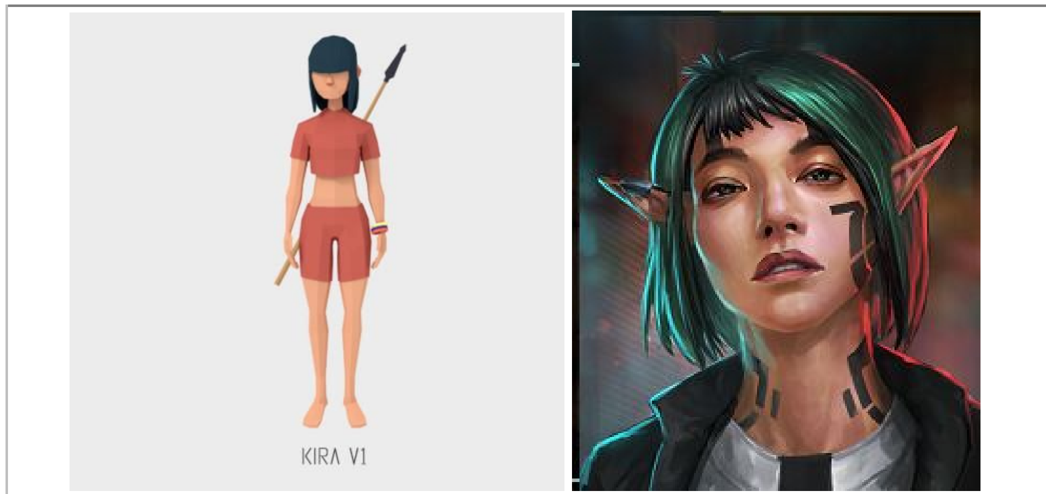


***Figure 4:** Aia references. In-game reference (left) [16]. Actual face reference (right) [17].*

2.4 Levels and procedural generation

Procedural generation is a method to create content dynamically using different algorithms. It is the opposite of any manual creation method and it can be applied to multiple fields, such as video games, simulations, animation and even music. This method is extremely popular in the video game industry right now, as players are often amazed by the power of digital tools to create completely new worlds. According to Wainer, Liu, Dalle and Zeigler [18] new content and gaming scenarios have to be added in order to maintain the player interest and to provide a degree of continuity. Procedural generation can easily achieve this, providing new game content every time someone plays. The most well-known and unthinkable example of a video game using procedural generation is "No Man's Sky" [19]. The game creates a completely unique, open and infinite universe where there can't be two equal planets.

Procedural generation is one of the main characteristics of a Roguelike game. But this type of games don't use it in such a wide way as No Man's Sky. The main purpose of the procedural generation in Roguelike games is to create different caves or dungeons every level and, also, to create a sensation of randomness and newness in the way the player has to play. Even though this sounds really nice, it has its drawbacks too. As Brian Walker, creator of Brogue, told Rock Paper Shotgun in 2015 [20], procedurally created levels will never come close to hand-crafted levels designed by experts.

There are multiple types of procedural generation and different techniques and algorithms to be applied, though. For example, A. Adonaac explains in Gamasutra [21] a way to generate random connected dungeons following this steps: generate a lot of rooms of different height and width inside a circle, keep deleting rooms that collide until no more rooms collide, choose the main rooms and apply a minimum spanning tree technique to the created graph in order to connect the rooms, resulting in levels as the one in Figure 5. Among many other techniques available for procedural generation, Unity3D offers another example of generating dungeons in a set of tutorials by Sebastian Lague [22], with cellular automata as the main pseudo-random algorithm to be applied. These tutorials include 3D rendering and are oriented to more complex and organic dungeons, not just squared rooms. As this is the desired effect for the game, this set of techniques will be the reference for the dungeon

generation in Aia and will be explained as the work is done. A result of this set of techniques is shown in Figure 6.

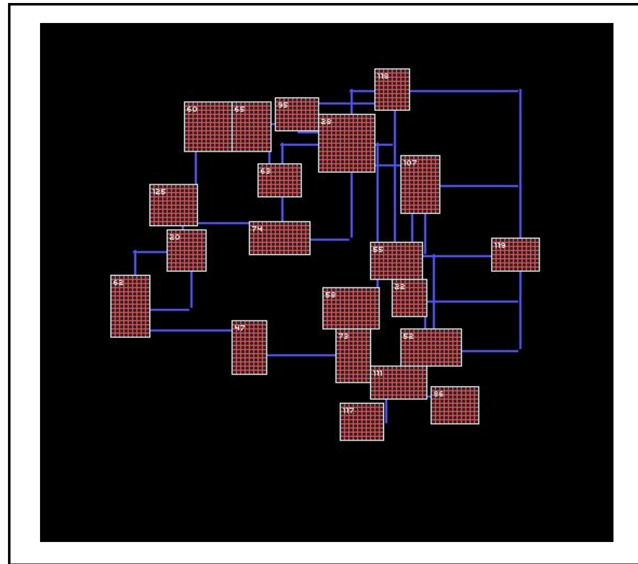


Figure 5: *Resulting level of A. Adonaac's way of generating dungeons [23].*

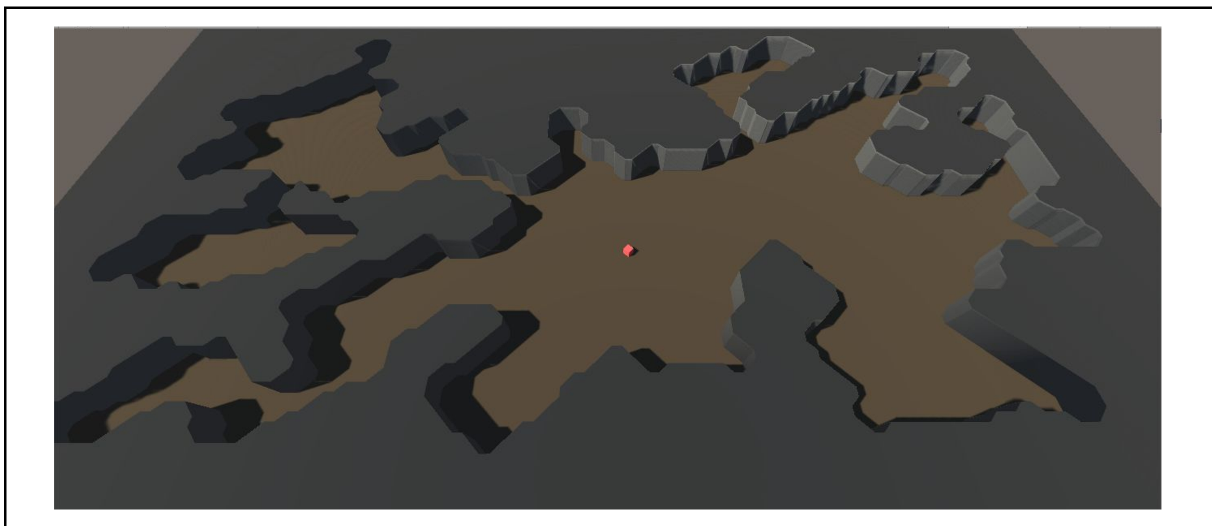


Figure 6: *Resulting level of Sebastian Lague's way of generating dungeons [24].*

2.5 Interfaces

A main menu will be implemented. Also, the game will need a general UI with the economy (health bar, stamina bar and ammo count). They will be placed on the upper left corner in a non-intrusive way. Also, a level count will be shown in the bottom left corner so the player knows which level is playing every time. As the rest of the game style, the interfaces will be very simple but attractive, easy for the player's eyes but without ruining the gameplay experience.

2.6 Artificial intelligence in games: state of the art

Artificial intelligence is about making computers able to perform the thinking tasks that humans and animals are capable of [25]. In games, non-playable characters (NPCs) should behave in way that looks like they have any kind of intelligence. There are a lot of techniques in order to achieve this and they can be grouped into 3 classes: movement, decision making and strategy [26]. These groups and their interactions are shown in Figure 7, a model proposed by Ian Millington and John Funge.

2.6.1 Movement

Movement refers to the motion of the characters after a decision is made. For example, actually chasing the player when the NPC has decided to chase him. Steering behaviors are popular techniques used to make the characters move in a realistic way. Some of them are Seek (steer to a position), Flee (inverse of seek), Arrival (seek, but slowing down while reaching the target), Pursuit (chase predicting the next target location), Collision avoidance, Wall avoidance… These behaviors are not the main purpose of the project and thus, are not going to be implemented. But they have to be used in order to make the spiders chase the player in the dungeons. Luckily, Unity3D's Navmesh [27] already implements these behaviors and makes them easy to use by telling the agent the destination in the Navmesh surface. Furthermore, it also uses a pathfinding algorithm (A* to be precise) in order to decide the correct path to the set destination. Pathfinding algorithms would be between movement and decision making groups in the proposed model.

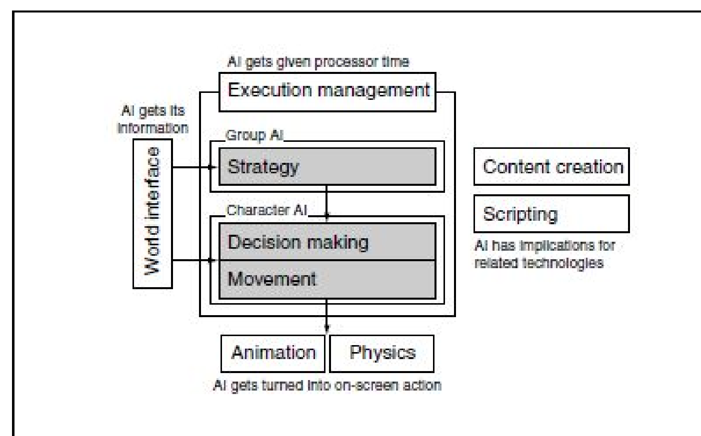


Figure 7: *AI model [26].*

2.6.2 Decision Making

Decision making is basically choosing what to do next. There's usually a set of different behaviors that the agent can perform, and choosing which of those behaviors to use next is what decision making is all about. There are different techniques that are used in video games for taking decisions. Some of the most popular are decision trees, state machines and behavior trees.

Decision trees are basic and fast decision mechanisms represented as a series of choices that lead to a desired effect. The root of the tree is a condition and one of the available choices to that condition is chosen, leading into another condition or into a task to be

performed. For example, the condition could be "Is the player close enough to attack?", and the choices would be "yes" or "no". Those choices could lead into another condition or into an action.

Finite State Machines (FSM) or just State Machines are models of machines that can be in one of their multiple finite states. States are connected together so the machine can transition between them, as can be seen in Figure 8 (a very basic example of how they work). Despite being very easy to write and not very costly they can become harder to scale and to maintain.

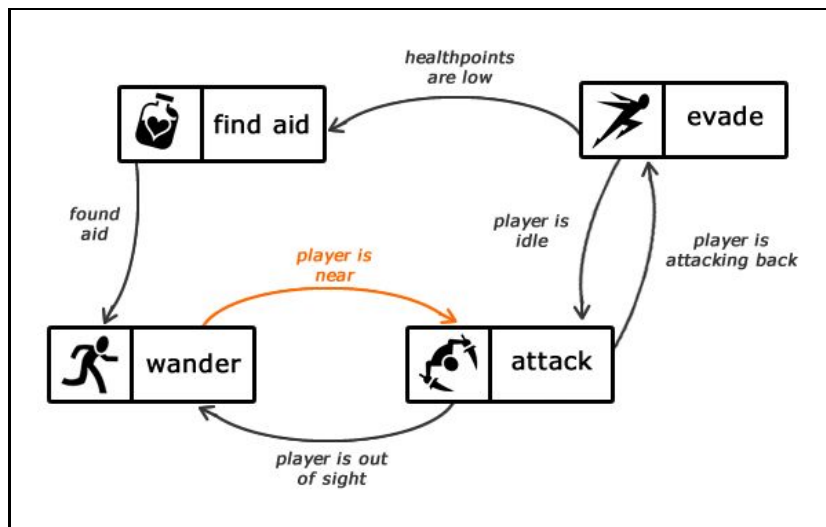


Figure 8: *A simple state machine in a game[28].*

Finally, behavior trees are one the most popular ways for AI in the video game industry. Chris Simpson [29] explains them as trees of hierarchical nodes that control the flow of decision making of an AI entity, being the leaves of the tree the actual commands that control the AI entity, and forming the branches are various types of utility nodes that control the AI walk down the trees to reach the sequences of commands best suited to the situation. They are easily scalable and, once fully understood, are one of the biggest tools for AI decision making.

Probably, behavior trees would be the best option to implement in the game in order to compare them to the AI developed with neural networks. But as they can take a lot of time to learn and implement, simple finite state machines will be used for the basic AI of the spider NPCs. Also, state machines will be used for the animations of the player and spiders and its transitions.

The strategy part of the model refers to the coordination of a whole team of agents. It could be used for this game but, once again, for simplicity purposes and schedule it is not going to be used nor deeply explained.

2.6.3 Machine Learning and Neural Networks

Machine learning (ML) [30] is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. The main applications of Machine Learning until now have been computer vision, finances, text prediction… But now every big company is using it for different purposes. ML could help in the video game creation too by adapting to the player's playstyle and the environment and by not always making the same preset decisions like in the

systems explained before, creating a sensation of uncertainty and unpredictability. This is why it is going to be used in this game.

For this purpose, the Machine Learning Agents of Unity (which are currently in beta, a consequence of ML still being in an initial phase in video games) will be used. In particular, these agents use Reinforcement Learning, a type of Machine Learning that allows the agents to learn by receiving rewards and punishments. These agents don't have a specific objective, they learn by trying and failing. They interact with the environment and, when making a decision and depending on the outcome of that decision, they will receive a reward or a punishment. So, the reinforcement learning technique consists of:

- States (S). A set of the states in which the agent can be.
- Actions (A). A set of actions that the agent can perform.
- Rewards and punishments for the agent's actions in order to know if the action performed was good or bad.

The agents to be trained in the ML agents of Unity3D are neural networks. They are systems based on biological neural networks of our brains. They learn by looking at examples without the need of being programmed to perform a task. They are formed by nodes in different layers (input layer, hidden layers and output layer) like in Figure 9. The network has some inputs, then the hidden layers do some operations which result in the output layer. Combined with Machine Learning techniques they can be very powerful. That's why they are used combined with Reinforcement Learning in the ML agents. By telling them if the output was right or not (rewards and punishments), the operations the next time will be different in order to meet the expectations, creating an abstract representation of the environment in the hidden layers. Marvin Minsky, one of the fathers of AI said that an intelligent machine would tend to build up within itself an abstract model of the environment in which it is placed. And that's basically what neural networks do.

But, as said in the technical proposal, the resources allocated to artificial intelligence in games are usually very low. In fact, only from one to five percent of them are used for AI. Although there are some exceptions like the game Creatures [31], that destined 50% of their resources for it, this is rarely the case. Also, as for Neural Networks (which are going to be used in this game), Darryl Charles and Stephen McGlinchey point out that it is often perceived that they are too computationally expensive to be used in the majority of commercial games, particularly when the AI is to be trained while the user plays [32]. Fortunately, this is changing as technology progresses. Proof of that is the recent addition of the Machine Learning Agents to the Unity3D game engine. Another misconception that Charles and McGlinchey talk about is how people often think of the high computation of NN because of the costly algorithms usually used in them. But there are other algorithms that are not as costly that could be used, such as the algorithm used in the ML agents of Unity3D: PPO [33]. PPO (Proximal Policy Optimization) [34] is a new class of reinforcement learning algorithms which perform comparably or better than state-of-the-art approaches while being much simpler to implement and tune.

2.7 Platform

The targeted platform of the game is PC, offering support for keyboard and controller players.

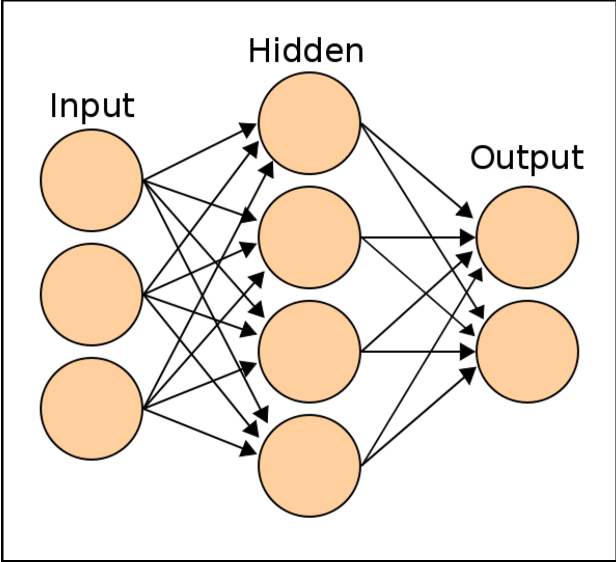


Figure 9: *A simple scheme of a Neural Network layers.*

3 Development

3.1 Procedural generation

A set of different algorithms and techniques, based on Sebastian Lague's tutorials [35], has been implemented in order to reproduce the right caves for the game. Firstly, a Cellular Automata technique [36] has been implemented for generating random caves in a tile map. Then, a mesh is created and the Marching Squares algorithm [37] is used to make the squared cell walls into diagonal and consistent walls. After that, in order to detect the different regions of the massive caves, a Flood Fill algorithm [38] is applied to the map, a graph is created, and they are connected with passages. Finally, a Perlin Noise function [39] is responsible for providing the different levels of terrain. A random seed will be used for all the process in order to be able to repeat a level if wanted.

3.1.1 Cellular Automata

Cellular Automata is a mathematical model initially designed by Stanislaw Ulam and John von Neumann in 1940. Since then it has evolved into different variants and techniques. The main concept of the model is to dynamically evolve a system using discrete steps. The system is usually a grid in which the cells have a finite number of states, such as true or false (like in this game). Depending on a rule previously set by the user, in every step of the process the grid evolves and turns on and off some cells. Figure 10 shows an example of how the model works.

Figure 10: *An example of how the Cellular Automata model works [40].*

Cave generation can be achieved using this model by applying the following steps. Firstly, random values have to be assigned to the tiles of the grid (1 for wall tiles and 0 for floor tiles), except for the grid limit cells which will always be wall tiles in order to have closed caves. Also, despite being random, the percentage of desired land and percentage of desired walls can be set. After that, a step rule to smooth the random map will be applied. This step rule goes through every cell in the room (except limit wall tiles again) and, if it has 4 or more floor (0) neighbors, it will become a floor. If it has not more than 4 neighbors, it will become a wall (1). Figure 11 shows a possible implementation of the algorithm.

This smooth filter is repeated a desired number of times, depending on the size of the grid and the wanted results. Coming up with this number is not really a matter of science but a

matter of trial and error. For example, depending on the number of times that the algorithm is repeated, it can generate from very closed caves to very open caves.

But sometimes more control over the shapes and the regions of the dungeons is wanted, as it is in this game. With this algorithm alone it is difficult to achieve a map with different rooms that does not look very irregular. To solve this problem, a simple approach is taken: generate different small open caves and then glue them together. The map generated with this technique (Figure 12, in which every tile in the map is represented as a green cube) lets you have more control over the size of the map, the number of rooms and the position of the rooms, without ever stop being procedural. Having control over this means well-crafted dungeons, where every part of the cave can be used as a room with different properties.

```
GenerateCelularAutomataMap(room_width width, room_height height, percentage
                           landPercentage) {
        roomMap[] = new array of dimensions width and height
        auxMap[] = new array of dimensions width and height

        //First random fill map
        random = new Random object with random seed

        for every tileX in roomMap’s width:

            for every tileY in roomMap’s height:

               //If it is a border, always walls
                if it is a border of the map:
                    roomMap[tileX, tileY] = 1

                //If it is not a border, random tile
                else:
                    roomMap[tileX, tileY] = random integer between 0 and 1
                                            taking into account landPercentage

        //Smooth the map
        for every time the smooth filter is applied:

            for every tileX in roomMap’s width:

                for every tileY in roomMap’s height:

                    //Function that gets the neighbour wall tiles
                    neighbourWallTiles = get the number of neighbor wall tiles
                                         of tile (tileX, tileY);

                    if neighbourWallTiles is higher than 4:

                        //Store the result in an auxiliar map in order to
                        //prevent contaminated results
                        auxMap[tileX, tileY] = 1

                    else if neighbourWallTiles is lower than 4:
```

```
                    auxMap[tileX, tileY] = 0

            //The result of applying the smooth filter is stored in roomMap
            roomMap = auxMap;

        return roomMap;
```

Figure 11: *Cellular Automata algorithm.*

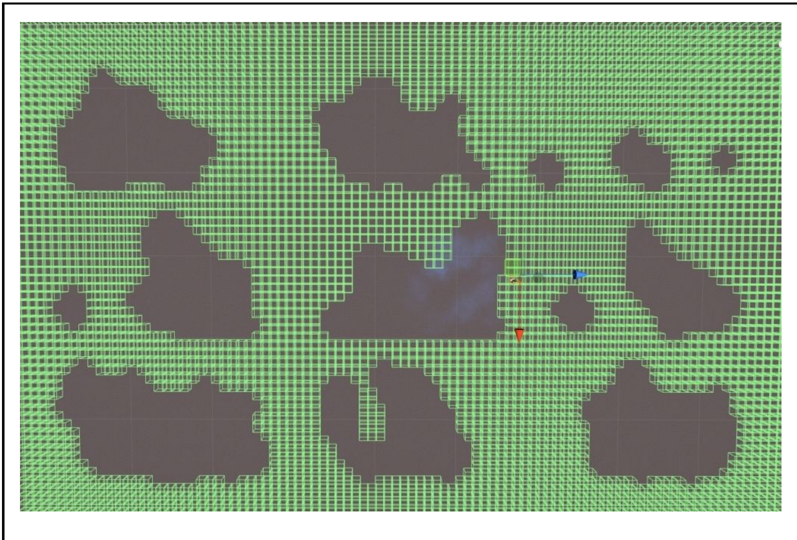


Figure 12: *Map generated with the Cellular Automata algorithm repeated 9 times.*

3.1.2 Creating the mesh and applying the Marching Squares algorithm

In Unity3D meshes are created with triangles. And, in order to create meshes programatically, a set of vertices and a set of triangles are needed. So a square formed by two triangles is going to be generated for every wall tile in the map. Those triangles are going to be stored in a triangle list and their corresponding vertices in a vertices list. This is a simple approach that has a big problem: walls are squared in diagonal parts of the map as can be seen in Figure 13. To solve it, a Marching Squares algorithm is going to be used.

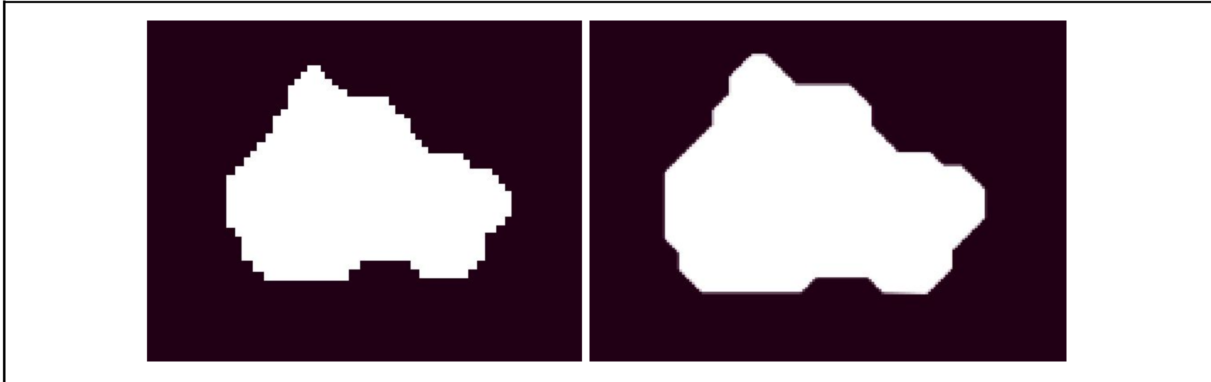


Figure 13: *Squared room (left). Desired result (right).*

Marching Squares [41] is a computer graphics algorithm that generates contours for a two-dimensional scalar field, an array for example. The map generated by the cellular automata algorithm is going to be used. But in this case, every position in the map will not be represented as a square. Instead, every position will represent a corner of a square with a 1 for walls and a 0 for floor tiles. This way, the Marching Squares algorithm can be applied, following the next steps for every formed square:

1. With the numbers of the corners of the square (0 or 1) a 4 bit binary index is built following a clockwise direction, starting in the top-left corner and finishing in the bottom-left corner.
2. Using this built index, a pre-built table (Figure 14) with all the possible cases is consulted. For example, if the corners are (1,0,0,0), the index will be 1000 and the corresponding case would be 8. Depending on the result, different geometry will be built.

Figure 15 shows a basic implementation of the algorithm in the game. *Square* is a data structure that stores all the corners information, the index built from the corners, etc. Figure 16 shows a result of the algorithm applied to the caves of the game with the corresponding created mesh.

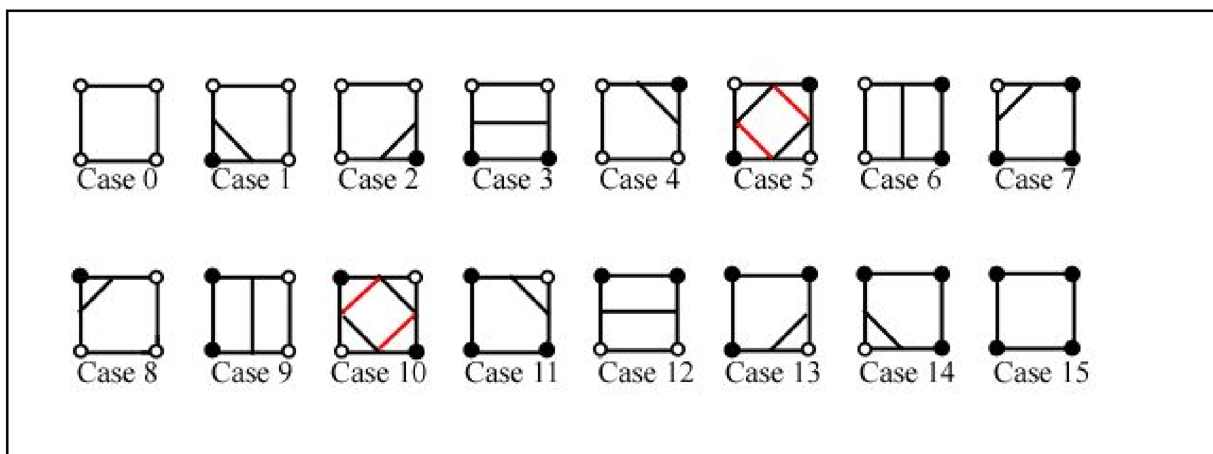


Figure 14: *Pre-built table with all the possible cases of the Marching Squares algorithm [42].*

```
MarchingSquaresAlgorithm(Square square)

   switch 4bit binary index translated into decimal:

      case 0:
         break
      //1 activated point cases
      case 1:
         make mesh from square’s points centreLeft, centreBottom and bottomLeft
         break
      case 2:
         make mesh from square’s points bottomRight, bottomRight and centreRight
         break
      case 4:
         make mesh from square’s points topRight, centreRight and centreTop
         break
      case 8:
```

```
        make mesh from square's points topLeft, centreTop and centreLeft
        break
    //2 activated point cases
    case 3:
        make mesh from square's points centreRight, bottomRight, bottomLeft and
                                        centreLeft
        break
    case 6:
        make mesh from square's points centreTop, topRight, bottomRight and
                                        centreBottom
        break

    //
    //repeated with every case of Figure 14
```

Figure 15: *Marching square algorithm implemented in the game.*

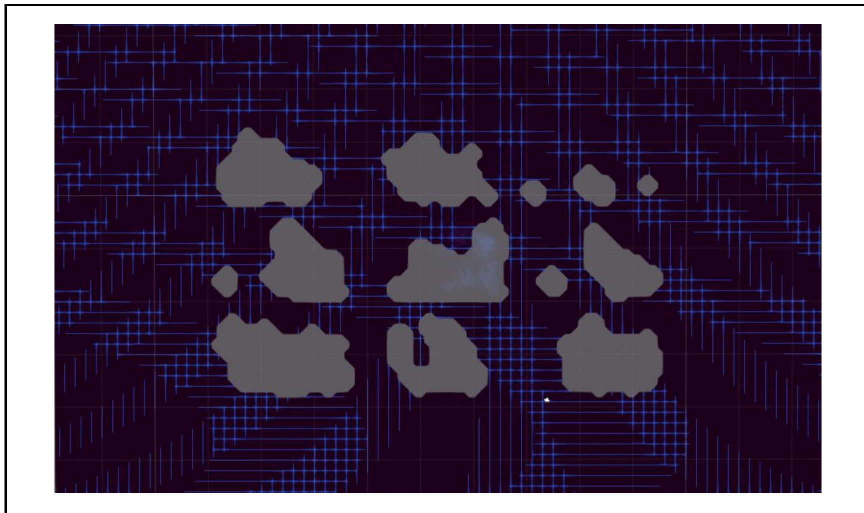


Figure 16: *Result of the Marching Squares algorithm applied to the generated caves.*

3.1.3 Flood fill

Flood fill [43] is an algorithm that searches for connected areas in an array. It is often used in photography and drawing software like Photoshop or Gimp for the "bucket" tool, which fills an area with another color. It is a very simple algorithm. It only takes as arguments the area color, a start point and a color to replace the area with. To do that, every neighbor of the starting point in the array is searched. If they are the same color as the starting color, they are painted with the replacement color and another flood fill is applied to that point, searching again the neighbors of the given point. This is usually implemented with a queue to store the points and it is repeated until the area finishes.

This algorithm is necessary in the game to search for all the generated rooms because, although we tried to control the number of created rooms, cellular automata can generate more small rooms. For example, in the example of Figure 16, the cellular automata algorithm was applied nine times, but there are 13 rooms. For the case of the game, the tiles of the map don't really need to be replaced with another color (in fact, there are no colors, just walls and floor tiles). Just the floor areas will be searched in order to separate the rooms. These areas

will be stored in objects called Room in order keep track of the tiles of each room. To get all the regions, the algorithm needs to be run the necessary number of times until all areas have been detected. The algorithm is represented in C# code for the game in Figure 17.

By using this algorithm, small detected areas (floor areas or even wall areas) can be deleted in order to avoid too small rooms or obstacles. Also, the detected rooms of the example can be seen in Figure 18. These rooms will be the nodes of a graph that will need to be connected.

```
FloodFillAlgorithm(start_tileX startX, start_tileY startY){

        tiles = new list_of_tiles
        zoneMapTiles = new array of width map.width and height map.height //array to put
                                                                   the tiles of the region
        tileType = map[startX, startY]; //type of the start tile

        tileQueue = new Queue //queue to iterate through the algorithm
        tileQueue.Enqueue(tile(startX, startY));
        zoneMapTiles[startX, startY] = visited;

        while tileQueue is not empty:
            tile = tileQueue.Dequeue();
            tiles.Add(tile);

            for every neighbor of the tile:

               if the neighbor is inside the map:

                     if the neighbor is not visited && the neighbor's type == tileType:

                            zoneMapTiles[x, y] = visited
                            tileQueue.Enqueue(neighbor)

        return tiles;


//Get all the regions of type "tileType" (0 or 1) running the floodFill algorithm the
necessary number of times and storing the areas in Rooms

getAllRegions(type tileType):

        regions = new list of regions //List of all the regions which are list_of_tiles
        visitedTilesMap = new array of width map.width and height map.height //aux map
                                              for the already visited tiles of the map

        for every tileX in map's width:

             for every tileY in map's height:

                if visitedTilesMap[tileX, tileY] is not visited && map[x, y] ==
                                                                  tileType):

                    newRegion = FloodFillAlgorithm(tileX, tileY); //Get the tiles of the
```

```
                                                        region with the flood fill algorithm
                regions.Add(newRegion);

                if tileType == 0: //If its a room, add it to the room list
                    newRoom = new Room with tiles of newRegion;
                    rooms.Add(newRoom);

                foreach tile in newRegion: //Set the tiles of the region to visited

                    visitedTilesMap[tile.tileX, tile.tileY] = visited;


    return regions;
```

Figure 17: *Code for the flood fill algorithm and the region detection.*

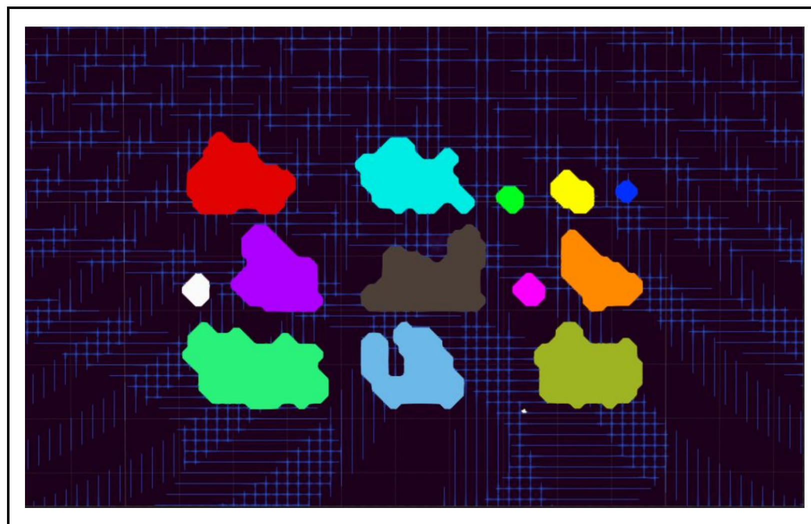


Figure 18: *Different areas found by the flood fill algorithm.*

3.1.4 Connecting rooms

Now that the rooms have been detected it is time to connect them with passages. First a graph has to be created with the rooms as nodes of the graph. It has to be an undirected strongly[1] connected graph. But it can't really be created randomly. Rooms need to be directly connected with their closer ones. To do that, tiles of every room are compared with tiles of every other room. The closest ones are connected. This is not computationally heavy as the number of rooms and tiles in the game levels is really low. Now the rooms are connected, but the graph may not be strongly connected. To ensure connectivity, a main room is chosen. All rooms should be able to get to that room by going through the graph. If they can't, a connection has to be made so they can access it. For this case, the central room of the levels will be the chosen one as the main room.

[1] Strongly connected means that every node is accessible from every node.

At this point, the undirected strongly connected graph is created, but just abstractly. The tile map and the mesh need to be updated with proper corridors. To do that, a line is drawn from the closest tiles of the connected rooms. With the simple equation of a line ($y = mx + n$) it is easy to compute which tiles of the map would form the corridor. Figure 19 is a very illustrative example of this. But not only the marked as red tiles need to be corridors because they would be too narrow. Also the neighbor tiles will be chosen in order to make wider corridors. The finally connected rooms would be the ones shown in Figure 20.

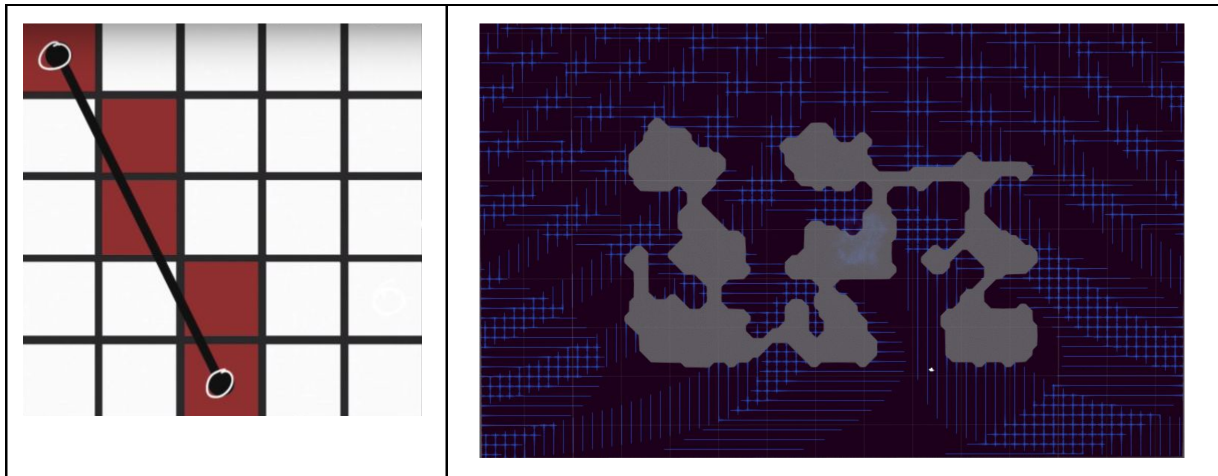


Figure 19: *Corridor line [44].* ***Figure 20:*** *Rooms connected.*

3.1.5 Start and finish

The rooms are now created and the graph is finished. So it is time to place the start of the cave (where the player will appear) and the finish of the the cave (where the player transports into a whole new level). Those rooms need to be the farthest ones. The used example can be confusing. It is not that simple as using the first created room as start and the last created one as finish. Rooms can be connected in other ways and Figure 21 is a perfect example of why that wouldn't work. The solution is performing a Breadth First Search [45] for every node in the graph. Breadth First Search is chosen because the graph can contain cycles.

Breadth First Search (BFS) is an algorithm that searches a graph by exploring all of the neighbor nodes at the present depth before moving on to the nodes at the next depth level. This way, by applying BFS to a room (being the room the root node in the algorithm), all distances to the room can be calculated. The farthest distance and the corresponding farthest room are stored in the room properties. BFS is applied to every room in the graph, so now the farthest room distance for every room is known. The higher distance of all rooms is chosen, that room will be the start point and the corresponding stored farthest room, the end of the cave. The commented implementation of the algorithm for the rooms is in Figure 22. The resulted farthest rooms in the ongoing example are in Figure 23.

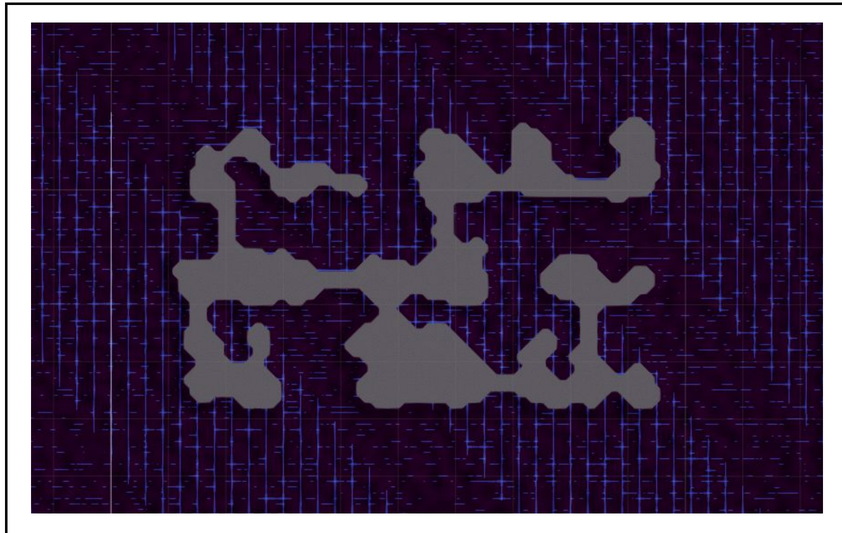


Figure 21: *Another example of a procedurally generated level.*

```
//Breadth First Search
void bfs(graph finalRooms, root index):

        //Set all the nodes to not visited, to distance of 0 and null parent
        foreach room in finalRooms:
            room.visited = false
            room.distance = 0
            room.parent = null


        //Set the root node to visited
        finalRooms[index].visited = true
        finalRooms[index].distance = 0
        finalRooms[index].parent = null

        q = new Queue of rooms //Create a queue
        q.Enqueue(finalRooms[index]); //Enqueue the root node

        while q is not empty:

            currentRoom = q.Dequeue(); //Dequeue node

            for every neighborRoom of currentRoom:
                //If it is not visited, it is set to visited, the distance and
                //parents are set, and the node is enqueue
                if neighborRoom is not visited:
                    neighbourRoom.visited = true
                    neighbourRoom.distance = currentRoom.distance + 1
                    neighbourRoom.parent = currentRoom
                    q.Enqueue(neighbourRoom)
```

Figure 22: *BFS implementation in the game.*

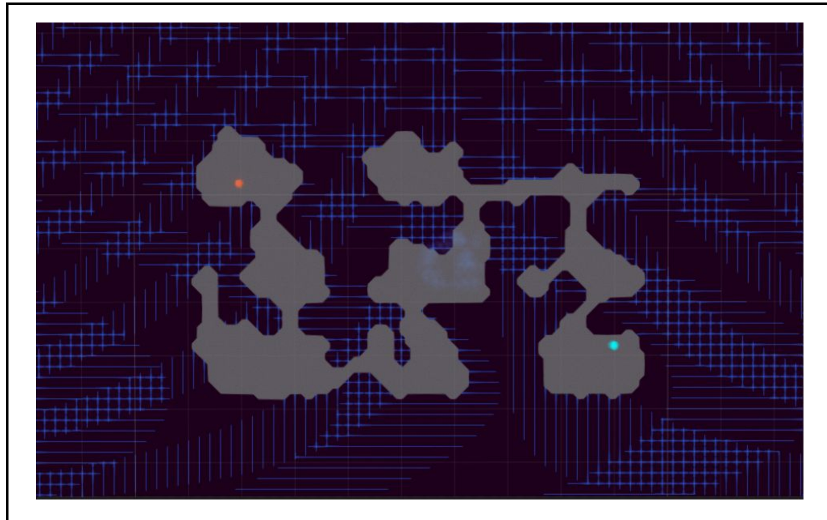


Figure 23: *Start and finish of the level marked in orange and cyan.*

3.1.6 Setting power-ups, eggs and crystals

Setting the power-ups, the eggs and the crystals in the map is as easy as instantiating them on random tiles of the rooms. The number of power-ups depends on the level, if it is a low level (easy) the number would be greater than in higher levels (more difficult). The number of eggs and crystals depends on the size of the room. As the room is bigger, more eggs and crystals can be added. But of course, it is not a fixed number, it varies between a random range. Figure 24 shows a result of this addition.

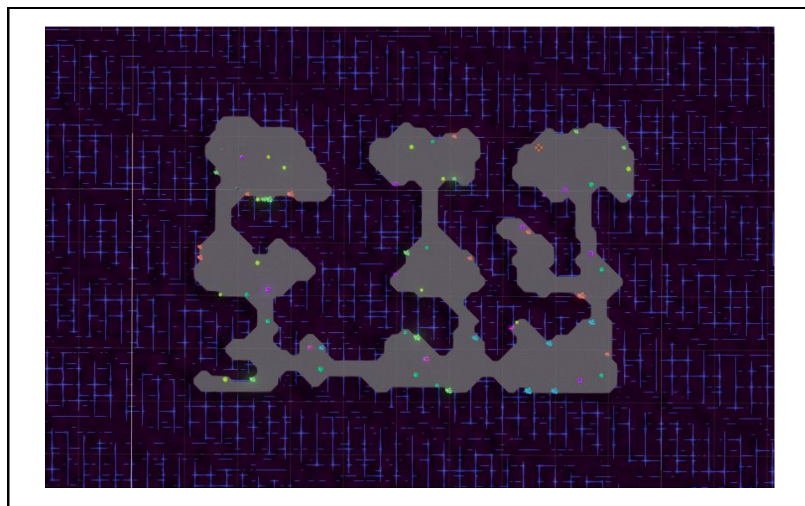


Figure 24: *Power-ups, eggs and crystals added to the level.*

3.1.7 Perlin Noise

Finally, in order to give some level to the terrain and to create more organic shapes, a Perlin Noise filter is applied to the walls. Perlin Noise [46] is a gradient noise texture (as in Figure 25) created procedurally. It is used in the computer graphics field for terrain generation. Depending on the noise color of a point (from white to black with gray scale), the point will have a different height by linearly interpolating the color. Unity3D provides a really easy way

of using it by calling *Mathf.PerlinNoise()*. Figure 26 shows the game terrain with the filter applied.

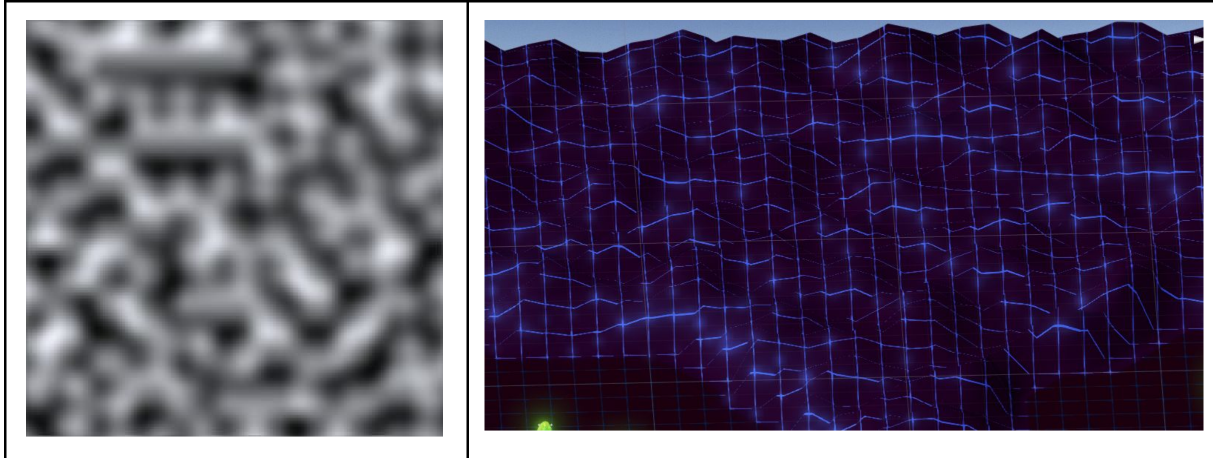


Figure 25: *Perlin noise texture.* ***Figure 26:*** *Perlin Noise applied to the wall terrain of the game.*

3.1.8 Colliders

Colliders have to be added so the player can't jump out of the rooms. Colliders as in Figure 27 are created like a mesh, but without rendering it. Triangles are raised between the mountains and the rooms putting a limit to the play zone.

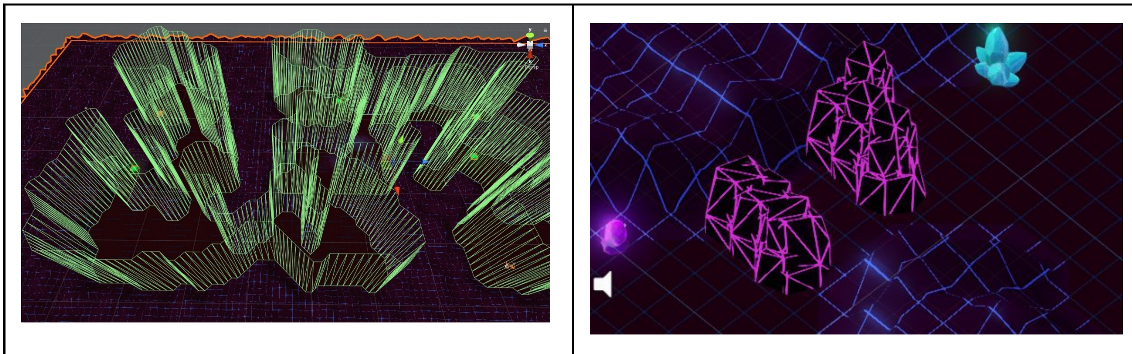


Figure 27: *Colliders.* ***Figure 28:*** *Rocks.*

3.1.9 Rocks

Rooms should not be accessible before the player completes the previous ones by killing the spiders that appear in the corresponding rooms. With the objective of separating rooms, rocks will be instantiated in the corridors that connect rooms. When the player completes a room, the rocks blocking the connections to the neighbor rooms are destroyed, letting the player advance in the level. Rocks can be seen in Figure 28.

3.1.10 Procedural Navmesh

In Unity3D, navmesh is not prepared to be generated in runtime and it can cause a lot of problems. It is true that it can be done by setting obstacles called "carves" (invisible cubes) where the navmesh is not wanted, i.e. instantiating carves in the wall tiles of the map in this

case. But it generates navmeshes with a lot of undesired triangles (thousands), making it impossible to perform fast pathfindings and steering behaviors. Here, the new navmesh surface [47] (which belongs to the new navmesh components of Unity3D for the newer versions) comes to the rescue. It can be attached to a GameObject and allows to change the tile size of the navmesh, making it easily customizable and it does not generate any undesired triangles as it can be seen in Figure 29, where the blue part is navmesh where the spiders will be able to move.

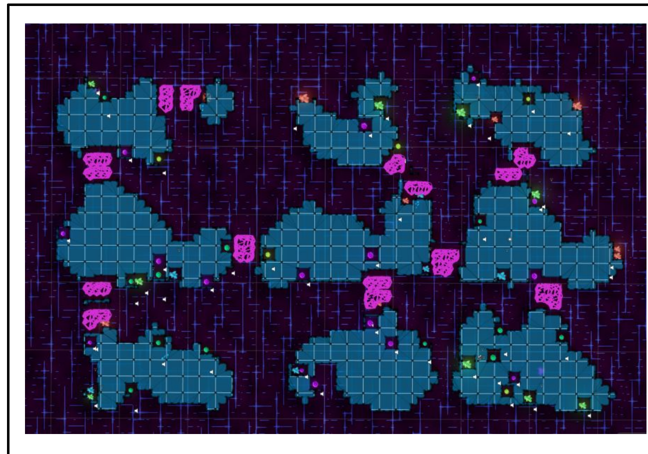
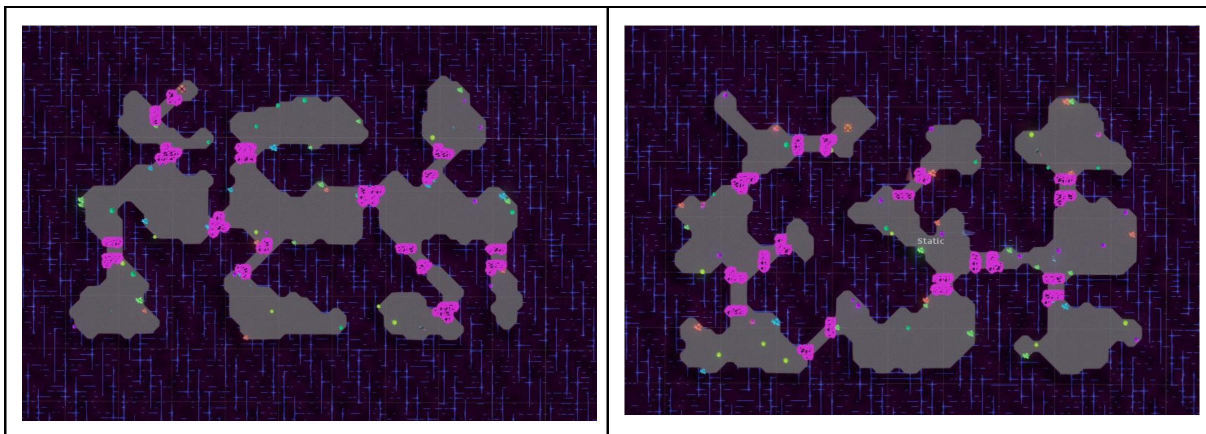


Figure 29: *Navmesh surface generated in runtime.*

3.1.11 Results

The result of these techniques being applied is a really strong way of creating caves and dungeons. They are totally procedural, but can be controlled by changing some parameters. For example, for bigger rooms in the caves, the minimum width and height of a cave can be changed. Also, despite not being used for this game, caves with a lot of rooms can be also generated just by changing the number of rooms wanted in the X axis and the number of rooms in the Y axis. These dungeons are generated really fast, needing just fractions of a second to be generated. This time slightly increases as the number of rooms increases too. But it is not really a problem unless you generate dungeons with hundreds of rooms and it is not the case of this game. Even then, a simple loading page could be implemented.



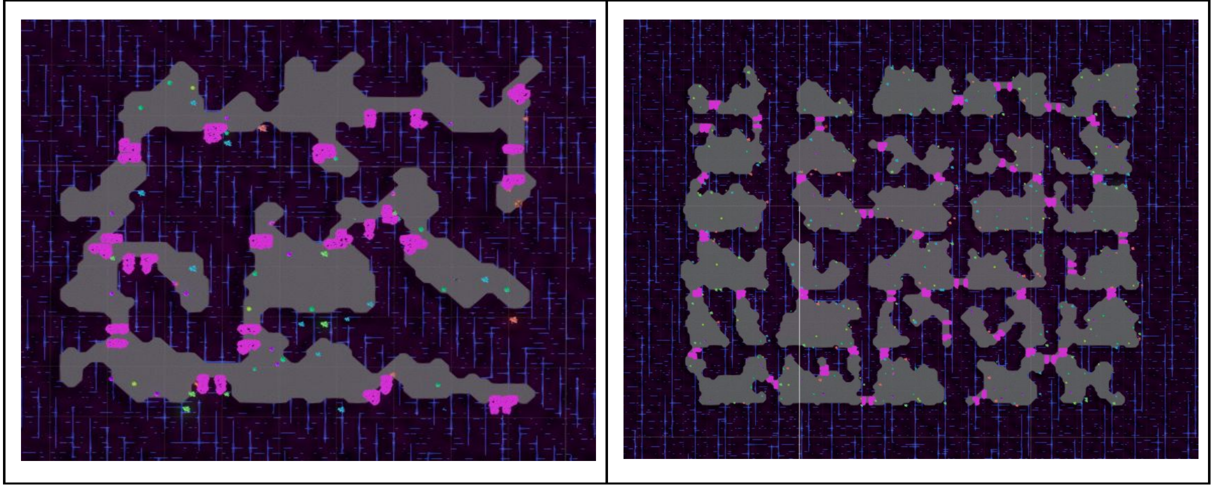


Figure 30: *Different results of the techniques applied.*

3.2 Art

This part of the project refers to the art that has been developed or just used in the project: models, interfaces, materials, shaders, sounds… This art follows a specific styled described in the design document. Simple models with low number of polygons, bright and neon colors trying to simulate a retro/synthwave/retrowave effect. This is achieved by applying certain shaders to the materials of the models and by using typical retro colors such as pink, blue or green.

3.2.1 Aia

Aia is the main character of the game. She had to be a strong, ninja-like and obscure character. Because of that, she was modeled (Figure 31) with really strong legs, so performing acrobatic movements like ninjas do would be credible. Also, another characteristic part is her green hair with a long fringe, creating a sensation of mystery and darkness. For the model creation, 3DS Max software was used. It was used for the rigging process too (assigning the skeleton and respective bones to the parts of the character so animations can be applied, as in the left part of Figure 32). The right part of Figure 32 shows Aia in-game already rigged performing an animation.

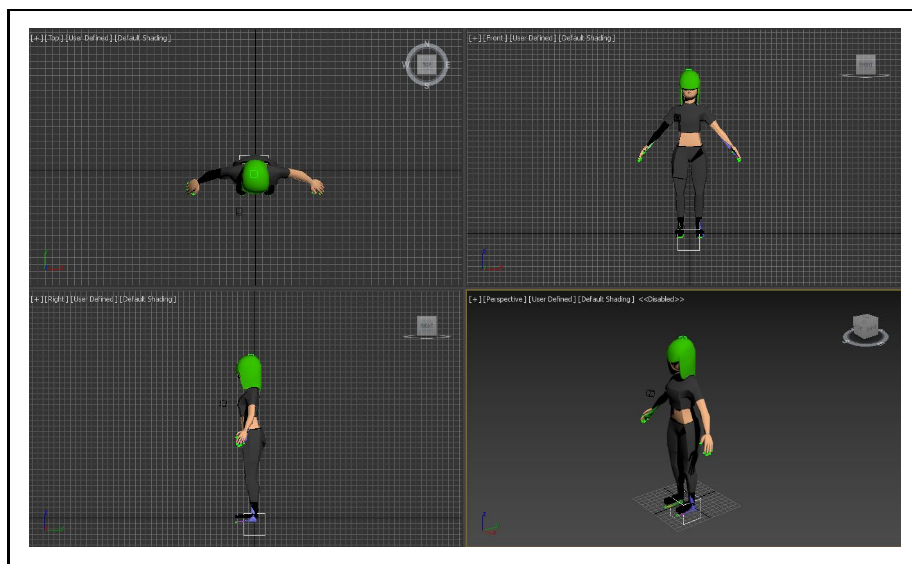


Figure 31: *Aia model in 3DS Max.*

3.2.2 Assets

The artistic part of the game is not the main purpose of the project. That's why a lot of the assets of the game have been downloaded from the Unity Asset Store [48], most of them provided for free by the developers for this project and others were already free. They are all listed in Table 2, with their names and authors, comments on how and why they have been used and links to their respective pages.

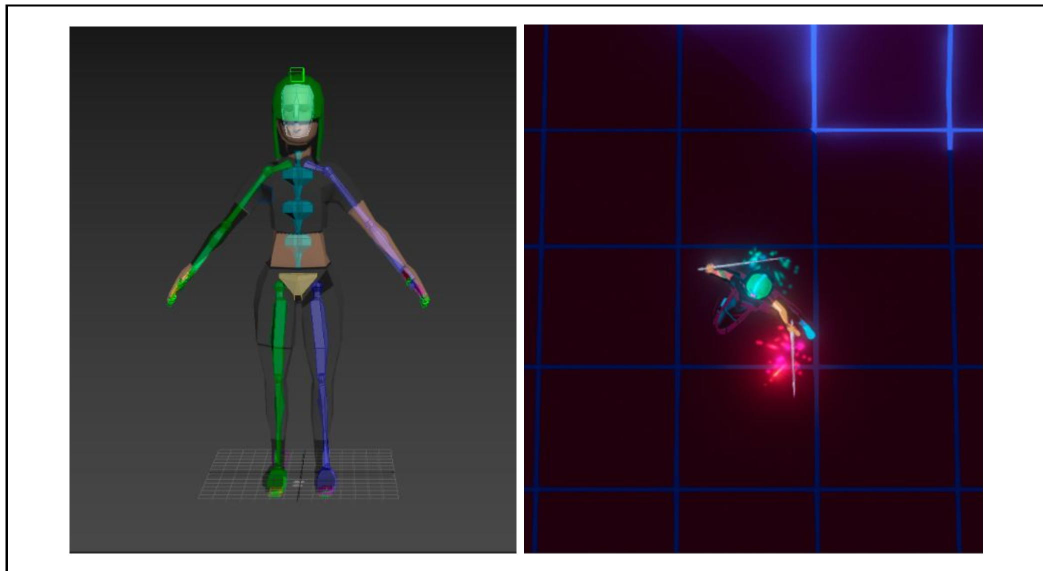


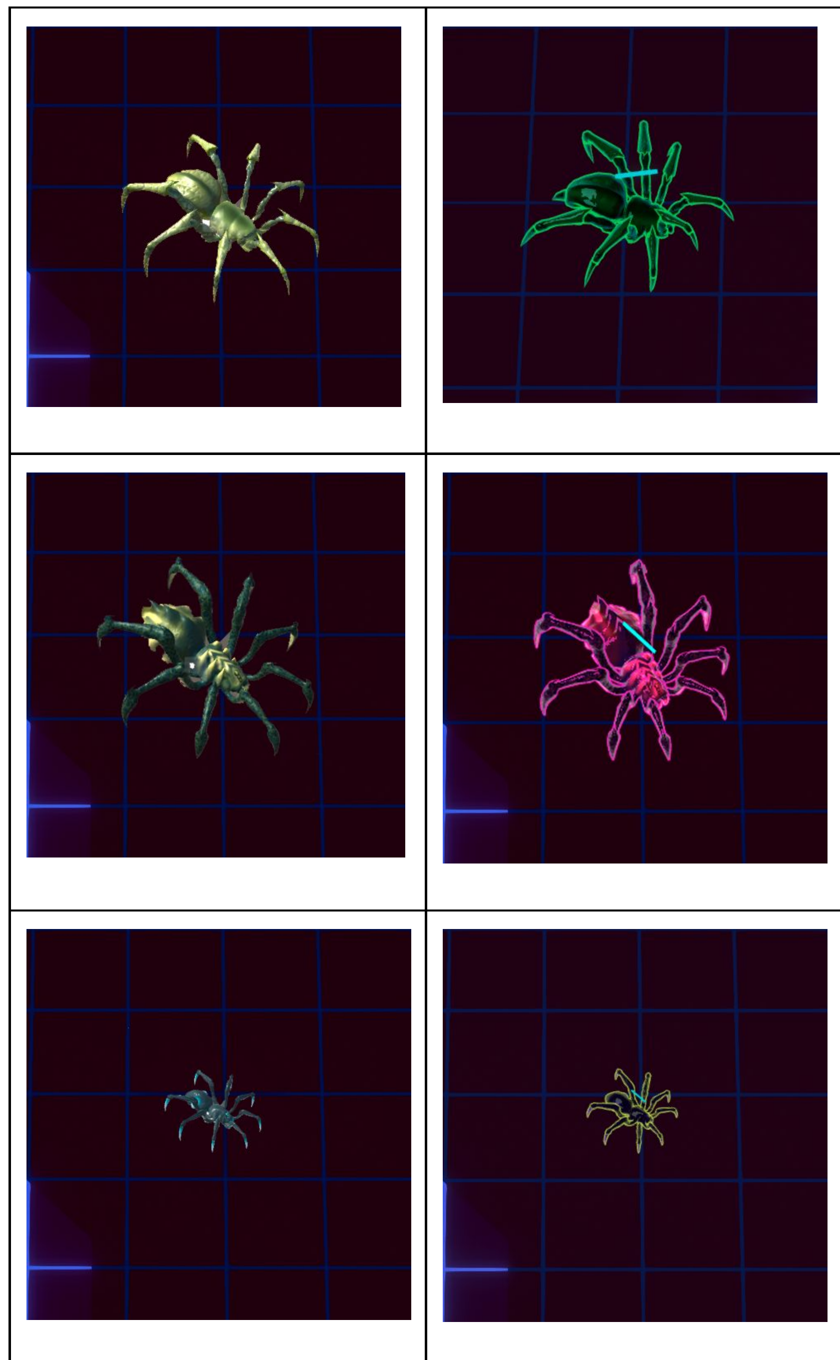
Figure 32: *Aia with the rigging skeleton (left). Aia rigged in position (right).*

Asset name	Comments	Url
Ninja Warrior Mecanim Animation Pack by Explosive	Animation pack used for the ninja melee animations of Aia.	bit.ly/AiaNinja
Sci-Fi Gun by Grasbock	Gun used for the shooting gun.	bit.ly/AiaGun
RPG Spider Pack v3.0 by Vagabond	Models and animations used for the NPC spiders.	bit.ly/AiaSpiders
Retro Wireframe by David Filipe	Shader used for the floor and terrain.	bit.ly/AiaWire
Translucent Crystals by SineVfx	Models used for the crystals.	bit.ly/AiaCrystal
Low poly styled rocks by Daniel Robnik	Rock models for the room separators of the game.	bit.ly/AiaRock
Retro Power Up Particle Pack by Anthony Avvento	Sprites and shaders for the power-ups.	bit.ly/AiaPowerUp
Stellar Sky by Wolfniey	Skybox used in the menus.	bit.ly/AiaSky
Low-poly anime sword by Erbgameart	Models used for the swords of Aia in the game.	bit.ly/AiaSwords
MK Toon Free by Michael Kremmel	Shader used for the majority of the game assets.	bit.ly/AiaToon
512 Sound Effects (8-bit style) by SubspaceAudio	Sounds used for the sound effects in the game.	bit.ly/AiaSounds

Table 2: *Assets used in the game.*

3.2.3 Unifying assets

Using assets from different origins and from different developers can make it difficult to make them look well together. Especially if the desired style of the art of the project is different than the style in which they were created. For example, spiders were too realistic for this game. In order to solve these problems, a toon shader (which is listed in Table 2, MK Toon Shader) is applied to most of the materials of the game. With this shader it is easy to create toon-ish neon-like materials. The color of the material, the emission color, the light, the shadows and shadow colors, the specular color, the shininess, the rim and the outline are some of the editable parameters of the shader, being the outline the most important one of all as it creates the neon-like effect. Figure 33 shows the models before and after applying the toon shader. Also the used power-ups with this neon-like style are shown.



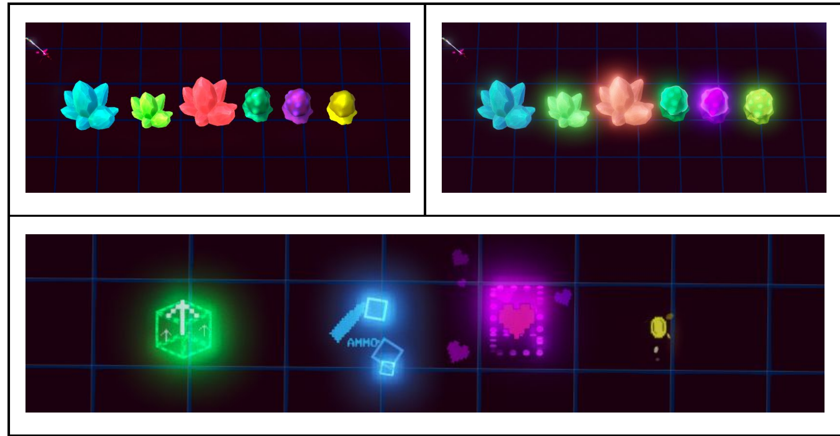


Figure 33: *Models before (left) and after (right) applying MK Toon Shader.*

3.2.4 Post-processing

Unity3D's Post-processing Stack [49] is a set of set of effects that are combined into the same post-processing pipeline. These effects are applied over the video game frames in order to make them look better. The applied effects in this game are:

1. Antialiasing, a filter to smooth sharp edges.
2. Ambient Occlusion, a technique to calculate how spaces are exposed to light in order to create more realistic shadows.
3. Bloom so that lights are enhanced and the retro look is increased.
4. Color grading in order to unify assets, particles and everything that happens in the screen, providing more bluish tones for everything.
5. A very subtle vignette effect, to create a more cinematic look.

The before and after of these effects being applied is shown in Figure 34.

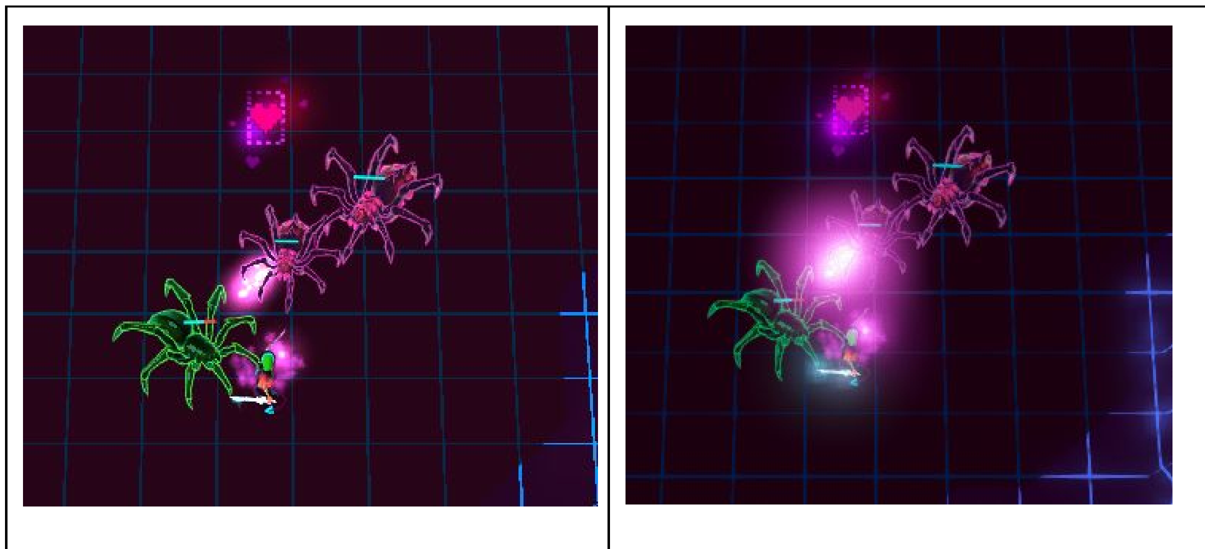


Figure 34: *Before (left) and after (right) of the post-processing effect being applied.*

3.2.5 Interfaces and menu

The user interfaces of the game had to be really simple so the player does not get distracted. Figure 35 shows the accomplished result. The center lever interface is only shown at the start of every level with an horizontal transition (coming from right to left). The font used for that text is Lazer84 [50]. In the upper-left corner, a pink bar for the health points, a green bar for the stamina points and the number of bullets left are placed. In the upper-right corner, there is a text and an icon representing the number of coins that the player has gotten. Finally, in the button-left corner the number of the level is written so the player knows what level is playing at any given time. The font used for the bullet number and the last level number is Retro Computer [51].

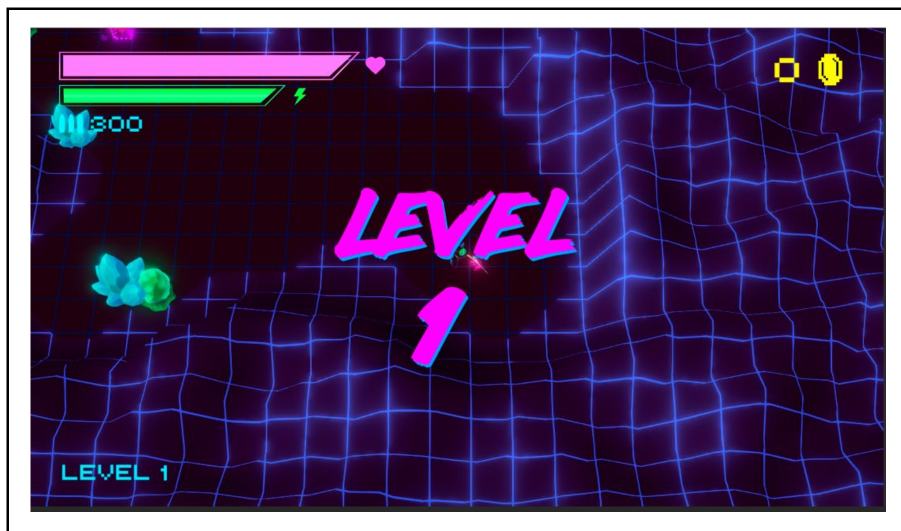


Figure 35: *In-game interfaces.*

A main menu has been implemented too (Figure 36), with the option to exit the game, to play versus finite state machines and to play vs neural networks. The used fonts are the ones listed before. Also the skybox listed in Table 2 is used to achieve a sensation of depth.

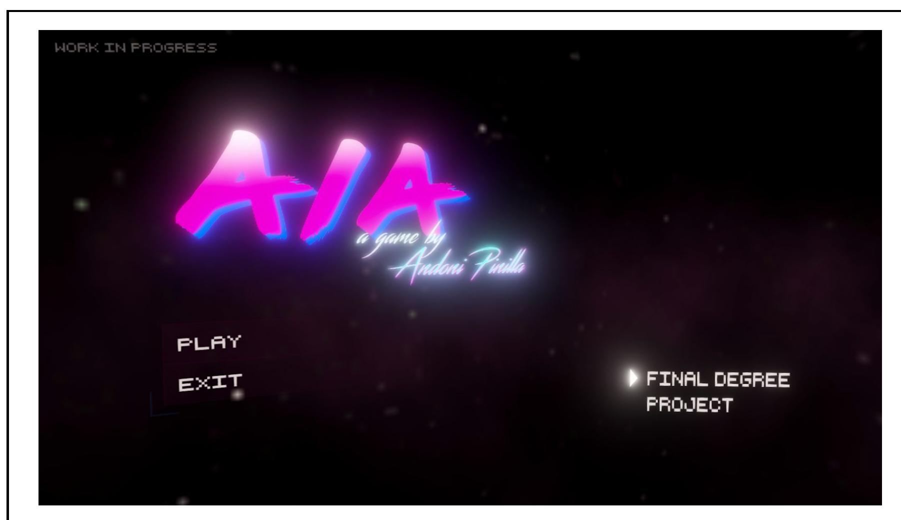


Figure 36: *Main menu.*

Finally, a controls interface (Figure 37) was added after the player chooses to play.

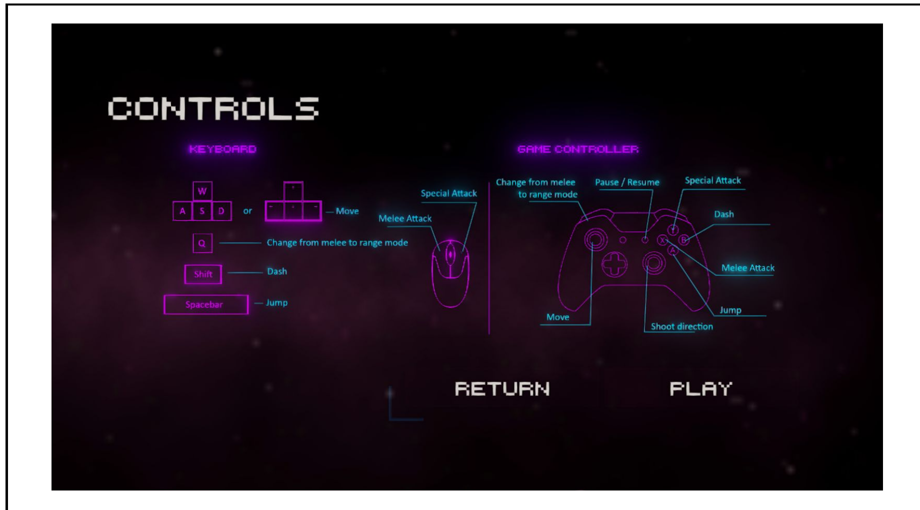


Figure 37: *Controls menu.*

3.3 Mechanics

The main mechanics of the game are running, melee attacking, area attacking, blocking, jumping, dashing and shooting. Animations are needed for all of these mechanics in addition to idle and death animations. The used animations are listed in the first element of Table 2. In order to control them and the transitions between them, the Animator Controller [52] component of Unity is used. It uses a State Machine (which was previously explained in this document), making it easy to manage animations and transitions. It can get messy when there are a lot of animations to be handled, though. In Aia, a lot of animations are need and, to solve this problem, sub state machines can be used. Figure 38 shows the State Machine of Aia's animations. Idle is the idle animation, but all of the other grey blocks are sub state machines that control different animations. For example, Attacks controls different kind of attacks, Movement controls movement in different directions, PistolGun controls shooting while standing still or while moving, etc.

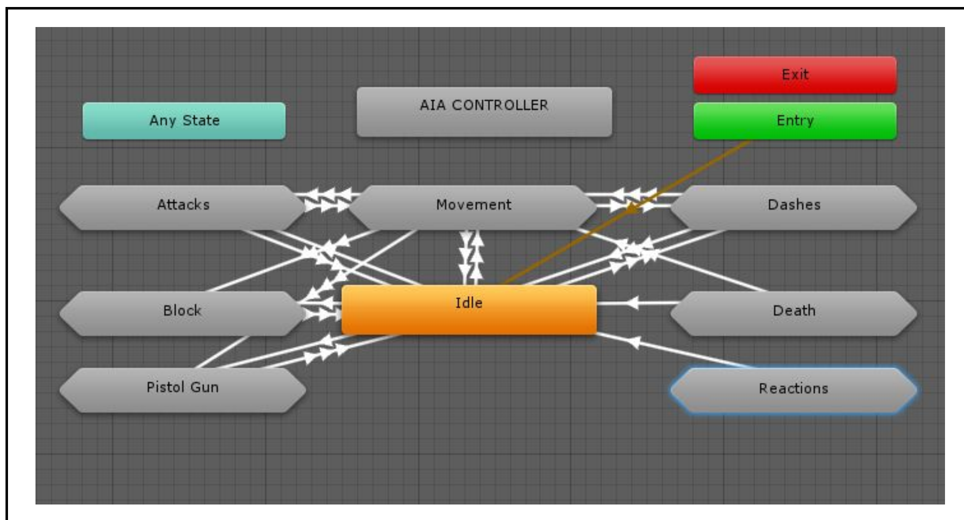


Figure 38: *Animator State Machine of the animations of Aia.*

The mechanics of the game also need to be controlled programmatically but there is no need to implement another state machine. The Animator Controller already provides easy ways to access the current state of the machine, to trigger some animations, etc. With the purpose of managing all the mechanics, the combat system and the resources, four classes have been implemented: *AiaController, AiaAttacking, AiaShooting* and *AiaHealth*.

AiaController is the main manager of all the mechanics. It registers the inputs of the keyboard and mouse or the gamepad and, depending on the state of the machine, triggers an animation, moves the player in space if needed (moving animations) and locks the input while performing an animation. It also handles all the sounds of the character.

AiaAttack is the class that manages the melee combat. When attack animations are triggered, this class activates and deactivates the sword colliders the needed amount of seconds to make a smooth feeling when attacking spiders. When a collision between a sword and a spider is produced, some collision particles are played and the spider health is reduced.

AiaShooting controls the shots when the player is shooting. For that, a raycast is casted from the gun into the forward direction of the player. The raycast is rendered in the game as a pink line that only lasts fractions of a second in screen, making a bullet effect. When the raycast reaches an object, collision particles are played and if the object is a spider, health from the spider is taken.

AiaHealth manages all the resources of the player: health, stamina, bullets and coins. If Aia is hitted by a spider, a function in this class is called to decreases Aia's health. Same thing happens when an action is made (stamina is taken), when shooting (bullets are reduced) or when taking a power-up. It also updates the user interfaces with the current state of the resources.

3.4 Game flow

Once the player has pressed the play button from the menu, the game starts. The *GameManager* class is in charge of managing all the game flow process. Firstly, it calls the *ProceduralGenerator* class to create the level, following all the steps explained in the procedural generation part of the project. Once the level is generated, the player is placed on start of the level. Then the player has to go through the different rooms of the level. Every room has a manager (*RoomManager*) in charge of managing things that happen in the room, such as spawning spiders or destroying rocks when the room is finished. When the player enters a new room, a new RoomManager is activated and the old one is deactivated. Finally, when the player gets to the end of the level, the GameManager will destroy the level and will create a new one with new rooms and new room managers. Depending on the number of the level, the difficulty will change. Higher levels have increased difficulty (more spiders, faster spiders, stronger spiders and so on). If the player dies, the "Game Over" menu is showed.

3.5 Artificial Intelligence

The artificial intelligence of the spiders is going to be implemented in two ways. First, with Finite State Machines and then with Machine Learning and Neural Networks. FSMs are implemented in order to be able to compare the Neural Networks with them.

3.5.1 State Machines

States Machines were explained in the design and investigation part of the project. They need some states and the transitions between those states to be defined. In this game, there are three kind of spiders, so three state machines are going to be implemented.

The main states that a melee spider can be are: attacking, chasing, wandering, escaping and dodging. When the spider is really far from the player, it just wanders moving to random positions. If the player gets close enough so the spider notices, the spider will start chasing the player. This range is far enough so all the spiders in the room the player is will chase him, but if the player has left behind some spiders in other rooms, they will not. If the spider gets into the attacking range, it will start to attack the player. If while attacking, the player attacks back, the spider the will dodge the attack as long as the dodge cooldown[2] is up. Also, the spider can escape for healing if its health points are low. This is only possible while chasing, because if the spider is attacking it means that it is next to the player and, as the player is faster than the spider, it is better to stay attacking than escaping and dying without taking health from the player. This behavior and and the state machine are visually represented in Figure 39.

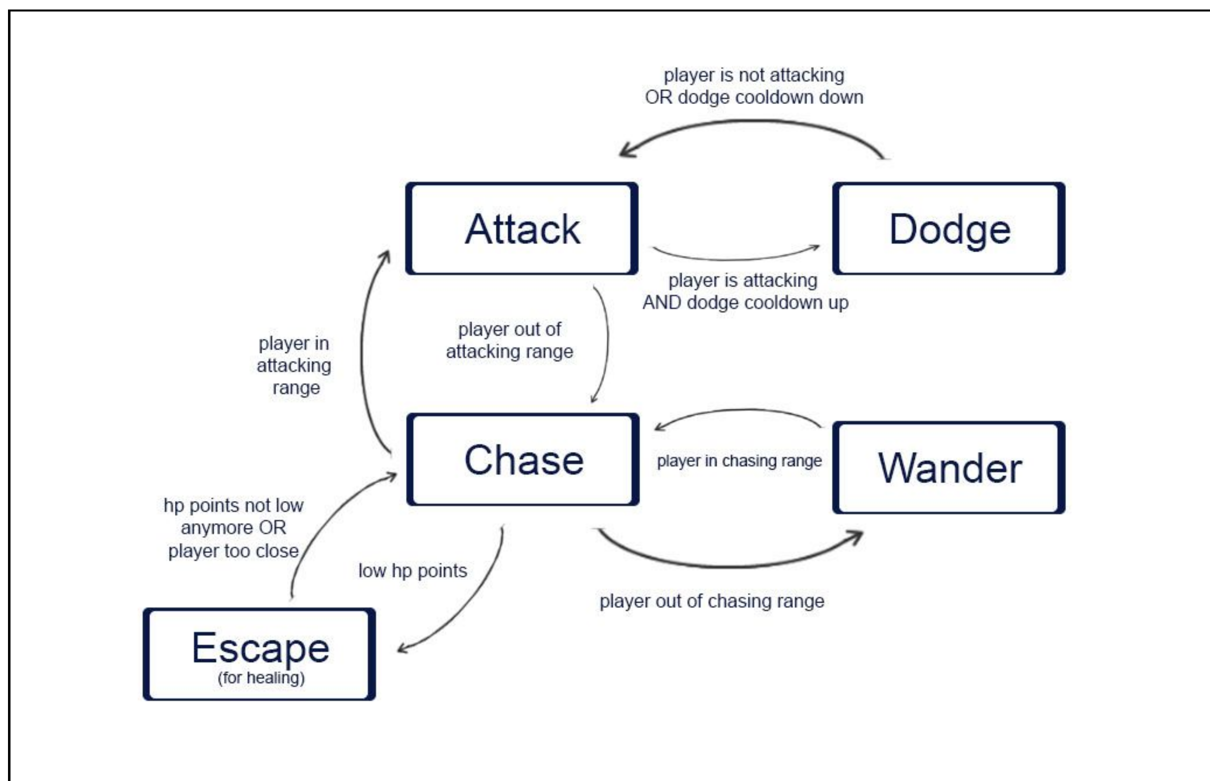


Figure 39: *Melee spiders state machine.*

[2] Cooldowns are time delays in which an ability cannot be used. For example, when dodging, the ability can't be used in the next 15 seconds.

The behavior and the state machine for the shooting spiders is almost the same. But they don't do melee attacks, they just shoot or spit to the player from distance. This way, the shooting range has to be bigger than the attacking range of the previous spiders. The shooting spiders state machine is in Figure 40.

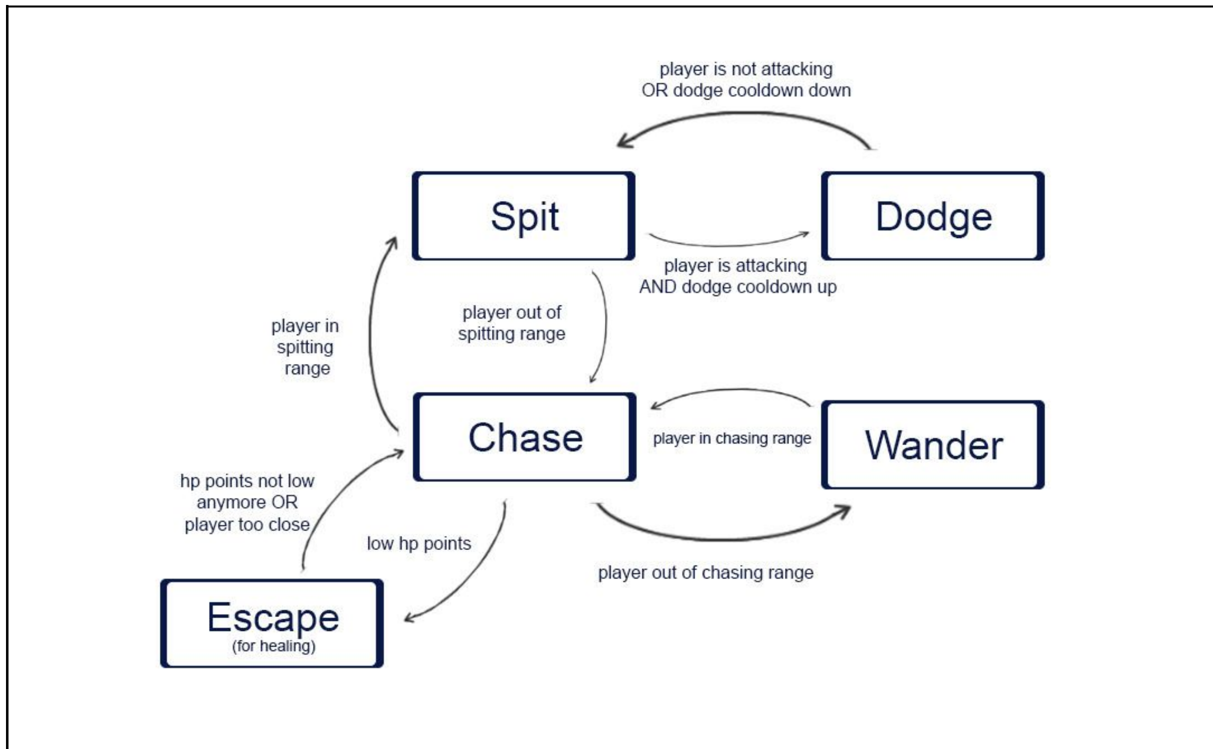


Figure 40: *Shooting spiders state machine.*

Lastly, the behavior and state machine of the exploding spiders is the simplest of all. They just chase the player until they are close enough to explode and make some damage. The only way for the player to not receive damage from them is to jump or to dash away in the exact same moment they explode (or just kill them with the shooting gun before they come too close). These spiders can't dodge or escape from the player because it would be unfair as they are already too annoying.

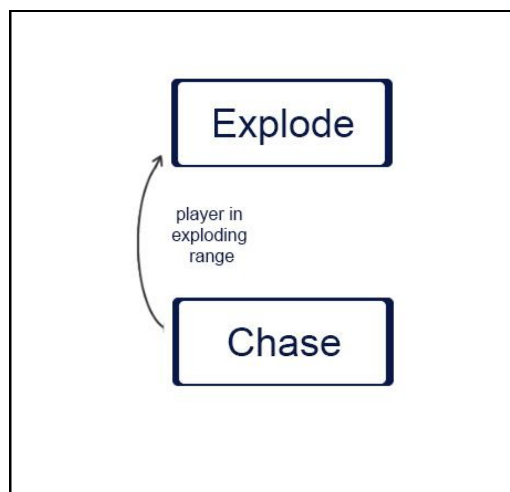


Figure 40: *Exploding spiders state machine.*

It is worth to remember that the movement of the spiders uses steering behaviors. They are already implemented in Unity's navmesh and by just calling the function navAgent.SetDestination() with the destination, the arrival, pursuit, collision avoidance and wall avoidance steering behaviors are performed. Also, the state machines are implemented in the game with simple switches. Figure 41 shows the implementation of the melee spiders state machine.

```
switch state:

       case state is attacking:

            if attack cooldown is up:
               attack()
            //Transition to chasing
            if distance to player is greater than the attack range:
               state = chasing

            //Transition to dodge
            else if player is attacking and dodge cooldown is up:
               state = dodging

            break

       case state is chasing:

            navmesh_destination = player_position //Chase

             //Transition to attack
            if distance to player is lower than attack range:
               state = attacking

            //Transition to escape
            else if health is lower than healthHP_threshold:
               state = escaping

            //Transition to wander
            else if distance to player is lower than the wandering range:
               state = wandering

            break

       case state is wandering:

            navmesh_destination = randomPoint() //Move to random point

            //Transition to chase
            if distance to player is lower than the wandering range:
               state = chasing

            break

       case state is escaping:
            navmesh_destination = pointAwayFromPlayer() //Move away
```

```
        //Trans to chase
        if distance to player is lower than attacking range OR health is
           higher than the highHP_treshold:
           state = chasing

        break

    case state is dodging:

        dodge()
        state = attacking //Transition to attack

        break
```

Figure 41: *State Machine implementation.*

3.5.2 Machine Learning and Neural Networks

Unity's ML-Agents use Reinforcement Learning to train Neural Networks. The agents perform some actions that change the environment and, depending on those changes, rome rewards and punishments will be applied. With those rewards, the neural network will change its abstract model of the environment in its hidden layers, making it more intelligent. The agents are going to be used in this game to make an intelligent AI and, hopefully, it will be more intelligent than the one implemented with state machines.

The ML-Agents learning environment is connected to an external Python API which is not included in Unity, so a whole process of connection had to be done. The learning environment is formed by three components [53]: agents, brains and an academy. The agents are the GameObjects or just characters that perform the actions and give the rewards and punishments. The brains encapsulate the logic behind the decision making of the agents. They also provide an easy way of setting the actions in the Unity's editor. Finally, the academy manages all the process including the observations and the decision making. A diagram of this is represented in Figure 42.

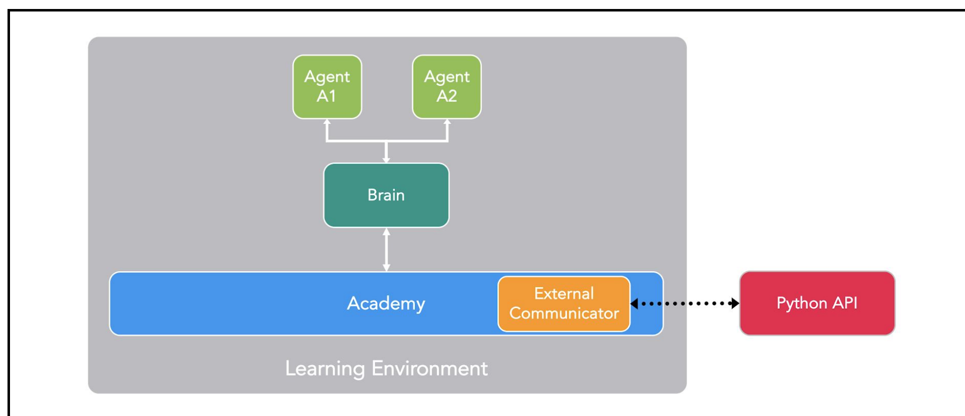


Figure 42: *Unity's ML-Agents diagram [48].*

3.5.2.1 Setting up the training

The agents of this game are the spiders. Only the melee spiders and the shooting spiders are going to be trained. There is no need to use neural networks for a spider that only chases the player (exploding spiders). Melee spiders will be assigned one brain and shooting spiders another brain so decisions don't interfere between them. As they learn with Reinforcement Learning, states, actions and rewards have to be declared first. Setting them up is a really hard job as any minor change can lead into completely different results. The whole training process of doing a lot of trial and error changes lasted more than a week to get it right.

In order to try to give the spiders more freedom at the time of learning intelligent behaviors, they'll have more actions than the state machines do. This is changing "chasing the player" and "escaping" with "turning right", "turning left" and "going forward". So the final list of actions would be:

1. Turning right
2. Turning left
3. Going forward
4. Left dodge
5. Right dodge
6. Staying still
7. Attacking (spitting for ranged spiders)

The inputs (environment variables) are:

1. The distance to the player
2. The player's local position
3. The spider's local position
4. The spider's health points
5. The player's health points
6. The player's stamina
7. The player's ammo left
8. A boolean telling if the player is attacking
9. A boolean telling if the player is shooting
10. A raycast that tells what is in front of the spider and at what distance

With this variables the neural network should be able to generate an abstract model of what is happening.

The rewards are probably the hardest ones to get right. As Cristian Opris [54] points out, a well known difficulty in reinforcement learning applications is correctly and accurately specifying a reward function such that it would lead to learning the desired behaviour. Poor or unspecified rewards can lead into behavior loopholes, difficult to anticipate by humans. The final rewards are:

1. Making damage to the player
2. Killing the player
3. Dodging an attack
4. Restoring health
5. Getting closer to the player (if the spider has more than a set amount of health points)

For the shooting spiders, this last reward changes to get into shooting range. And the punishments are:

1. Failing an attack (attacking to the air, for example
2. Taking damage from the player
3. Getting away from the player (if the spider has more than a set amount of health points).

But these rewards and punishments were not straight forward.

Figure 43 shows how the AgentAction function of the agent works. Depending on the input, some actions will be taken and the rewards explained above will be granted.

```
switch decision:

    case attacking:
         if attacking cooldown is up:

             attack();

             //If the attack hits the player, a reward of 1 is granted, but it's
             //controlled on a collision event. If the player is killed reward
             //is 5. It it does not hit the player reward is negative.

         break

    case going forward:

         navmesh_destination = forwardPoint() //Go forward
         distToPlayer = CalculateDistToPlayer()

         if distToPlayer  is lower than previousDistance:
             AddReward(0.005f)

         else if health is higher than lowHP_threshold:
             AddReward(-0.005f)

         //Restoring health rewards are granted in another function
         previousDistance = distToPlayer
         break

    case turn left:
         turnLeft()
         break

    case turn right:
         turnRight()
         break

    case dodge left:

         if dodge cooldown is up:

             dodgeLeft()
```

```
        if player is attacking AND player is close AND no damage received:
            AddReward(1f)

    break


case dodge right:
    if dodge cooldown is up:

        dodgeRight()

        if player is attacking AND player is close AND no damage received:
            AddReward(1f)

    break

 case staying still:

    navAgent.isStopped = true

    if distance to player is lower than chasing range:
        AddReward(-0.005f)

    break
```

Figure 43: *Basic code for the Action function of the agent.*

Now the environment has to be set in the Unity's scene. Justesen, N., Rodríguez, R., Bontrager, Kalifa, A., Togelius, J. and Risi, S. [55] point out that when networks are trained using RL in a fixed environment, like a level of a videogame, it is probable for it to overfit and get used to that specific level, having difficulties to generalize to new levels. To fix this problem, for this game, different rooms with different sizes and different obstacles will be prepared. Also in every room, a different number of spiders will be placed. This is all made so they don't get used to a specific room sized or to play by themselves. This is more solid than repeating the same thing all the time. But the before cited authors also say that the best way to fix this would be generating a procedural level every time. For this game this is possible, as levels are procedurally generated, but having to connect the procedural system with the ML-Agents environment and how they work would be a time consuming task. And having to generate new levels every time could slow down the training process significantly for the available time. Nonetheless, this will be considered in the future to improve the results.

Now, the player that spiders will play against is a simple heuristic that attacks the them, moves randomly in the map and shoots from time to time. The player being an heuristic means that it is a practical method to make, but it is not always the best choice. Making an intelligent player representing real life players would be as difficult as this entire thing alone, and playing with real players would be very time consuming, so the heuristic approach is finally taken. The final set environment is in Figure 44.

Figure 44: *Training environment.*

Lastly, the Academy component is set so the training can be done 100 times faster. But first it is worth to try it with normal speed to check that everything works correctly. Also double checking the code and that the rewards are coherent is a must. That's because a build of the training scene has to be made to be able to train it within the python environment, so building a malfunctioning scene can be time consuming and frustrating (and even when aware of that it still happens). Once the training is completed, a model is generated and can be imported into the game and the brains.

3.5.2.2 Training

The initial trials had a lot of flaws. At first, the spiders were rewarded when getting closer to the player but not punished when getting farther. This way, they came up with a behavior to exploit this reward by going back and forward repeatedly. They didn't even try to attack the player. To solve this, the getting away from the player punishment was added. That's only one of the many exploits they found. Any little change in the rewards and they could learn a completely different behavior, so there were a lot of changes in the reward system.

Another problem was that, originally, the raycast wasn't used for the environment variables. The training did go as expected, but in the real game they weren't able to avoid wall collision. To solve this, the raycast was added in order to know what the spider has in front and at what distance.

Approximately 750.000 steps of the training were executed for each type of spiders, stopping every 50.000 steps to check that it was working correctly and with intervals of 1000 steps to calculate the mean reward. The accomplished behavior was kind of what it was expected, but it also had some unexpected behaviors. It is similar to the state machines, but with some details that make them stand out.

The melee spiders get close to the player and attack him in both cases (FSM and NN). The state machines attack when they are in range of attack so the attack hits the player. But the neural networks can attack whenever they want (even to the thin air) and what they do is to take advantage of the velocity that they carry, attacking even when they are not in range of

attacking because thanks to the velocity, the attack will hit the player. This behavior wasn't expected at all and, despite not being difficult to implement with state machines, probably nobody would have thought of implementing it. Also, they have learned to dodge attacks. But they also do another smart use of the dodge skill. If the player is low and trying to escape, they will use the movement in order to get closer to the player without having to turn and move. This was the most surprising behavior of all as how smart it is. The only thing that wasn't accomplished was escaping for healing. They just prefer to attack the player having the possibility to kill her (killing Aia has a big reward). And that's not really a problem as it represents the goal of the spiders: killing. Finally, the movement looks a little bit weird as sometimes it stops and keeps going forward. It is so fast that it is almost not noticeable but it gives this weird movement style. Someone could think this as a problem but, surprisingly, it resembles more the way spiders would move than the previous one.

Figure 45 shows the reward over time during the training. In the first 150.000 the spider is still learning to chase the player. After that, until 250.000-300.000 the spider learns how to attack properly to get the most points out of it. At 450.000 the behavior is almost following the explained pattern, but it is left training until 750.000 to perfectionate it.

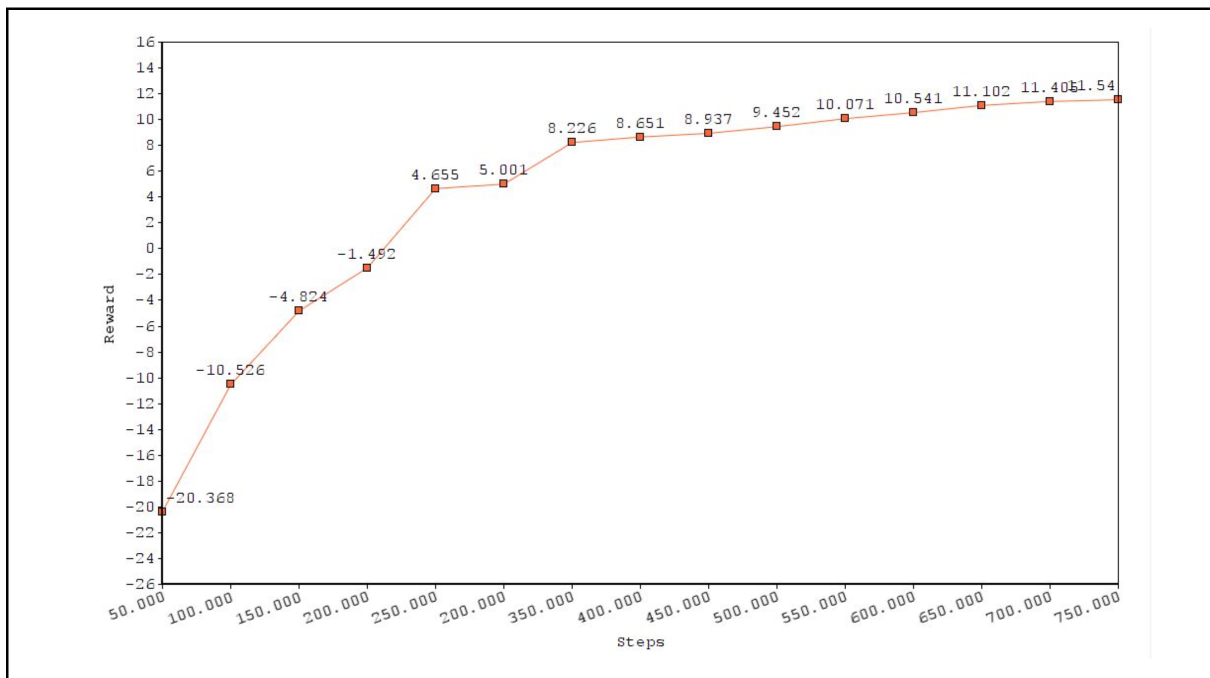


Figure 45: *Melee spider NN mean reward over time (steps).*

The trained shooting spiders behavior is similar to the state machines one. But it seems like sometimes, when shooting, they try to predict where the player is going to be (not always directly shoot to the player's position). They also use the dodging movement for something more than dodging the player attacks. They use it to get to a place where it is easier to turn and shoot the player. These spiders haven't learned to get away from the player. Mainly because they shoot from distance, so getting away isn't really intelligent when player is far and neither when the player is really close (the player is faster).

These spiders, at approximately 100.000 steps have already learned to position themselves in order to shoot the player. The reward is still negative because as the player moves, it is difficult to hit at first. After that it starts to shoot better little by little. At 300.000 steps the spider shoots properly, dodges and uses the dodge to be positioned better. The

remaining steps the spider perfectionates the technique. This is all visually represented in Figure 46.

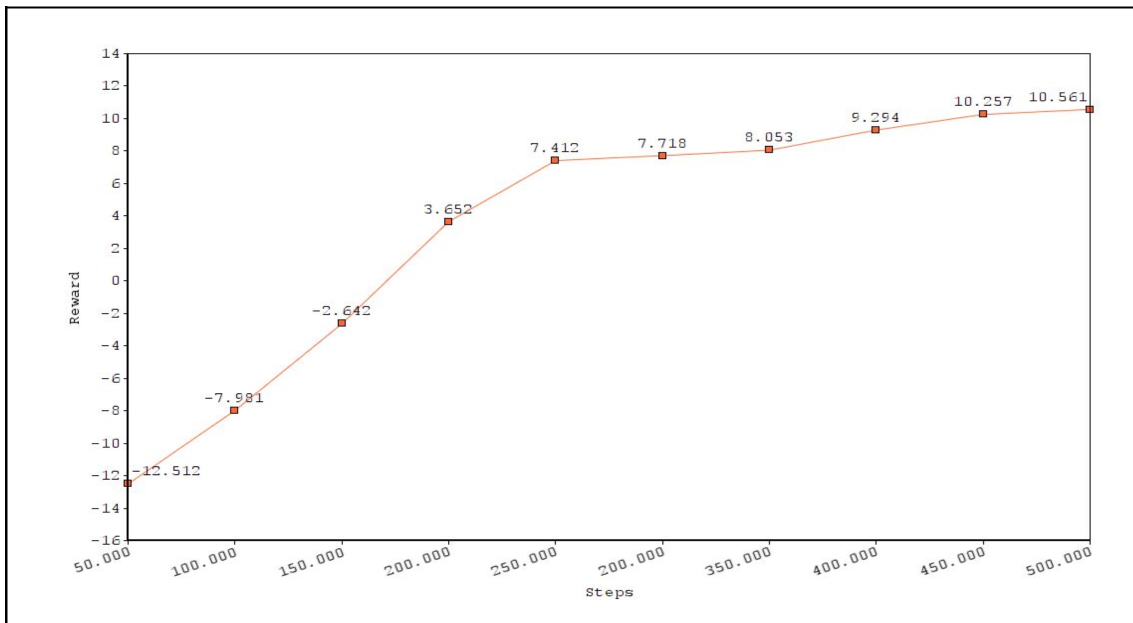


Figure 46: *Range spider NN mean reward over time (steps).*

Other trials were made with other actions for both spiders. Because after thinking about it, it didn't seem that the given actions to try to give more freedom to the spider (turning right, turning left and going forward) had paid off with some smart and different conducts, instead of using the "chasing the player" and "getting away from the player" actions which are more straightforward. To check if this was the case, some trials were made with last two actions. The achieved behavior was practically the same. Even the unexpected behaviors that the others had, were still the same. What is more, they only needed half of the steps to achieve the same behavior (partly because they didn't need to learn how to turn to the player). So implementing the more freedom actions ended up being more of a waste of time than really a more intelligent AI, and also the problem with the raycast and the wall avoidance wouldn't have even occurred. But it was worth trying for the sake of the investigation.

3.5.2.3 Results

To graphically represent the results, and compare NN spiders with FSM ones, a NN spider and a FSM spider have been set to play versus the heuristic player. Each one have played 50 games and a mean of the damage done and received has been calculated. Maybe playing versus an heuristic is not the perfect way of comparing them, but because of the lack of resources (not being able to test with a lot of players as a big company could in a testing process) this has to be done.

Figure 47 shows that trained spiders do a little more damage over time than the state machine ones. This may be because of the attacking with velocity exploit. Also, as it can be seen in Figure 48, they receive a lot of damage (almost the same) because the player is stronger. This also falls within the nature of the spiders of attacking even if they get hurt.

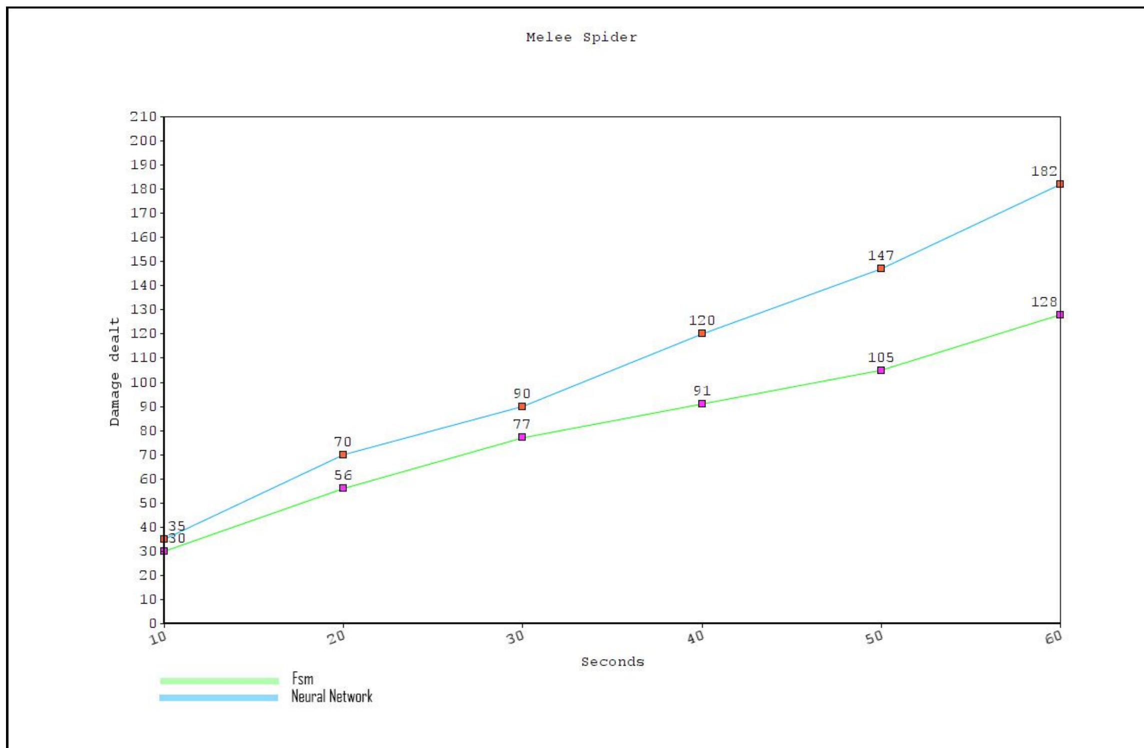


Figure 47: *Melee spiders mean damage dealt over time.*

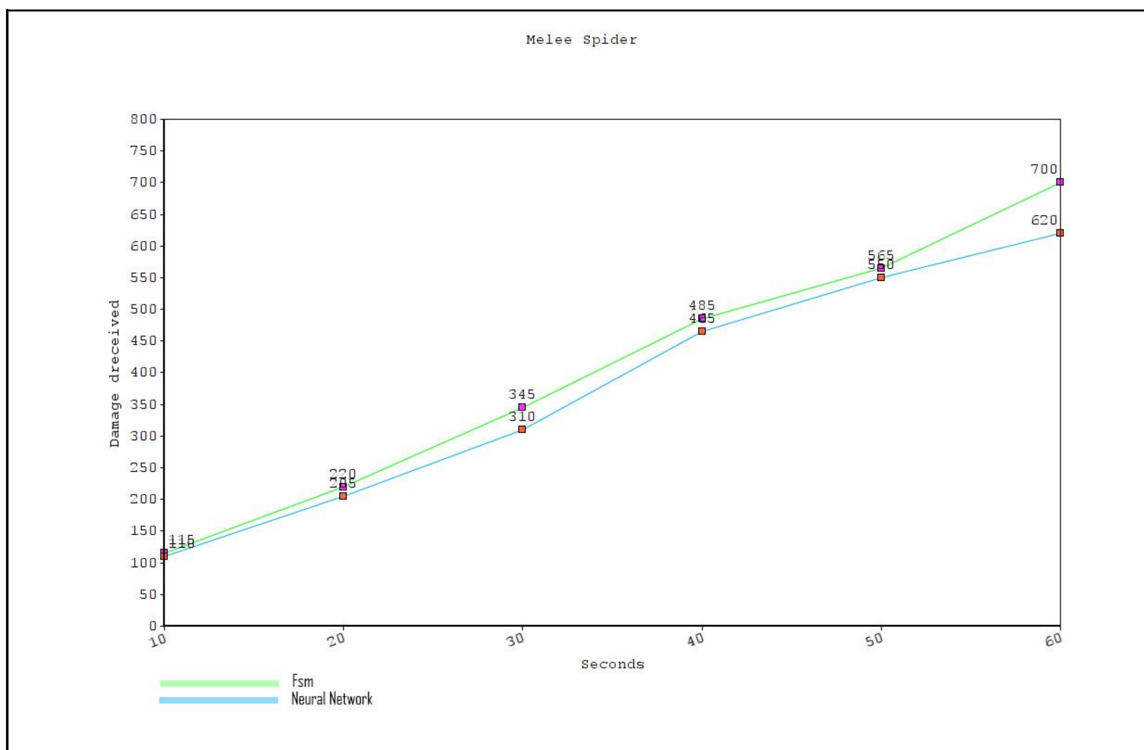


Figure 48: *Melee spiders mean damage received over time.*

Shooting spiders deal less damage than melee ones (Figure 49), because it is more difficult to hit a spit than to make a close attack. Also, the trained shooting spider kills the player almost fifteen seconds faster than the state machine one. Lastly, state machine spiders

receive more damage than the others (Figure 50). Also, this damage received is a lot less than the melee spiders as they are usually far from the player.

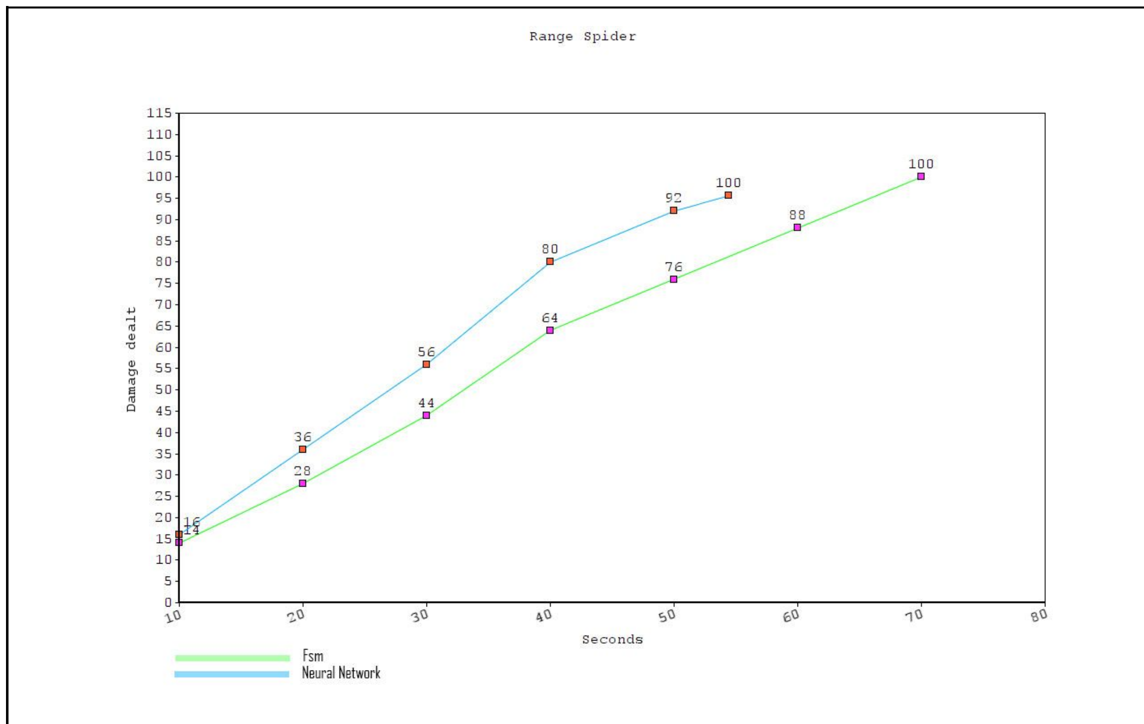


Figure 49: *Range spiders mean damage dealt over time.*

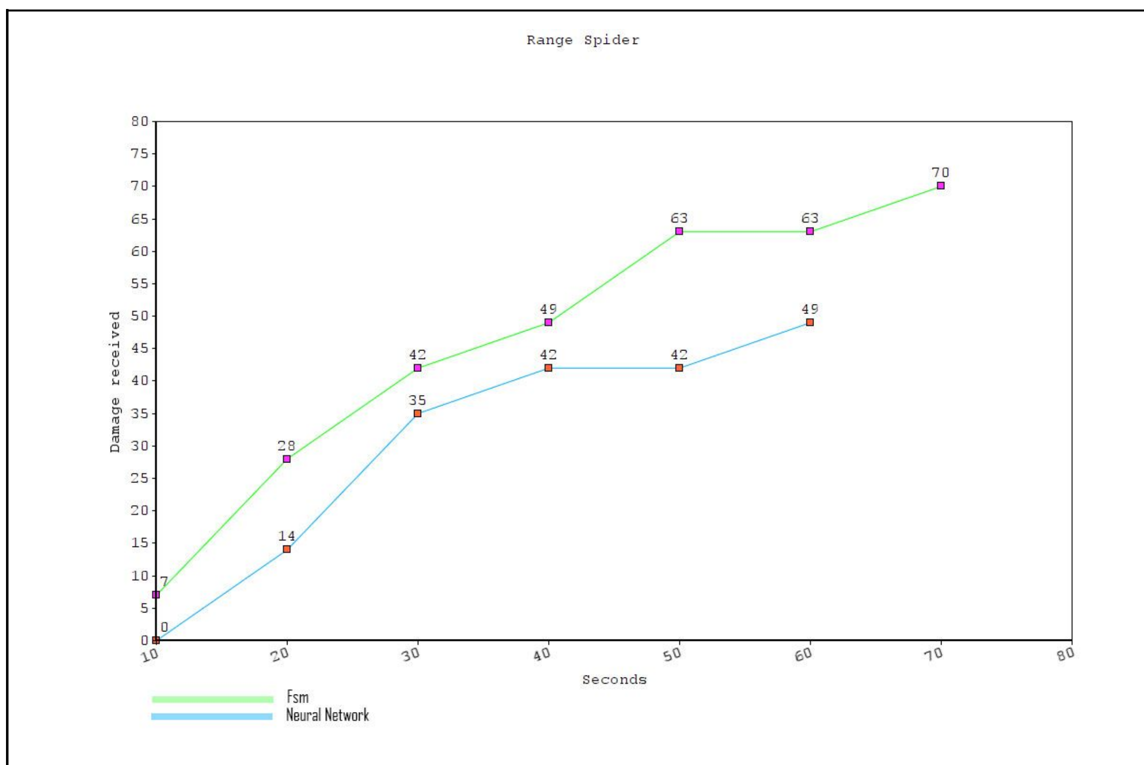


Figure 50: *Range spiders mean damage received over time.*

With all of this data in mind it is obvious that an improvement has been achieved with neural networks. Furthermore, some real life players that have tried the game state that it is a lot more difficult to play versus neural networks than versus state machines. The result of this

attempt to apply Machine learning and Neural Networks to the artificial intelligence of a game has been quite good. It is clear that they are techniques that can be implemented in video games and, despite requiring a lot of work to get them to work properly, can lead to amazing results. They provide a new way to think about the AI of a game, as they usually find exploits and new ways to beat the enemy. It is likely that these mechanics are much easier to implement in a state machine, but probably would not have occurred to anyone if it had not been for the training. They also provide more unexpecting behaviors as they don't always follow the same pattern. Maybe Machine Learning is easier to apply in other kinds of games (like strategy games, for example) or in games with the necessary resources to make a longer and more detailed trainings. Machine Learning could also be applied for testing purposes and preventing exploits both from the players and the artificial intelligence.

4 Final results

After the end of the development of this project, a final result has been achieved. The final procedural generation system, created with the techniques explained before, is very robust as it can create different levels every time, also being able to easily edit some variables in order to make bigger/smaller levels with bigger/smaller rooms. Two kinds of artificial intelligence have been developed too. One of them with Finite State Machines and the other one with Neural Networks. It was expected that the neural network AI would be smarter and would have better results, and so it was. A really well-made and thought, difficult to beat AI with unexpected behaviors, has been created, making the game fun to play. What also makes players enjoy the game are the smooth the mechanics and the combat system that were implemented.

The art was not the main purpose of the game, but the final look of it is far better than it was expected when developing the technical proposal. The chosen style (retro but simple), the assets and the shaders to unify them really helped to achieve that look. Figure 51, Figure 52, Figure 53, Figure 54 and Figure 55 show final screenshots of how the game looks.

The game repository of the code is at this Repository. A total of 25 classes with an approximately total of 6000 lines of code have been implemented.

Also, a generated build to play the game can be found here. That build has lower difficulty at first (spiders do less damage and are slower) so everything can be tried without dying all the time.

Lastly, a gameplay of the game can be watched in this YouTube Video.

Figure 51: *Final screenshot 1.*

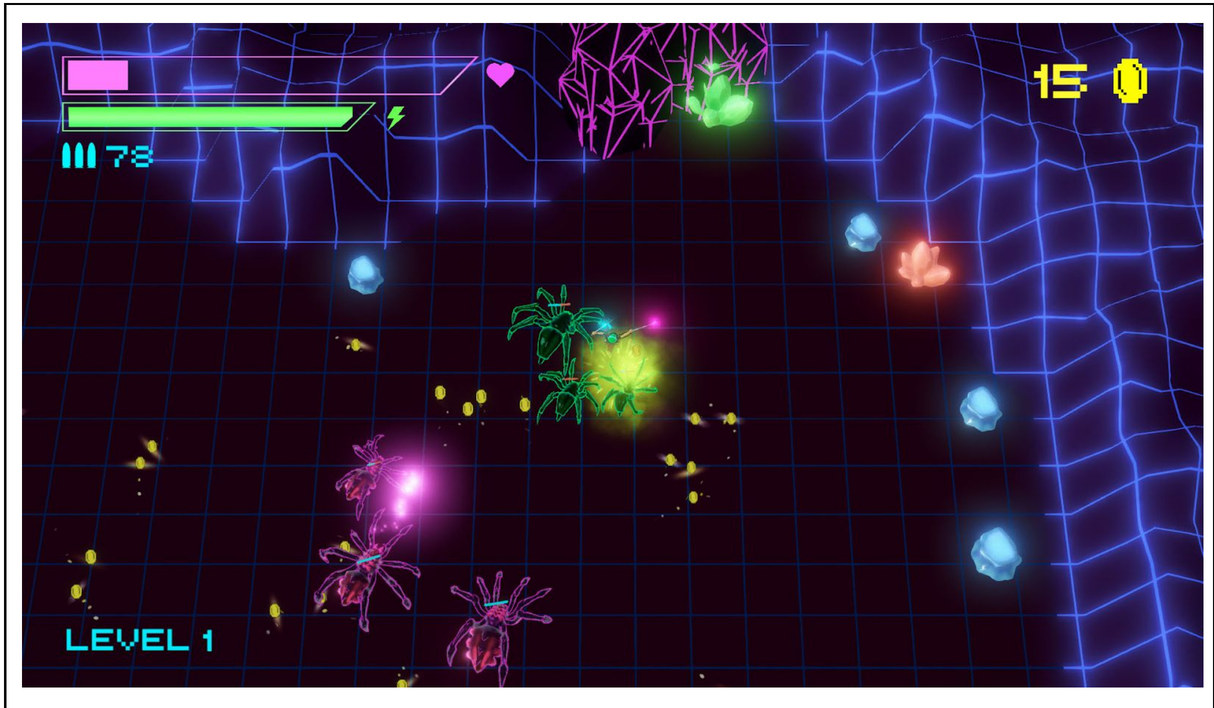


Figure 52: *Final screenshot 2.*

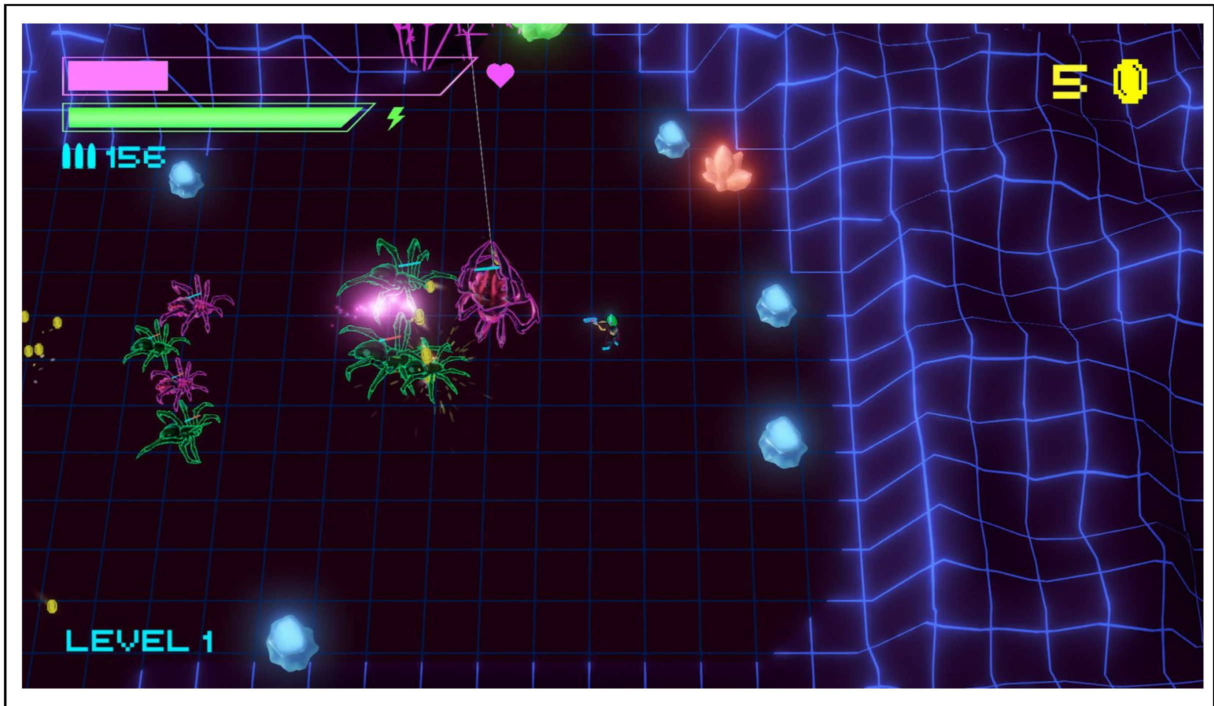


Figure 53: *Final screenshot 3.*

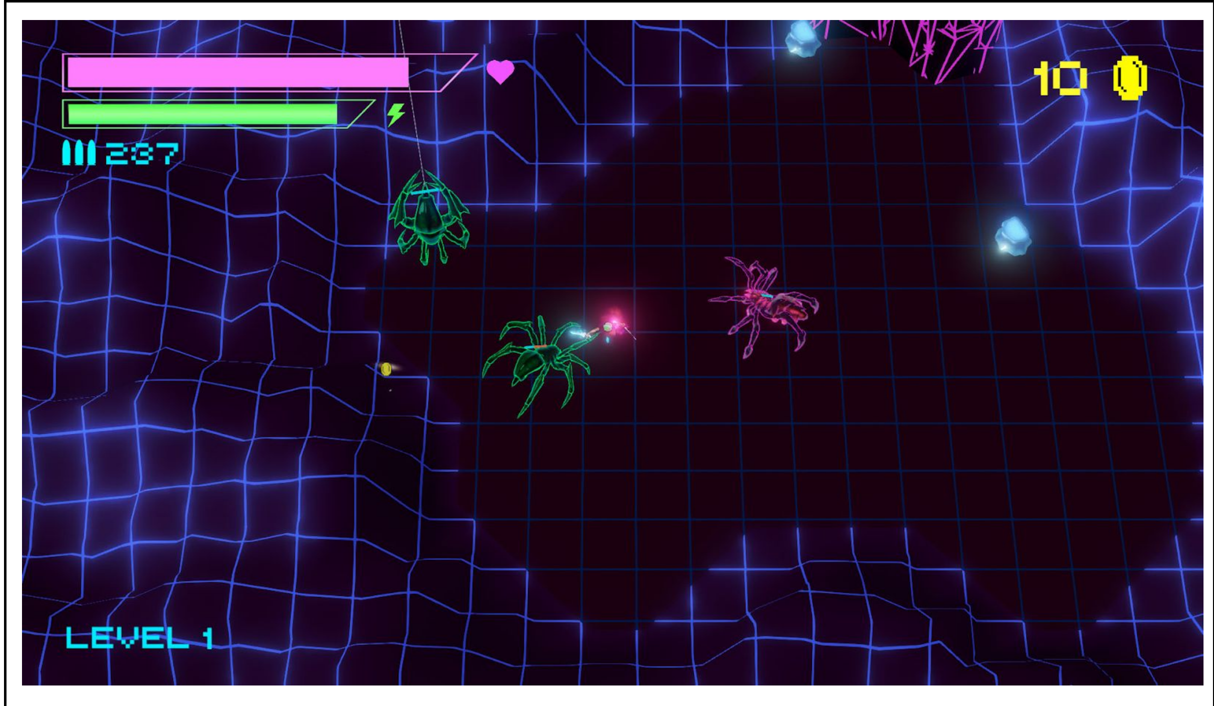


Figure 54: *Final screenshot 4.*

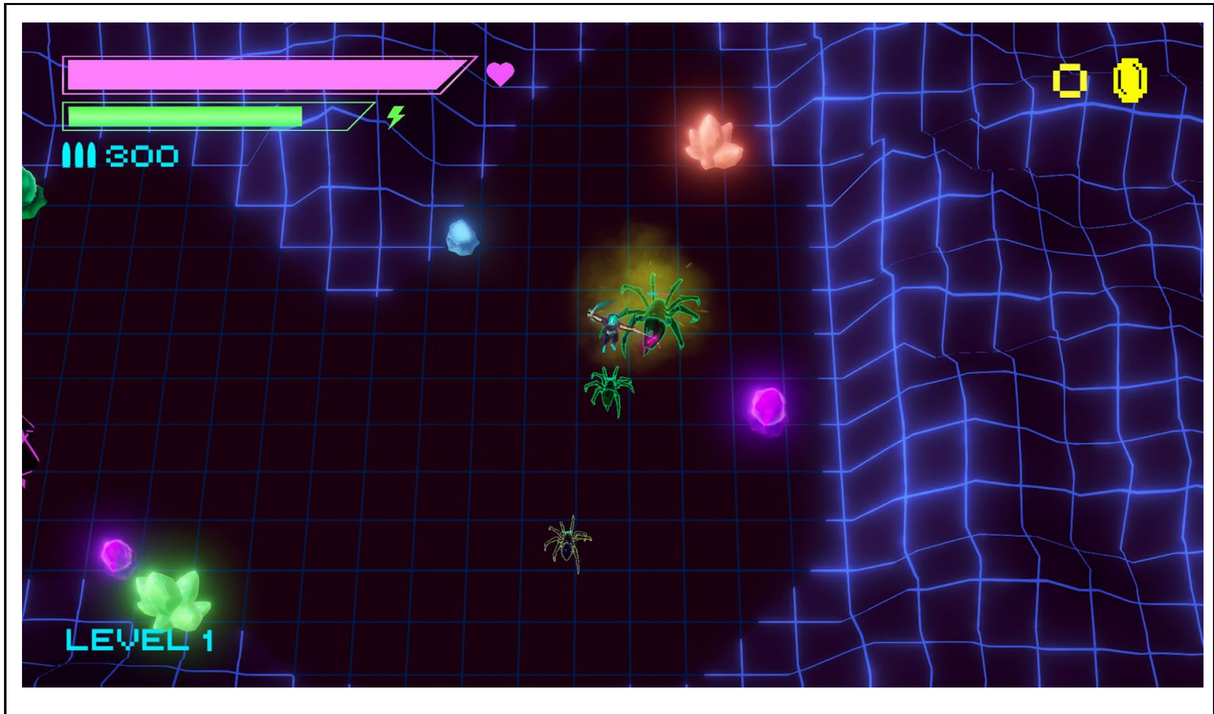


Figure 55: *Final screenshot 5.*

5 Conclusions and future plans

All the game expectations have been met. In fact, it could be said that they have been exceeded as the finished look of the game is far better than expected and the neural networks have ended working properly (there was a big possibility that they didn't with the time available and the beta state of the ML-Agents plugin). Furthermore, the procedural generation system generates organic and robust caves without any major flaw.

In addition, in the investigation part of the project a lot has been learned about the different topics of the project: different ways of creating procedural worlds and levels, different algorithms and techniques, how to create a robust combat system and a lot about Machine Learning and Neural Networks and why they are so popular at the moment.

At the beginning of the project it was said that only a small part of the resources of the majority of games is used for artificial intelligence, and that Machine Learning and Neural Networks were not popular in the video game industry because of the misconception that they tend to be computationally expensive. But computers are more powerful than ever and new algorithms are being developed (like PPO) that are not expensive at all. Machine Learning has been used by every big company during the last years and it was a matter of time that it would start to be used in the video game industry. In fact during the last E3 conferences, Microsoft announced that they were using Machine Learning to decrease loading times in Xbox One games [56]. And with the addition of the ML-Agents plugin to Unity, it is going to rapidly increase the use of it in games as it is a very popular game engine, even if they are still in beta (it has been proved in this project that they can be applied and that they work).

Lastly, it is fair to point out that Machine Learning can't only be applied to enemies or artificial intelligence for NPCs. It can potentially be used for any imaginable thing if you are creative enough. For example, the main characteristics of this project could be combined as Procedural Content Generation via Machine Learning (PCGML) [57]. At the investigation part of the procedural generation it was mentioned that Brian Walker said that procedurally created levels will never come close to hand-crafted levels designed by experts. With PCGML, new levels could potentially be generated with Machine Learning models trained on existent levels created by experts.

Future plans have been thought for the game. Firstly, game content will be added. For example, new items, different colors and skins for the different procedural levels, spider final bosses for the final rooms (and its corresponding neural network training), etc. The plan is to upload it for free in sites like itch.io and to keep investigating about Machine Learning and Neural Networks by trying new things with Unity's ML-Agents.

6 References and bibliography

[1] Nigretti, A. “Using Machine Learning Agents in a real game: a beginner’s guide” (11/12/2017) https://blogs.unity3d.com/es/2017/12/11/using-machine-learning-agents-in-a-real-game-a-beginners-guide/

[2] Juliani, A. “Introducing: Unity Machine Learning Agents” (19/09/2017) https://blogs.unity3d.com/es/2017/09/19/introducing-unity-machine-learning-agents/

[3] Charles, D. McGlinchey, S. “The past, present and future of artificial neural networks in digital games” (17/06/2018), page 6

[4] Sutton .S, .R “Reinforcement Learning: An introduction”

[5] “Rogue” https://es.wikipedia.org/wiki/Rogue

[6] “The Binding of Isaac” https://es.wikipedia.org/wiki/The_Binding_of_Isaac

[7] “Nuclear Throne” https://es.wikipedia.org/wiki/Nuclear_Throne

[8] “The Binding of Isaac: Rebirth Screenshots (6)” http://www.pushsquare.com/games/ps4/binding_of_isaac_rebirth/screenshots

[9] “Berlin Interpretation” http://www.roguebasin.com/index.php?title=Berlin_Interpretation

[10] Dotson, C. “The Beginner's Guide to Roguelikes” (06/12/2016) https://www.lifewire.com/what-are-roguelikes-4117411

[11] “Rime” https://en.wikipedia.org/wiki/Rime_(video_game)

[12] “Inside” https://en.wikipedia.org/wiki/Inside_(video_game)

[13] “Rime” http://gamechanger.co.ke/product/rime/

[14] Rao, V. “Playdead's Inside Ending Walkthrough Part 6 - The Escape, Control the Blob Creature” (04/07/2016) https://www.gamepur.com/guide/23532-playdeads-inside-ending-walkthrough-part-6-escape-control-blob-creature.html

[15] “Far Cry Blood Dragon” https://www.ubisoft.com/es-es/game/far-cry-3-blood-dragon/

[16] “Kira” https://assetstore.unity.com/packages/3d/characters/humanoids/kira-stylized-character-100303

[17] “Shadowrun Hong Kong” https://shadowrun.gamepedia.com/Official_Shadowrun_Wiki

[18] Wainer, G. Liu, Q. Dalle, O. Zeigler, P. (13/08/2010) “Applying Cellular Automata and DEVS Methodologies to Digital Games: A Survey” , pages 6-7

[19] “No Man’s Sky” https://es.wikipedia.org/wiki/No_Man%27s_Sky

[20] Smith, G. “Interview: How Does Level Generation Work In Brogue?” (28/07/2015) https://www.rockpapershotgun.com/2015/07/28/how-do-roguelikes-generate-levels/

[21] Adonaac, A. “Procedural Dungeon Generation Algorithm” Gamasutra. (09/03/15) https://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php

[22] Lague, S. “Procedural Cave Generation tutorial” https://unity3d.com/es/learn/tutorials/s/procedural-cave-generation-tutorial

[23] Adonaac, A. “Procedural Dungeon Generation Algorithm” Gamasutra. (09/03/15) https://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php, Hallways image

[24] Ivanov, D. “Procedural cave generation for Unity3D” https://github.com/nzhul/procedural-cave-generation

[25] Millington, I. Funge, J. “Artificial Intelligence for Games” (2009), page 4

[26] Millington, I. Funge, J. “Artificial Intelligence for Games” (2009), page 8

[27] “NavMesh” https://docs.unity3d.com/ScriptReference/AI.NavMesh.html

[28] Bevilacqua, F. “Finite-State Machines: Theory and Implementation” https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

[29] Simpson, C. “Behavior trees for AI: How they work” Gamasutra. (07/17/14) https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

[30] “Machine Learning” https://en.wikipedia.org/wiki/Machine_learning

[31] “Creatures” https://en.wikipedia.org/wiki/Creatures_(video_game_series)

[32] Charles, D. McGlinchey, S. “The past, present and future of artificial neural networks in digital games” (17/06/2018), page 7

[33] “Training with Proximal Policy Optimization” https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md

[34] “Proximal Policy Optimization” https://blog.openai.com/openai-baselines-ppo/

[35] Lague, S. “Procedural Cave Generation tutorial” https://unity3d.com/es/learn/tutorials/s/procedural-cave-generation-tutorial

[36] “Cellular automaton” https://en.wikipedia.org/wiki/Cellular_automaton

[37] “Marching Squares” https://en.wikipedia.org/wiki/Marching_squares

[38] “Flood Fill” https://en.wikipedia.org/wiki/Flood_fill

[39] Biagioli, A. “Understanding Perlin Noise” http://flafla2.github.io/2014/08/09/perlinnoise.html

[40] Kun, J. “The Cellular Automaton Method for Cave Generation” (29/07/2012) https://jeremykun.com/2012/07/29/the-cellular-automaton-method-for-cave-generation/

[41] “Marching Squares” https://en.wikipedia.org/wiki/Marching_squares

[42] “The marching squares algorithm” http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html

[43] “Flood-fill algorithm” https://www.hackerearth.com/practice/algorithms/graphs/flood-fill-algorithm/tutorial/

[44] Lague, S. “Passageways” https://unity3d.com/es/learn/tutorials/projects/procedural-cave-generation-tutorial/passageways?playlist=17153

[45] “Breadth First Search” https://en.wikipedia.org/wiki/Breadth-first_search

[46] “Unity docs: Perlin Noise” https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html

[47] “Unity docs: NavMeshSurface” https://docs.unity3d.com/Manual/class-NavMeshSurface.html

[48] “Unity Asset Store” https://assetstore.unity.com/

[49] “Unity docs: PostProcessing Stack” https://docs.unity3d.com/Manual/PostProcessing-Stack.html

[50] Hodgson, J. “LAZER 84”
https://www.behance.net/gallery/31261857/LAZER-84-Free-FontInferno

[51] Szabó-Lencz, P. “Retro Computer” https://www.dafont.com/es/retro-computer.font

[52]“Unity docs: Animator Controller”
https://docs.unity3d.com/Manual/class-AnimatorController.html

[53] “ML-Agents Overview”
https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md

[54] Opris, C. “Reinforcing safety, with style: exploring reward shaping through human feedback“ (2017) http://cs229.stanford.edu/proj2017/final-reports/5220081.pdf

[55] Justesen, N., Rodríguez, R., Bontrager, Kalifa, A., Togelius, J. and Risi, S. “Procedural Level Generation Improves Generality of Deep Reinforcement Learning“ (28/06/2018) https://arxiv.org/abs/1806.10729

[56] Daws, R. “MICROSOFTE3 2018: Xbox FastStart uses machine learning to get into content faster” (11/06/2018)
https://www.artificialintelligence-news.com/2018/06/11/e3-2018-xbox-faststart-machine-learning/

[57] Summerville, A., Snodgrass, S., Guzdial, M., Holmgard, C., Hoover, A. K., Isaksen, A., Nealen, A., Togelius, J. “Procedural Content Generation via Machine Learning (PCGML)” (02/02/2017) https://arxiv.org/abs/1702.00539