Neural networks applied to a tower defense video game

Final Degree Project

By

ADRIÁN GONZÁLEZ RAMÍREZ

Videogames design and development Degree
JAUME I UNIVERSITY

MENTOR: RAÚL MONTOLIU COLÁS

3RD JUNE 2018

Table of Contents

List of Tables

List of Figures

Abstract

This project has been created by Adrián González Ramírez as his Final Degree Project of the *Degree in Design and Development of Video Games* of the *Jaume I University* [1].

This project focuses on creating a tower defense video game [2] and incorporating different models graphs (Neural Networks [3]) generated with *Machine Learning Agents* [4] and the *Proximal Policy Optimization (PPO)* [5] *Reinforcement Learning* [6] algorithm, which will determine the behavior of the *Non Playable Characters (NPC)*.

Machine Learning [7] is, in the decade of 2010, a very present technique in a large number of areas. The basis of the machine learning studied in this project are the same for any project that uses Reinforcement Learning [6].

The method presented in this project shows a way to get different difficulty levels without hardcoded behaviors, as Rubber banding [8], making less evident the manipulation of the difficulty in the attempt to keep users desire to keep playing.

CHAPTER 1

TECHNICAL PROPOSAL

The present section composes the technical proposal of the Final Degree Project that *Adrián González Ramírez* [9] will develop in the *Degree in Design and Development of Video Games* at the *Jaume I University* [1].

The proposed Final Degree Project consists in the development of a video game of the Tower Defense genre in the Unity3D [10] engine that incorporates a Non-Playable Character (NPC) that uses Machine Learning [7] techniques to simulate the behavior of a human player and adapt it to the different situations of inside a game. This NPC will improve its skills based on previous game experiences by using neural networks.

Keywords: "tower defense", "artificial intelligence", "reinforcement learning", "machine learning", “neural networks”.

1.1 Project motivation

Nowadays there are countless video games in which the player plays against a player not handled by a human being, i.e. against a machine. Many of these games have an artificial intelligence based on rules that often become predictable. A predictable AI can provoke in the player a loss of desire to continue playing since he can learn certain techniques the machine don’t know how to react to or, just the opposite, the machine takes control shamelessly of what happens in the game. Neural networks are a nice option to create a decent AI.

Rubber banding or *Dynamic game difficulty balancing* [8] is a technique that tries to maintain games equalized by adapting the difficulty to the player’s level. The *rubber banding* AIs are quite

maddening for players with great skills that look for fair games. One example of *rubber banding* is the *US7278913B2* [11] used in *Mario Kart: Double Dash!!* [12] and similar behaviors are still visible in latest video games. An explanation of the behavior of some games of this king can be seen in *Paste Magazine* [13].

Although companies like *Electronic Arts* deny the existence of this kind of AI (read the words of *Matt Prior* in *Eurogamer* [14], creative director of *FIFA 18* [15]), a lot of players are still uncomfortable with certain advantages that the AI takes at certain moments. One example is to see an NPC car in *Need for Speed* [16] suddenly accelerating and slowing down whenever it gets too far from the player. Another example is to see goalkeepers in *FIFA 18* making fantastic saves or incredible mistakes depending on the match status.

Using neural networks properly would allow getting AIs adapted to player's level avoiding the weird results that other widely used techniques currently show. Keeping the same level of difficulty during all the match in FIFA would also keep the players desire to play without making the management of the difficulty so obvious, with specific actions, as now it does.

1.2 The game

The video game *TAJ* will be a tower defense and it will have totems as main characters. The game area will be divided into several straight paths that connect the base of one player with that of the other. The main goal of the player is to defend, with three types of totems and four types of power-ups, his 3 power stones and to attack the enemy's ones. The winner will be the player who conquers the 3 enemy's power stones.

1.3 Related subjects

This project is related to the following subjects:

- *VJ1231 Artificial intelligence*
- *VJ1234 Advanced Interaction Techniques*
- *VJ1227 Game Engines*
- *VJ1226 Character Design and Animation*

1.4 Objectives

The main objective is to develop an artificial intelligence for a video game by using neural networks. This neural network must detect the best action given a game status. This is how the NPC will be able to adapt its actions to the changing situations during a game. It will have to

learn from its experience in previous games against another random NPC and some tests against human players.

1.5 Target

TAJ is aimed at casual players of all ages looking for fun in short periods of time such as commuting to work or college. These casual players focus much more on short-term competitiveness and entertaining mechanics than on the story and narrative of the game.

1.6 Expected Results

The main objective of the work is to implement neural networks algorithms that result in an NPC able to simulate the actions that a human player would perform. The ideal NPC would allow an addictive, challenging and often unpredictable gameplay, avoiding the predictable behavior of the predefined rules based AI's. It is important to obtain a pleasant audiovisual result in the game by using animated 3D models and sounds that maintain the player's attention.

1.7 Planning

Table 1.1: Planning of the Final Degree Project.

Dedicated hours	
Hours	**Task**
20	Neural networks algorithm's research
20	Modeling and animating video game components
80	Video game programming (gameplay, mechanics, Head-Up Display HUD, insert 3D models and animations)
50	Adding neural networks into the video game
30	Creation of the analysis and design document
70	Report of the project
30	Prepare presentation
10	Creation of video about the project
End of Dedicated Hours	

CHAPTER 2

GAME DESIGN DOCUMENT

2.1 Introduction

TAJ is a Tower Defense video game for two players who will try to conquer the enemy's power stones by using a different kind of totems and power-ups on a map based into two riverbanks and three paths that connect them.

2.1.1 Game concept

In this adventure, the player finds himself in a village called *TAJ*, known by everyone as the greatest power source on the Earth. As legend has it, the one who dominates the territory of *TAJ* will shape the future of the planet by attracting cosmic energy that will cause either prosperity or decay.

Nature is defenseless of innumerable dangers caused by the power of *Nerta* totems. The continuous power battles between *Nerta* and *Ittla* factions to obtain the power of *TAJ* will determine the continuity of the harmony between men, animals, plants, and Earth.

The player will decide to choose between helping the Earth by joining *Ittla* or getting the eternal mandate in the darkness by joining *Nerta*.

2.1.2 Main features

TAJ is a video game with two salient features:

- Simple approach: Make three totems reach the enemy's base to conquer the enemy's power

stones and win the battle.

- Tactic-mental game: The way to enter totems into the enemy's base depends both on how the player plays and how the enemy plays.

2.1.3 Genre

TAJ is composed of the mixture of two video game genres:

- Arcade: Is a game with short battles, based on easy to understand and addictive mechanics.
- Tower defense: The goal is to defend possessions (power stones) and conquer the enemy's ones.

2.1.4 Purpose

The purpose of *TAJ* is to offer players a fast and simple entertainment. At the same time, it is interesting to get players competitiveness and force them to play with different tactics that make them better against different opponents.

It will exist an arduous NPC for the players to train before playing against human opponents.

2.1.5 Visual style

The representation of the video game will be based on 3D models that will contribute to the game experience. A pleasant environment with contrasted and saturated colors will be used to attract more players and encourage them to play.

Both the environment, based on a tropical island, and all the characters and power-ups will be modeled in low-poly to get a casual look and nothing overwhelming.

2.2 Game mechanics

This section talks about the gameplay of *TAJ* and details the mechanics that the player can perform to achieve their goals. It also offers a list of both characters and power-ups that make up the game.

2.2.1 Gameplay

The village of *TAJ* is divided into two clearly separated riverbanks that make up the balance of power. Each one of these riverbanks has three power stones. The riverbanks are connected to each other with 3 different paths through which the totems can move.

Each player will have 3 different types of totems that can be used on one of the 3 paths only every so often because they need to be charged to be used. These totems, once used, will advance

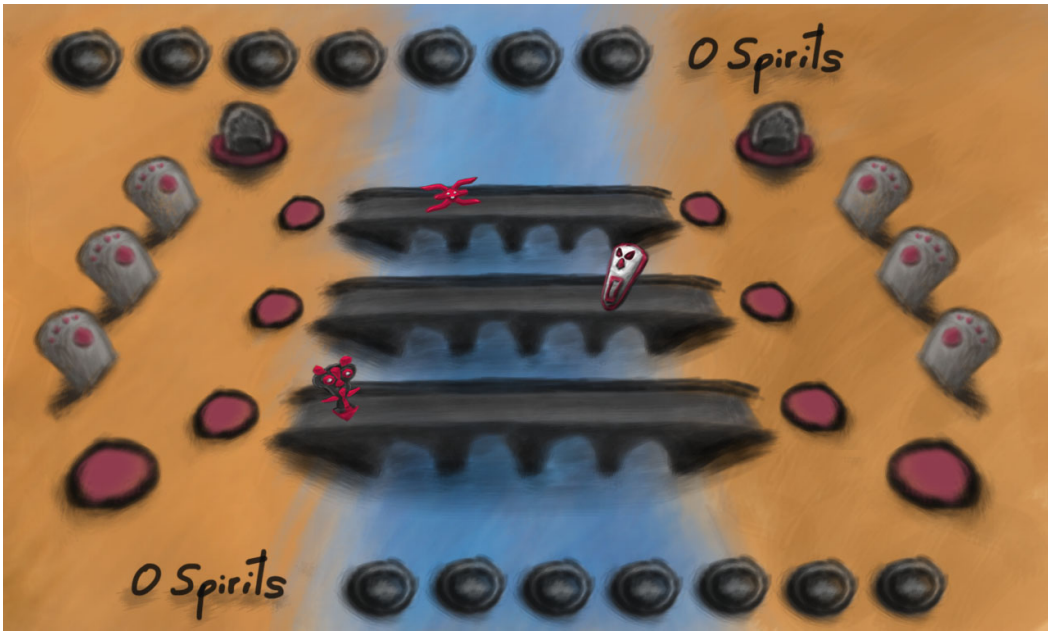

FIGURE 2.1. Sketch of the *TAJ* Game screen.

towards the opponent's riverbank. If on the way to the enemy's base they find an enemy totem, they both will fight. The one who remains alive will continue advancing to the enemy's base. The winner will be the player who manages to enter one totem (of any type) in each path of the enemy's riverbank.

In addition, the player will have the option to insert totems in a shrine (instead of in a path). This insertion accumulates *spirits* which can be used together to get benefit from power-ups that will suddenly change the course of the game.

When entering a totem into the enemy's riverbank by using a path, the first totem introduced will conquer the power stone related to this path. The following totems will not conquer any other power stone.

There will be 4 power-ups:

- *The force of ice* will paralyze every enemy totems at all. One *spirit* is needed to use it.
- *The force of wind* will accelerate the advance of the totems of the player who uses it towards the opposite base. One *spirit* is needed to use it.
- *The force of the sun* will reload all the totems charge of the player who uses it and it will allow to insert them instantly. One *spirit* is needed to use it.
- *The force of the faith* will eliminate all the enemies of the selected path. This is the one which will require more *spirits* to be spent. Two *spirits* are needed to use it.

2.2.2 Game environment

TAJ will be the village where the power battles will take place. As commented in Section 2.2.1, the village of *TAJ* is divided into two clearly separated riverbanks.

Each player is located on one side of the screen just to create a well-known confrontation environment easy to understand for the user.

Each riverbank has three gaps where it is possible to insert the different totems. Each one of this three gaps is part of each one of the three paths that connect both riverbanks. To place a totem in one of this gaps means to send them to fight towards the enemy's riverbank.

In addition to this, each riverbank has one shrine, where the totems are able to get *spirits*.

2.2.3 Challenges

The main challenge that the video game proposes is to defeat the different opponents against whom the player plays. There can only be one winner in each power battle.

There is a global victories-defeats rank of players that will keep the players' competitiveness on each battle trying to keep his status.

2.2.4 Rules

TAJ presents a series of basic rules:

- A totem can only be inserted into a gap if it has enough charge to be used.
- The use of a totem in any gap or in the shrine will empty the charge of this totem. Each totem has a certain recharge speed.
- The use of a totem in the shrine will give the player one spirit.
- The use of a totem in a gap will make it move towards the enemy's riverbank.
- Each totem has certain units of life. If one totem faces another on the same path, both will lose as many units of life as the enemy owns.
  - Example: Totem A with 3 units of life versus totem B with two units of life. The result: Totem B will die and totem A will continue moving towards the enemy's riverbank with 1 unit of life.
- Each power stone is related to a path and it can only be conquered by "the first" enemy totem that reaches the riverbank using this related path.

- If a totem reaches the riverbank of the enemy using a path "and it is not the first", it will not conquer another power stone not related to its path.
- If a player achieves three enemy's power stones (although he has previously lost two or fewer stones), he will automatically win the power battle and the game will be over.
- A power-up can only be used if the player owns the needed *spirits* for that power-up.

2.2.5 Characters and Enemies

Nerta and *Ittla* are the two factions in the universe of *TAJ*. Both factions count with the same kind of totems and the only difference between them are the color of their energy:

- *Vasu* (see Figure 2.2)
  - Description: It symbolizes strength, endurance, and greatness.
  - Features:
    - Life: 6 life units
    - Moving speed: 1 space unit per time unit
    - Charging speed: 10 seconds for a full charge

FIGURE 2.2. Sketch of the *Vasu* character.

- *Kaapo* (see Figure 2.3)
  - Description: It symbolizes determination, balance, and serenity.
  - Features:
    - Life: 4 life units
    - Moving speed: 2 space units per time unit
    - Charging speed: 5 seconds for a full charge

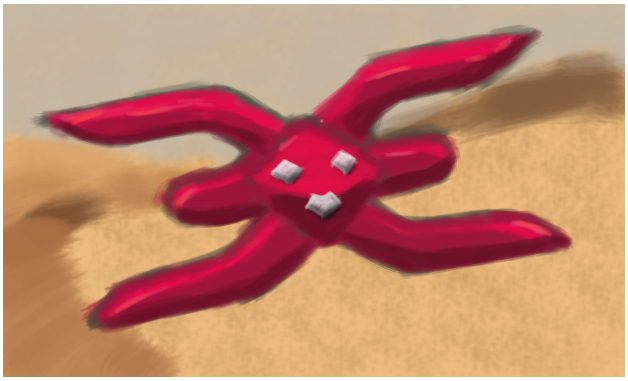

FIGURE 2.3. Sketch of the *Kaapo* character.

- *Rad* (see Figure 2.4)
  - Description: It symbolizes speed, cunning, and elegance.
  - Features:
    - Life: 2 life units
    - Moving speed: 3 space units per time unit
    - Charging speed: 2.5 seconds for a full charge

FIGURE 2.4. Sketch of the *Rad* character.

2.3 Interface and controls

2.3.1 Screens' flowchart

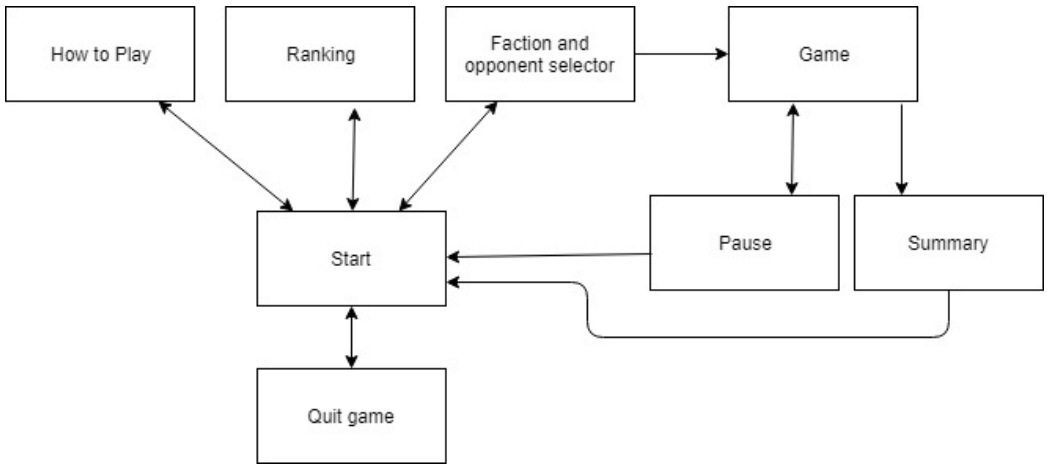

FIGURE 2.5. Screens' flowchart of *TAJ*.

2.3.2 Start screen

The *Start* screen consists of the *Play*, *Ranking*, *How to Play* and *Exit* buttons.

- *Play* will take the player to the *Faction and opponent selector* screen and follow the usual flow to play a power battle.
- *Ranking* will open a pop up which will contain a list of players, their victories, and their defeats.
- *How to Play* will open a pop up which will contain a small text teaching the player how to play *TAJ*.
- *Exit* will show the player a confirmation screen with two options:
  - *Yes* will close the game and take it back to the operating system.
  - *No* will close the dialog box and return to the *Start screen*.

2.3.3 Faction and opponent selector

The *Faction and opponent selector screen* let the player choose which faction to play with, *Ittla* or *Nerta*. It also let him choose the opponent, human or NPC. It includes the button *Back* and *Play*.

- *Play* is also available if the faction and opponent have been previously chosen. Once clicked, it will take the player to the *Game screen*.
- *Back* will take the player back to the *Start* screen.

2.3.4 Game

The *TAJ* Game screen will be composed by:

- The decorative 3D environment of the game inspired by a tropical island. This topic is further explained in Section 2.5.
- The playable scenario of the game. Two riverbanks, three paths connecting both riverbanks, one gap on each side of each path, one shrine on each riverbank and three power stones.
- Head-Up Display (HUD) (see Figure 2.1):
  - Several icons will be located on the top left corner representing the game status of the player placed in the left riverbank. The icons will be seven, three for the totems and its charge and four for the power-ups. A label to see how many available *spirits* the player has will be also visible.
  - Same information icons as in the top left corner will be located on the bottom right corner, but these ones will represent the game status of the player placed in the right riverbank.
- The pause pop up menu will be only visible if the *Pause* button is pressed and it will contain two buttons:
  - *Exit* that will finish the current game and take the player to the *Start* screen.
  - *Resume* that will close the pause pop up menu and resume the current game.
- The game summary pop up will be only visible when the game is over. It will report the players some game statistics and who has won. It will also contain an *Exit* button that will take him to the *Start* screen.

2.3.5 Controls

The whole gameplay is controlled with the mouse.

- Using a totem:
  - If the totem has a full charge, the player only has to left click the icon on the HUD that represent the totem. Afterwards, he will see the four gaps where he can place it highlighted (the three paths and the shrine path). The next step is to left click over the gap he wants to place the totem at and the totem will start moving from them to the enemy's riverbank.
  - If the totem has not a full charge, the player cannot left-click over its icon. If he left-clicks on it, nothing will happen.

- Using a power-up:
  - If the player has enough *spirits* to use a power-up.
    - If the player chooses *The force of the faith*, he will see all paths highlighted. He will have to left-click on the path he wants to use the power-up over and the power-up will take effect.
    - If the player chooses a power-up different from *The force of the faith*, he only has to left click the icon on the HUD and the power-up will take effect.

The *Esc* key will pause the game and show the *Pause* pop up.

Figure 2.6 helps to understand the actions the player has available during the gameplay.

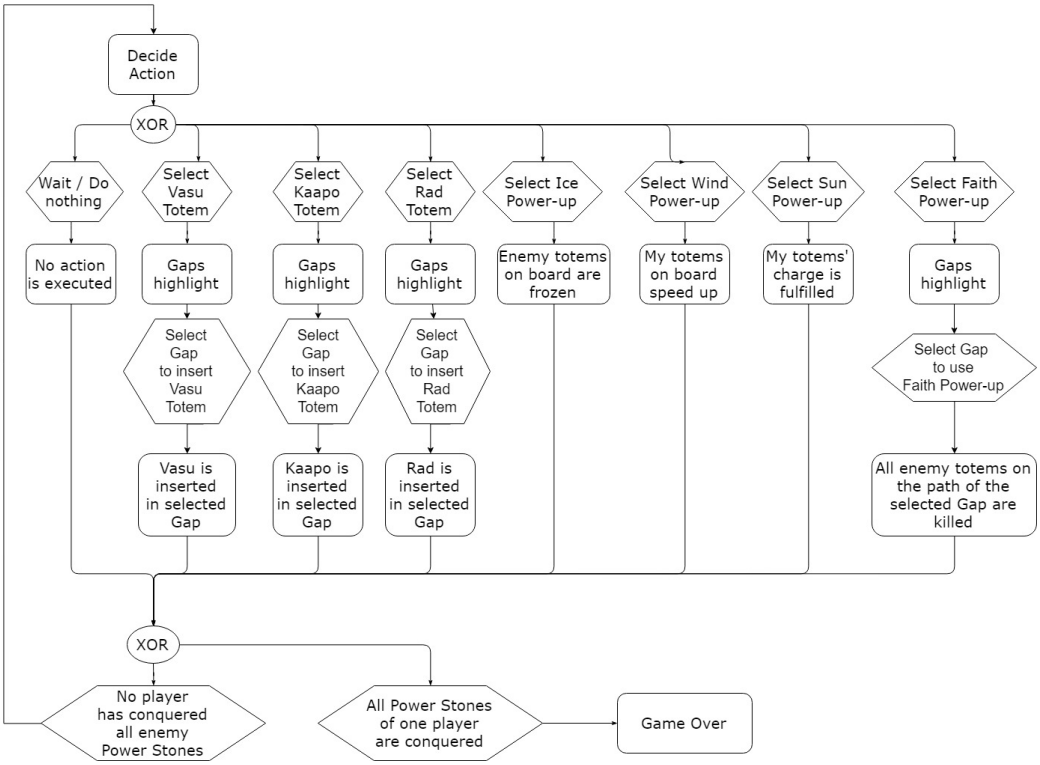

FIGURE 2.6. Gameplay's flowchart.

2.4 Music and sounds

The music and sounds both of the game menu and gameplay wants to show freshness and dynamism, involving the player inside the environment of a tropical island influenced by a battle that will determine the future of the Earth.

There are two main kinds of sounds in *TAJ*:

- Environmental sounds: small decorative sounds that make the player go into the world that is playing, such as the river water, the battles between totems, the acts of faiths, etc.
- Totem and power-up sounds: each action in the game will have a representative sound that will warn the own and enemy's actions. For instance, once a totem is placed, a sound will be played.

2.5 Art

The art of *TAJ* is inspired in the *Donkey Kong Country Returns* (see Figure A.5(a)) video game and in other artistry from anonymous artists of the same style (see Figure A.5(b)). The camera perspective used will be "Top-down", although slightly tilted to highlight the shadows and depth of the 3D models.

The environment will be composed of elements that are usually seen on tropical islands, such as palm trees, shrubs or sand. In addition, the spiritual touch of *TAJ* will make the player see elements such as magic lamps, chains with precious stones or skulls that will further enrich the game's scenario.

Totems, power stones, and every item seen in the gameplay will be modeled in low-poly 3D.

The treatment of color will abuse the use of flat, very saturated and attractive tones, just to draw attention to the player. The idea is to show an ideal fantasy world able to offer a feeling of freshness and freedom.

2.6 NPC and Artificial Intelligence

2.6.1 IA NPCs

In the case where the player decides to play against the machine, *TAJ* incorporates an NPC that uses machine learning techniques to simulate the behavior of a human player and adapt it to the different situations of inside a game. This NPC will improve its skills based on previous game experiences by using neural networks.

This neural networks will consider a lot of variables as proximity to the riverbank, number of enemies on each path, level of charge of own and enemy's totems, number of *spirits*, etc. Depending on the state the NPC will try to choose the best action to defeat its enemy.

2.6.2 Technical aspects

2.6.2.1 Target device

The target devices are the *Windows* [17] personal computers and *Android* [18] devices.

2.6.2.2 Hardware and development software

The hardware needed:

- Computer with *Windows 7* [19] or higher (only 64 bits) or *Mac OS X 10.9+* [20] .
- Central Processing Unit (CPU) with *SSE2* [21] instruction set support.
- Graphics Processing Unit (GPU) compatible with *DX9* (shader 3.0) or *DX11* with feature level 9.3 capabilities [22].

The software needed:

- *Unity3D 2017.3.1f1* [10] engine with the *C#* programming language, for the development of video game and artificial intelligence techniques.
- *Tensorflow 1.4* [23] open source software library for highperformance numerical computation.
- *Machine Learning Agents (ML Agents) 0.3* [4] and TensorFlowSharp as the plugins to hold reinforcement learning along with *Unity*.
- *Anaconda 3 5.1.0* [24], which includes *Python 3.6.4* [25], *Conda 4.4.10* [26], *Jupyter Notebook 4.4.0* [27] under which *Tensorflow* runs.
- *NVIDIA CUDA 9* [28] and *NVIDIA cuDNN 7.1* [29] to speed up the processes of *Tensorflow*.
- *Blender 2.78.3* [30] for the modeling and animation of video game components.
- *Photoshop CC 2017* [31] for the creation of sprites.
- *Google Drive Slides* [32] for the presentation slides
- *Visual Studio [33] Ultimate 2012 4.7.03056* as integrated development environment IDE.
- *Overleaf* [34] to lay out the document.

Chapter 3

Progress of the project

This section relates the development of the project from its beginning, the way things have been done, the problems found on the way and how they have been solved.

3.1 Art of the video game

Five models (sand terrain, *Rad*, *Vasu*, *Kaapo*, shrine and gaps) have been modeled in low poly 3D with *Blender* [30]. The saturated and plain colors of the models have been added by directly coloring faces with materials. Each one of these models has a material which refers to the color of the faction. Once the character is inserted, the game replaces this material with the one that corresponds to its faction. Go to Appendix A to see sketches and screenshots.

The skull [35] and the palm tree [36] models have been downloaded from the *Unity Asset Store* [37] and positioned in the environment to enrich it.

Water was added to the game by using the scripts created by *Blendcraft Solutions* [38].

Sound effects has been downloaded from *Freesound* [39] and musics has been downloaded from *Youtube Audio Library* [40]. Sound effects play whenever its related *action* is done and the music is always in the background.

A *Particle System* pack [41] has been downloaded from *Unity Asset Store* as well. Four of these particle systems have been modified to get a proper appearance for each one of the four power-ups and two more have been edited just to decorate the environment (one for sand swirls and another for a waterfall).

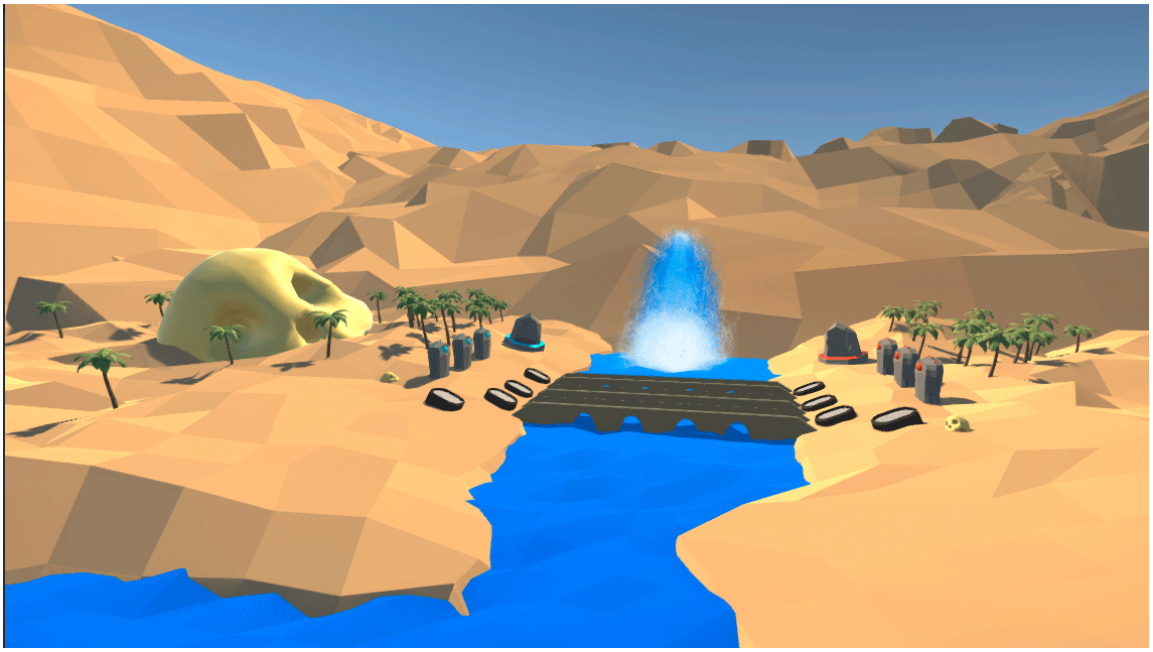

FIGURE 3.1. TAJ Environment.

When the game screen loads, before the battle starts, a horizontal panoramic has been programmed to the camera to show the playing field. In addition, *Post Processing Stack* [42] has been used to add fog, *Fast Approximate Anti-Aliasing FXAA* [43], *Ambient Occlusion* [44] and a vignette effect.

3.2 Neural networks research

Machine learning [7] is a branch of artificial intelligence [45] whose objective is to develop techniques that allow computers to learn.

There are several ways of making machines learn. The one used in this project is the *Reinforcement Learning* [6] technique called *Proximal Policy Optimization (PPO)* [5] whose purpose is to determine what better *actions* a software *agent* should choose in a given environment in order to maximize some notion of *reward* or accumulated prize.

A clear example is teaching a dog how to sit. Since the dog is not able to understand human language, if every time it does things well (for instance, it sits down after listening "Sit") it receives a cookie, it will little by little link the *action* of sitting takes it to get the reward of eating. This principle is basically how *Reinforcement learning* [6] works.

The main objective in this project is to get a neural network [3] composed by a lot of neurons

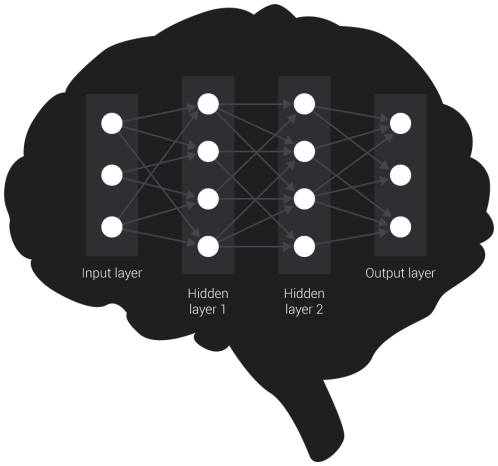

FIGURE 3.2. Scheme of a neural network [46].

(or nodes) connected to each other able to give an output result (*actions* of the NPC, output layer in Figure 3.2) after passing several input variables (environment state/observations, input layer in Figure 3.2).

To get this neural network [3] it is needed to make a repetitive process called training that will define which *actions* lead to have rewards or punishments.

The hidden layers between the input and the output layers (see Figure 3.2) are not human understandable, but having a neural network [3] guarantees that the machine will always address its behavior to get as much *reward* as possible. So, it is very important to think possible logic gaps the machine could take to achieve its goal. For example, in a First Person Shooter (FPS) [47] game [47], rewarding the machine every 5 seconds for staying alive will probably make it avoid confrontations. It will maybe find a place on the game map where it is always safe and it will stay still there forever. For sure, this is not the ideal behavior of an NPC that should entertain the human player and be, somehow, competitive. Rewarding the machine for hurting an enemy or continuing alive after a confrontation would be a much better option.

Therefore, it should be noted the importance of collecting the variables that best represent the state of the game to facilitate the agent's learning (observations).

More information about neural networks can be found in the free online book "Neural Networks and Deep Learning" [48].

3.2.1 Software needed

Unity Machine Learning Agents (ML-Agents) [4] is an open-source *Unity3D* [10] plugin that enables games and simulations to serve as environments for training intelligent agents. The

implementations of *ML-Agents PPO* [49] algorithm are built on top of the open-source library *Tensorflow* [23], originally developed within *Google's AI organization*, which allows deployment of computation across a variety of platforms such as *CPUs*, *GPUs*, and *Tensor Processing Units (TPUs)* [50]. The training made by *Tensorflow* is executed in a separate *Python* process (communicates with *Unity* over a socket) and it generates a model in a file with extension *.bytes* that can be inserted into an Internal *brain* to make *agents* behave with the reached knowledge.

The use of *Tensorflow* along with *Nvidia CUDA* [28] and the *CUDA Deep Neural Network library (cuDNN)* [29] speeds up in a remarkable way (up to 3x) the process of training an *agent* in the current project, since the power of the *CPU* and *GPU* are better-exploited thanks to the parallel processing. In cases where a *GPU* is not present, setting *Tensorflow* properly will make it work only with the *CPU* [51].

A nice platform to install the complex *Tensorflow* environment and work with it is *Anaconda* [24], which takes care of installing more than 250 data science packages, a *Python* [25] compiler, the *Conda* [26] package to allow handling libraries outside the *Python* packages and the *Jupyter Notebook* [27] to allow writing code with embedded visualizations.

Tensorboard [52] is installed along with *Tensorflow* and it allows the visualization of certain *agent* attributes (e.g. learning rate or entropy) throughout training, which can be helpful in both building intuitions for the different model attributes (called hyperparameters) and setting the optimal values for the *Unity* environment.

Since *Tensorflow* does not provide a native *C# Application Programming Interface (API)*, *Unity* offers a third-party library called *TensorFlowSharp* which provides *.NET* [53] bindings to *TensorFlow*, allowing the use of Internal *brains*. This add-on, currently marked as experimental, allows to apply trained models into the Internal *brains* to see the learning progress of the *agent* inside the *Unity* editor.

Summarizing, the software used for integrating machine learning with *Unity* is:

- *Tensorflow 1.4* [23] open source software library for high-performance numerical computation.
- *Machine Learning Agents (ML Agents) 0.3* [4] and TensorFlowSharp as the plugins to hold reinforcement learning along with *Unity*.
- *Anaconda 3 5.1.0* [24], which includes *Python 3.6.4* [25], *Conda 4.4.10* [26], *Jupyter Notebook 4.4.0* [27] under which *Tensorflow* runs.
- *NVIDIA CUDA 9* [28] and *NVIDIA cuDNN 7.1* [29] to speed up the processes of *Tensorflow*.

The installation process of this software is described in Section 3.4.

3.3 Programming the video game

From the beginning of the programming process, it has always been a priority to keep information (*actions* and *observations*) easy to access to facilitate the subsequent integration of neural networks.

Taking into account Section 3.2, the possible *actions* of the game have been clearly differentiated in order to be able to easily add machine learning rewards.

3.3.1 Setting up the game scene

The first thing done was setting up the scene with *GameObjects* [54] positioned the proper way. In the beginning, all of them were planes and cubes just to allow seeing the written code did what it was supposed to do. Art would be done afterwards.

3.3.1.1 Controls and Head Up Display (HUD)

The intro screen menu contains the options seen in Figure A.2. Sketches have been used to layout this screen.

Only one camera has been used, in perspective mode, rotated 45 degrees in order to see every *GameObject* and to allow the user to click with the mouse on the screen buttons (gaps) without trouble.

There is a *Canvas* [55] *GameObject* in the scene that contains the HUD. All buttons in the HUD are disabled from the beginning of a game. Each one of them contains a script that manages their status and enables them when required. Other types of buttons are the gaps at the edges of each path, used to choose the place where the totem has to be inserted. These buttons are located in the game world (not in the HUD), and they are *GameObjects* that contain *Box Colliders* [56].

Depending on the type of button, they are controlled by:

- The *CharacterCharge* script, that manages the speed of charge of each HUD character button and enables them when are fully charged.
- The *ForceButton* script, that enables the HUD power-up buttons only when the player has the minimum spirits that the button requires.
- The *Gap* script, that enables the game world buttons after clicking on a character or a faith power-up enabled button. Enabling them allows the player to click on them to select the path where the selected character has to be inserted or the faith power-up has to be used.

3.3.2 Movement of characters

Totems in *TAJ* start from an edge of a path and they move towards the opposite edge of the same path.

Location and sizes of *TAJ* paths are defined by the topology of several rectangular planes. *Unity* owns a functionality which bakes a walkable mesh by reading the shape of every *GameObject* tagged as *Static* (in this case, the rectangular planes) and allows moving *GameObjects* with pathfinding (it uses A* algorithm [57]) towards positions inside the baked mesh.

Each *GameObject* to be moved inside the *NavMesh* [58] must include a component called *NavMesh Agent* [59]. Inside this *NavMesh Agent* component, the speed of the character can be set.

NavMesh Agents avoid collisions by default, but this is not the behavior *TAJ* needs. It will be needed to set *Obstacle avoidance* to *None*, in order to avoid totems to elude themselves (another way never there would be deaths).

Something to keep in mind is that characters have different speeds and they can overtake other totems of their same teams, so this "no collision" behavior is still useful for this case. It will cause totems to cross each other, but this is a nice option for the little space the paths have. An option to decorate these crossings is making both totems semitransparent when colliding.

3.3.3 Collision detection and life of characters

Whenever a Totem collides with another, life has to be updated. The *Box Collider Unity* component has been used as a trigger for collisions. Each character has a script called *Totem* that stores its life, speed and some other attributes as the faction to which it belongs. It also has a *Unity OnTriggerEnter(Collider other)* method in charge of detecting collisions with that totem. The *Collider other* variable stores the *GameObject* the totem has crashed with and, if it is an enemy, life drops depending on the enemy's life.

When a totem receives an impact, its life is damaged, but if it is still alive, it will continue moving towards the enemy riverbank. If it dies, its movement stops and its 3D model will vanish. So, collisions are only being detected for life management, but totems are not avoiding each other and physics are affected by these collisions neither.

3.3.4 Power-ups

Ice Force, *Wind Force*, *Sun Force* and *Faith Force* are the four power-ups (or *force buttons*) that makeup *TAJ*. All of their *actions* are implemented in the *GameController* script in separate functions that are called when a HUD button is pressed.

GameController maintains two lists of *GameObjects* that result really useful for the *TAJ* power-ups needs. One of them maintains a list of the alive *Ittla* totems that are on the board. The other keeps the same information for the Nerta totems.

- *Ice Force power-up* function takes all the totems of the enemy faction that are alive and on the board. Then it gets their Nav Mesh *Agents* and stops their movement. Later, by adding *deltaTime* to a variable, when a constant number of seconds is reached, the movement of these Nav Mesh *Agents* is resumed.
- *Wind Force* takes the list of alive totems, but this time the ones of the own faction. For each of them, the speed of their Nav Mesh *Agent* is updated (multiplied by a constant number). As before, after a few seconds, the speed is again restored to the previous value.
- *Sun Force* power-up function fills up to the 100% the current charge of the character buttons, so they will become active and the player will be able to insert those characters into the board.
- The first *action* that the *Faith Force* power-up function takes is to enable all the gaps of the own faction, just to make them clickable. Afterwards it keeps waiting for a click over a gap. When a click over a gap is detected, it will look for the enemy totems located on the same path as the clicked gap, and all of them will be instantly killed.

3.3.5 Class diagram

Unity allows to work with scripts (*C#* classes in this project) attached to *GameObjects* as *Components*. Necessary references are passed to these *Components* so that, at runtime, the project works correctly.

This project has been much more oriented to get a functional game where the ML Agents plugin could be tested and a nice result in terms of machine learning could be obtained. Since this is the main purpose of the project and because of the project timing, the game scripts structure is all focused to easily pass information to ML Agents plugin, and it is quite improvable to make it more maintainable for future changes. As an improvement proposal, if the game had to expand its features and go on sale (outside this project), creating a class structure to separate the managing of different entities would be a needed task.

Anyway, it is useful to have in hand the composition of the classes, their attributes, and methods. A summarized class diagram can be seen in Figure 3.3, and the complete one [60] can be accessed from the following url .

`https://drive.google.com/open?id=1UXq8Opy5zAJARaXroWeZIMcMFPYY61UJ`

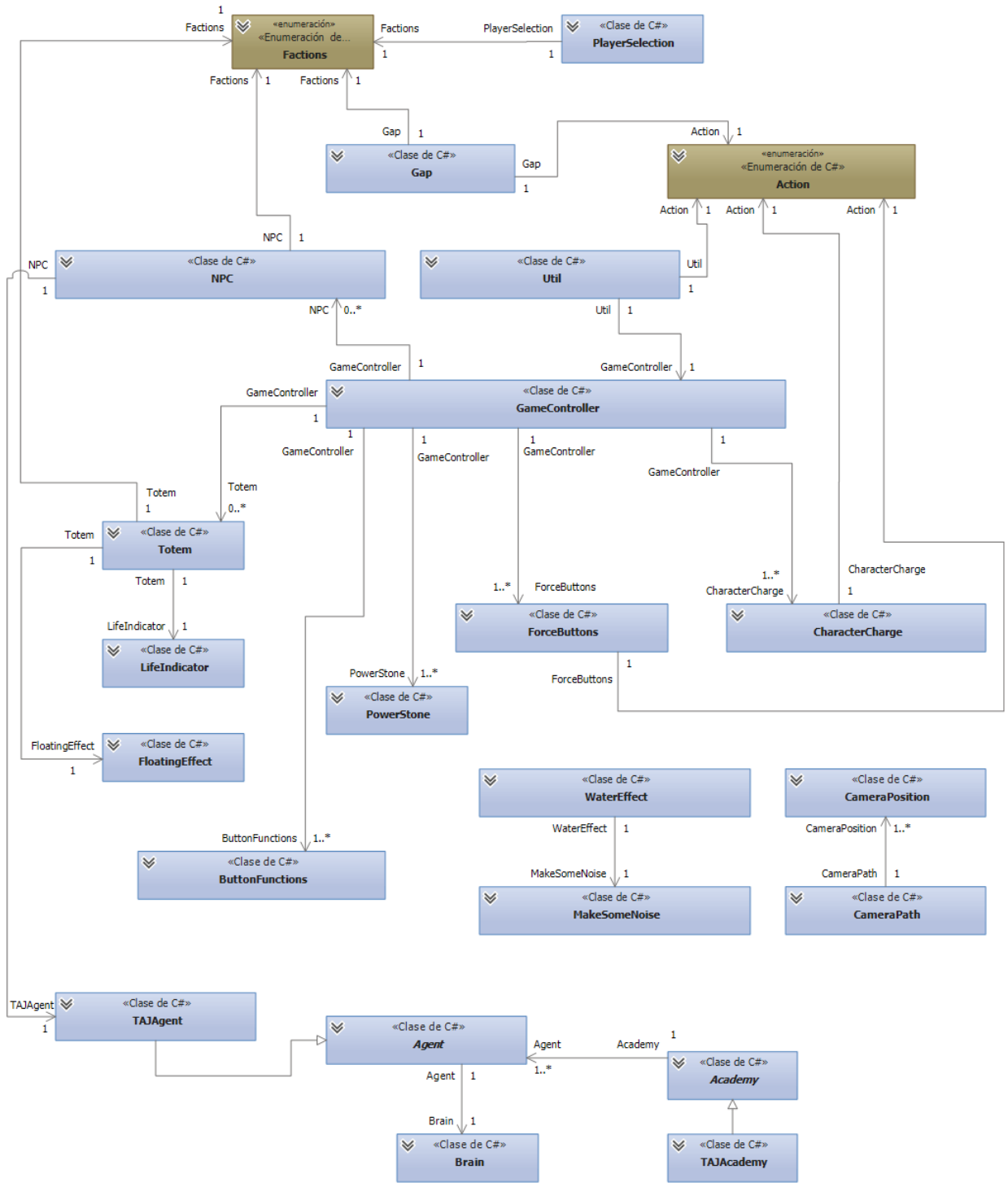

FIGURE 3.3. Class diagram of TAJ summarized.

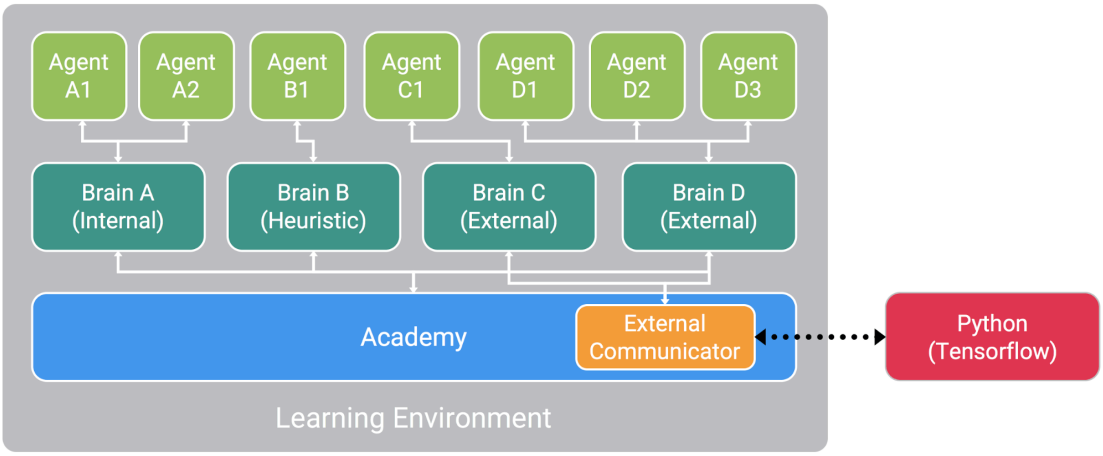

FIGURE 3.4. Learning Environment. Image from *Introducing: Unity Machine Learning Agents* [61]

3.4 Adding neural networks into the video game

The first step was creating an NPC able to play the game taking always random *actions*. The reason for this is that the ML *agent* will train playing against it, because the more different situations shown to this *agent*, the more it will learn. This random NPC will also be used to get results and see if the *agent* gets a great winning percentage playing against it.

It is a nice start to centralize the environment states and the available *actions* the player can take for later adding *rewards* for machine learning. After finishing this step, the process continued with the setup of the required software and the environment.

Figure 3.4 shows the big picture of the learning environment (*Unity* project) and how *ML Agents* components are connected. An external communicator has the responsibility to connect the learning environment with *Tensorflow*. The utility of each software that *Tensorflow* needs has been justified in Section 3.2.1).

3.4.1 Setting up environment

The project has been developed in *Windows 10* [62] and here it is described how to set up the environment under this operating system. It is very advisable to read the *ML Agents* installation documentation in its *GitHub* repository [4], and for the required dependencies is nice to lean on the *Unity 3D College tutorial* [63] and its installation video [64].

First of all, it is necessary to download the *CUDA toolkit* executable from the *NVIDIA CUDA* archive [28] and to follow the steps of till get it installed. Afterwards, is needed to download *CUDA Deep Neural Network library* from the *NVIDIA cuDNN* website [29] for the chosen version of the

installed *CUDA* toolkit. This installation is as easy as copying and replacing the downloaded *cuDNN* files into the *CUDA* folder.

To allow *Tensorflow* to find *CUDA* in the computer, some *Windows* environment variables must be set:

- The system variable *CUDA_HOME* must be added, and its value must be the *CUDA* installation path (something like *C:/ProgramFiles/NVIDIAGPUComputingToolkit/cuda/v8.0*)
- Two values must be added into the already existing *Path* system variable. These values are the paths to the *CUDA* and *cuDNN* libraries (something like *C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v8.0/lib/x64* and *C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v8.0/extras/CUPTI/libx64*).

The next step is configuring a *Python* [25] environment by downloading *Anaconda* from its download page [24] and installing it.

Once *Anaconda* ends its installation, is time to open its prompt to create the *Tensorflow* environment by using the *conda* command, activate it and finally installing *tensorflow-gpu*.

The exact process is:

- Openning *Anaconda* prompt
- Executing *conda create -n tensorflow-gpu python=3.5.2*
- Executing *activate tensorflow-gpu*
- Executing *pip install tensorflow-gpu* . This command installs the last version of *TensorFlow*. Section 4.3 refers a compatibility problem between *Unity* and *Tensorflow*.
- Executing *import tensorflow as tf*

Subsequently, is needed to download the *Unity*'s sample project from *Unity*'s *ML Agents* repository [4] to get the *Python* files which would allow training *Agents* by using *PPO Reinforcement learning*.

Last but not least, inside another *Anaconda* prompt it is needed to dive into the downloaded Unity project to execute the command *pip install*.

That is everything for the environment setup.

3.4.1.1 *Observations*, *actions* and *rewards* for the video game

Reinforcement learning [6] tries to give the best *action* given several *observations*. The *observations* have to be enough representative of the game status. Every time the *machine learning Agent* is asked to take a *decision*, it will take it based on these *observations* and the previous

experiences. In *TAJ*, the 44 chosen *observations* are parsed to *int* [65] because of the needs of *ML Agents*. All *observations* explained in Table 3.1 are considered twice, once for the *agent* and once for the enemy. So, the *agent* observes its life and the enemy's one, its conquered power stones and the enemy's ones, and so on.

Unity ML Agents team recommends in its *Making a New Learning Environment* document [66] to set all *observations* and *actions* clamped to the range of [-1,1] if they are *Continuous* (types of variables are explained in Section 3.4.2.2), for two reasons, to avoid numeric instability in numerical calculations and to always limit actions to reasonable ranges.

Table 3.1: *TAJ* Game *Observations*

TAJ Game *Observations*		
Observation	**Values (int)**	**Description**
Life	[0,1,2,3]	Indicates the quantity of *power stones* still conquered.
Power stone 1 is conquered	[0,1]	0 if the *power stone 1* is still intact. 1 if it has been conquered.
Power stone 2 is conquered	[0,1]	0 if the *power stone 2* is still intact. 1 if it has been conquered.
Power stone 3 is conquered	[0,1]	0 if the *power stone 3* is still intact. 1 if it has been conquered.
Spirits	[0,1,...,n]	The number of spirits.
Total life in path 1	[0,1,...,n]	The total life of the *agent*'s totems in the path 1.
Total life in path 2	[0,1,...,n]	The total life of the *agent*'s totems in the path 2.
Total life in path 3	[0,1,...,n]	The total life of the *agent*'s totems in the path 3.
Total life in shrine path	[0,1,...,n]	The total of life of the *agent*'s totems in the shrine path.
Total life of totems near end of path 1	[0,1,...,n]	The total life of the *agent*'s totems who have exceeded the 50% of the path in the path 1.
Total life of totems near end of path 2	[0,1,...,n]	The total life of the *agent*'s totems who have exceeded the 50% of the path in the path 2.
Total life of totems near end of path 3	[0,1,...,n]	The total life of the *agent*'s totems who have exceeded the 50% of the path in the path 3.

Continuation of Table 3.1		
Observation	**Values (int)**	**Description**
Total life of totems near end of shrine	[0,1,...,n]	The total life of the *agent*'s totems who have exceeded the 50% of the path in the shrine path.
Vasu enabled	[0,1]	0 if the *agent*'s *Vasu* is not charged. 1 if the *agent*'s *Vasu* is charged and able to be used.
Rad enabled	[0,1]	0 if the *agent*'s *Rad* is not charged. 1 if the *agent*'s *Rad* is charged and able to be used.
Kaapo enabled	[0,1]	0 if the *agent*'s *Kaapo* is not charged. 1 if the *agent*'s *Kaapo* is charged and able to be used.
Ice Force Enabled	[0,1]	0 if the *agent* does not have enough spirits to use *Ice Force* power-up. 1 instead.
Wind Force Enabled	[0,1]	0 if the *agent* does not have enough *spirits* to use *Wind Force* power-up. 1 instead.
Sun Force Enabled	[0,1]	0 if the *agent* does not have enough *spirits* to use the power-up. 1 instead.
Faith Force Enabled	[0,1]	0 if the *agent* does not have enough *spirits* to use the power-up. 1 instead.
Ice Force Is In Use	[0,1]	0 if *Ice Force* is in use. 1 instead.
Wind Force Is In Use	[0,1]	0 if *Wind Force* is in use. 1 instead.
TAJ Game *actions*		

It is not that important to consider how many totems are in each path and their remaining lives, because it increases a lot the number of observations and it is not that representative in this game. It is preferable to reduce all that variables to the total life of totems in each path because the goal is to kill all enemies.

Actions in *TAJ* are quite clear and they are defined in the Table 3.2.

Table 3.2: *TAJ* Game *Actions*

TAJ Game *Actions*				
Action ID	**Description**		**Action ID**	**Description**
0	Insert *Vasu* in Path 1		9	Insert *Kaapo* in Path 2
1	Insert *Vasu* in Path 2		10	Insert *Kaapo* in Path 3
2	Insert *Vasu* in Path 3		11	Insert *Kaapo* in Shrine Path
3	Insert *Vasu* in Shrine Path		12	Use *The Force of Ice* power-up
4	Insert *Rad* in Path 1		13	Use *The Force of Wind* power-up
5	Insert *Rad* in Path 2		14	Use *The Force of Sun* power-up
6	Insert *Rad* in Path 3		15	Use *The Force of Faith* in Path 1
7	Insert *Rad* in Shrine Path		16	Use *The Force of Faith* in Path 2
8	Insert *Kaapo* in Path 1		17	Use *The Force of Faith* in Path 3
TAJ Game *Actions*				

A lot of information is needed to allow machine learning *Agents* learn to play a game. *Rewards* are the way to indicate the *agent* is doing well or not (given the current status).

The table 3.3 details the *rewards* used in the game. In Section 4.2, some mistakes made with the values, and with the moment the *agent* was being rewarded in initial tests, will be commented.

As rewards table 3.3 shows, constantly repeated good *actions* are not rewarded too much. On the contrary, *actions* less repeated have much bigger values and nice *actions* are always proportionally bigger than bad *actions*.

Something to keep in mind is that the *agent* (as further explained in Section 3.4.2.3) will take one of all the *actions* previously declared (see Table 3.2) each time it requires for an *action*. The game has to be programmed the way it only really does an *action* if the game status allows it. For example, an *agent* cannot use a power-up if it has not enough spirits to use it. It is not needed to penalize the *agent* for choosing not available *actions* or to reward it for choosing available ones, since its experience in the training will lead it to choose the right *actions*.

Table 3.3: *TAJ* Final *Rewards*

TAJ Final *Rewards*	
Reward	**Description**
-0.05	If the *agent* loses life (enemy has conquered a *power stone* of the *agent*)
+0.3	If enemy loses life (the *agent* has conquered a *power stone* of the enemy)
-0.2	The *agent* dies (enemy has conquered all *agent*'s *power stones*)

Continuation of Table 3.3	
Reward	**Description**
+1	The *agent* wins (*agent* has conquered all the enemy's *power stones*)
+0.1	The *agent* inserts a totem (*Vasu*, *Rad* or *Kaapo*) in a path where the enemy's *power stone* is not yet conquered.
-0.002	The *agent* inserts a totem (*Vasu*, *Rad* or *Kaapo*) in a path where there are no enemies and it has already conquered enemy's *power stone* of this path.
+0.05	The *agent* inserts a totem (*Vasu*, *Rad* or *Kaapo*) in a path where the total life of the enemies adds up the same or less life than the *agent*'s totems in this path along with the totem the *agent* is inserting.
+0.05	The *agent* inserts a *Vasu* totem in a path where the total life of the enemies adds up more than the *Vasu* initial life (since it is the better option to try to kill enemies).
+0.1	The *agent* uses *Ice Force* when there is, at least, one enemy totem on any path (so, the power-up can stop the movement of some enemy totem).
+0.1	The *agent* uses *Wind Force* when there is, at least, one of its totems on any path (so, the power-up can speed up the movement of some of the *agent*'s totems).
+0.05	The *agent* uses *Sun Force* when, at least, one of its totems has less than 100% charge.
+0.2	The *agent* uses *Faith Force* in a path where there was already, at least, one enemy totem (so, the power-up can kill some enemy totem)
TAJ Final *Rewards*	

In the process of setting up *observations*, *actions*, and *rewards*, paths have been reduced from five (initial idea) to three, since *observations* and *actions* were considerably reduced and game experience was attractive as well. This has helped the *agent* to learn faster because the problem became easier.

3.4.2 Creating custom *agents* for the video game

After setting up the environment (explained in Section 3.4.1), next step is to create a custom *agent* based on the *observations*, *actions*, and *rewards* of the Section 3.4.1.1.

It is advisable to keep an eye on the *Getting Started with 3D Ball Environment* from the *Unity ML Agents GitHub* documentation [67] and run into the example to get used to *ML Agents* with *Unity*. Afterwards, it is recommended to do an own simple project from scratch to dive a little more into basics before entering fully into a bigger case. An idea for this simple project can be one from the *Unity*'s examples [68]. The first contact with *ML Agents* in this project was training

a lot of balls spawned in a random position between -1 and 1 in the x-axis till they learned to move to the 0 x position.

The *Step* is an important term that should be well understood from the beginning to avoid misunderstandings when working with ML *Agents*. There are two kind of *steps*:

- *Steps* of an *academy* or environment steps. A measure of time which actually corresponds to a *FixedUpdate*, so it does not actually correspond to frames or framerate. In the case the game uses physics, everything related to physics must be coded in *FixedUpdate*, since *Update* could lead to weird behaviors. These *steps* are used to measure when the *academy* environment has to be reset.
- *Steps* of an *agent*. A step of an *agent* is a decision. These *steps* are used to measure the length of the training sessions.

3.4.2.1 Academy

The *academy* is a *GameObject* that has to live inside the scene where the training happens (learning environment), and it includes a script (that inherits from the *Academy Unity* class) which allows setting some parameters as the time scale (speed), the quality or the size of the window of the game when training or inference. Inference allows using models without continuing training them. It is used to manage the environment.

Python and *Tensorflow* are key components to work with *ML Agents* because *Unity* is not able by itself to implement machine learning with so much efficiency. The most significant task the *academy* has is to communicate (through the *External Communicator* that lives inside it) with the *Python* API that manages the training .

The *Academy* class allows to override some methods:

- *AcademyReset()* is the function used to set up the environment at the beginning of each episode. In *TAJ*, all variables are reset here to initials, and the game is also restarted. This function is automatically called when *Academy.Done()* is executed.
- *AcademyStep()* executes every *step*. In *TAJ* it is used to check if a battle has ended (a *boolean* is checked), just to call *Academy.Done()* and reset all the environment. This function could also be used to update the environment by adding new objects to the problem incrementally.

Max Step variable of an *academy* specifies the max number of *steps* the *academy* can do before calling *AcademyReset()* automatically. If set to 0, *Academy.Done()* has to be called manually from code whenever is needed (it also runs *AcademyReset()*), otherwise, it is important to make sure that the main sequence of *actions* of an *agent* fits inside the *Max Step* specified *steps*. When the moment to reset is not that clear as in the *TAJ* project and it is wanted to do them manually, it

is typically best practice to have the simulation reset every once in a while (by using *Max Step* greater to 0).

3.4.2.2 Brain

A *brain* is a *GameObject* that lives as a child of an *academy* and includes a script (that inherits from the *Brain Unity* class).

The main function of the *brain* is "thinking" which *actions* the *agent* has to do (encapsulates decision making). The *agent* gives the *brain* the *observation* of the environment each time a decision is required. The *brain* returns an *action* (depending on previous experiences). This *action* produces a reward which the *brain* will consider for future decisions, leading the *agent* to the best *actions*.

Brains work with two main types of spaces:

- *Continuous* means that the variables will be numbers (*floats* or *integers*) that will compose the different (and sometimes countless) status in which *agent* can find itself. Most projects use this type of space.
- *Discrete* means the variables are essentially id's into tables of states. It is a less used type of space. It can easily be related to projects which need few *observations*.

Main parameters of the *brain* are the *Vector of Observations*, the *Visual Observations*, the *Vector Action* and the *Brain Type*.

The Vector of Observations stores the type (continuous or discrete) and size (quantity) of the *observations* needed for the current project. It also allows selecting the number of *stacked vectors* to keep within the training. *Stacked Vectors* means keeping in mind the last x vector *observations* to decide *actions*, which can be useful for projects where memory is important for the *agent*. An example that would need *stacked vectors* is an autonomous car traveling towards a fork that has to remember if the previous traffic signal indicates its destination to the right or to the left.

Visual observations allow learning from the difference between several images along time.

Vector Action stores, as the *Vector observations*, the type (*continuous* or *discrete*) and the size.

The Brain Type defines the behavior of the *brain* and the way it chooses its decisions. Defining a *Brain type* as *External* would make the *brain* to constantly expect orders from an *external application*, *Tensorflow* in this case, through a socket. Setting it as *Internal* makes the *brain* to look for an embedded graph previously trained (*.bytes* file) which contains the logic to choose one *action* or another. To summarize, *External* will be used for training and *Internal* for using a trained model. There are other *Brain Types*, *Heuristic* which allows overriding the default

Decision behavior and *Player* which helps to check every coded *action* to see everything works as expected (*actions* and their *rewards*).

So, after knowing a bit about how *Brains* work, *TAJ Brain* is set up this way (remember Section 3.4.1.1):

- *Vector of Observations* with a continuous space type and a space size of 44 (number of observations needed in *TAJ*, see table 3.1).
- *Stacked Vector Observations* with the value of 1. *TAJ* is a game in which all the *actions* are taken depending on the *observations* of the current moment, without needing to remember what has happened previously in order to win the game.
- *Visual Observations* to 0, since this is not needed for a project like a tower defense to learn from images.
- *Vector Action* with a discrete space type and a space size of 18 (*actions* needed in *TAJ*). Discrete is the better option because *actions* are clearly limited to the 18 seen in Table 3.2.

It is well to emphasize that, although the *Brains* define the type and size of the vectors of *observations* and *actions*, is the *agent* which assign their values.

Another important thing is that there can be a lot of *Brains* learning independently, or several *Agents* feeding the same *brain* to speed up the learning process. Only one *brain* and an *agent* has been used in *TAJ* because of the project timing, but using several *brains* would have reduced the time and the steps needed for the learning process.

3.4.2.3 Agent

An *agent* is a *GameObject* containing a script that inherits from the *Agent Unity* class and its main function is giving *observations* and *rewards* to the *brain* at the same time as computing the *actions* the *brain* returns to it.

Every *agent* must have a *brain* which will decide the *actions* the *agent* has to do. *Visual Observations* can be added to the *agent* by using a camera to observe the environment.

The *Max Step* variable of the *agent* is different from the one later commented (in the Section 3.4.3) and from the *Max Steps* seen in Section 3.4.2.1. The *Reset on Done* boolean determines whether an *agent* has to start over. , while the *max_steps* of the *trainer_config.yaml* determines the *steps* to do in the current training and *Max Steps* of *academy* says when the environment has to be reset. Do not confuse the three variables although their names are almost the same.

- Having *Reset on Done* to true makes the *agent* auto reset after reaching the *agent*'s *Max Step* count. This *Max Step* is 0 in *TAJ Agent* because it is wanted to manually reset the

agent (as did with the *academy*) every time a battle ends, and each battle can have different time lengths.

- *max_steps* of the *trainer_config.yaml* determines the *agent*'s steps to do in the current training. The *trainer_config.yaml* file is further explained in Section 3.4.4.
- *Max Steps* of *academy* says how many environment *steps* to do before the *academy* auto resets. *TAJ* sets it to 0 because here the *academy* is reset at the same time as the *agent* (whenever a battle ends).

The On Demand Decision check is quite important since it says if the *agent* has to do automatic decisions every *x steps* (these *steps* are set in the *Decision Frequency* variable) or just every time *RequestDecision()* is called. *TAJ* uses *On Demand Decision* to true and it is manually specified when to ask for a *decision*. The main reason is that this game does not need *decisions* to be constant and repeatedly made in very short spaces of time. *Decisions* and *actions* have to be made at the same time in *TAJ*. The *decision*'s default behavior when *On Demand Decision* is checked to *false* and a *Decision Frequency* is specified is performing an *action* every *Academy Step* and requesting a *decision* every *Decision Frequency steps*. As this is completely not the way *decisions/actions* have to work in this project, the use of coroutines is the adequate way of measuring when decisions have to be requested in training.

The random NPC works under a coroutine that is executed every *x* seconds. To make the game (and training) fairer, both the random NPC and the *agent* ask for a decision at the same moment, the same number of times, otherwise the *agent* would make many more decisions and that would provide it an advantage.

For the final generated models, it is not only wanted that the *agent* chooses the best *action*, but also to make it play "as a human-like". The decision frequency is managed with coroutines too, and it is commented in Section 3.5.

Reinforcement learning in *ML Agents* works following the *Markov Decision Processes* [69]. Within this context, the main functions within the learning loop are introduced. Calling the *agent*'s *RequestDecision()* executes the following process, called Experience:

1. Seeing the environment with the *CollectObservations()* function.
2. Making an *action AgentAction()* function.
3. Get *rewards* returned from the *AgentAction()* function.

The *Agent* class contains a few functions that can be overridden to adapt it to the current project:

- *InitializeAgent()* is used to set the initial variables the *agent* needs to work properly. *TAJ*'s *InitializeAgent()* function defines the initial life and spirits, which will then be useful to check if the previous *action* took the Agent to win or lose a life or a *spirit* and to return the corresponding *reward*. A lot of different variables can obtain value in this function, allowing to adapt the *agent* to any custom project.

- *CollectObservations()* is executed every time a *RequestDecision()* is called. By using the *AddVectorObs()* function and passing a number to it (*integers* in *TAJ*), the vector of *observations* get values. *TAJ* uses *AddVectorObs()* 44 times inside *CollectObservations()* because it is needed to pass the 44 observations of Table 3.1.

- *AgentAction()* is executed every time a *RequestDecision()* is called, just after *CollectObservations()*. Here the rewards of Table 3.3 are specified by using the *SetReward()* as many times as needed and passing it a float between -1 and 1. Another important function called here is *Done()* which resets the *agent* for the next simulation, by using automatically *AgentReset()*. *TAJ* calls *Done()* every time a game is over (when *agent* win or loses).

- *AgentReset()* is executed whenever the *agent* is done. *TAJ* resets here all the variables needed to calculate *rewards* (life, spirits, cleans totems inboard, resets charge, etc). In *TAJ*, once the *agent* is reset, the boolean that *AcademyStep()* checks is set to true, which will produce the *academy* to reset itself.

3.4.3 Training a model

The *TensorFlowSharp Unity plugin* is needed for *Unity* to be able to load generated models and get comparatives between them. To accomplish this installation, its zip package has to be downloaded from the installation docs of the *Unity ML Agents* repository [4]. Its content has to be pasted into the *Unity* project files.

Some other requirements and considerations to make life easier for training a *Unity Agent* are:

- Inside *Unity* game engine, go to Edit inside the toolbar, Project Settings, *Player Settings*.
  - In the *Resolution and Presentations tab*, leave *Run In Background* checkbox enabled to allow the game executable to be running although its window is not active. This is completely useful while training, so computer can be used while it keeps working in background.

  - In the *Resolution and Presentations* tab, leave *Disabled* the *Display Resolution Dialog* to avoid the executable to pop up the resolution dialog before launching the game. It prevents errors when connecting *Tensorflow* with the game environment since avoids requiring user *actions* and the training can start automatically.

- Keep in mind setting the *Brain Type* variable to *External* before building the game, just to let the *brain* learn through *Tensorflow*.

- Remember to set the *Brain Type* to *Internal* and pass the Graph Model (*.bytes* file) to let the *brain* "play" with the reached knowledge.

- Leave *Development Build* checkbox enabled before building the game, since *Unity* recommends it.

- Open *Anaconda* Prompt always as Administrator.

The process is:

1. Building the game inside the *.\unity_project \python* folder and remembering the exact name given to the executable.

2. Editing the *trainer_config.yaml* file located in the *.\unity_project \python* folder which allows customizing the trainer configuration for each *brain*. The most important thing to modify is the *max_steps* variable, with the purpose of setting the number of *steps* the training will have. The *.bytes* file (trained graph model) is saved once the *max_steps* count is met. Other configuration hyperparameters are commented in Section 3.4.4.

3. Opening an *Anaconda* Prompt (as Administrator) and executing the training with *Python python/learn.py <env_file_path> –train –run-id=<run_id_name>*. The parameter *<env_file_path>* will be the name of the exact executable name (without its extension). With the property *–load*, a previously calculated model can be loaded to start from its status (to continue teaching it). Models are saved in the folder *.\unity_project \python\models\<run_id_name>*, so using *–load* and passing a *<run_id_name>* to *–run-id* will try to load the previous model inside the *models* folder with this exact *<run_id_name>*. Passing *<run_id_name>* is also important to be able to see the progress of the training in *Tensorboard*, so an identifier should be always passed here.

4. Executing *tensorboard –logdir=summaries* to see the progress of learning. Its information will be now available from a web browser by default by accessing *localhost:6006*. Graphs in this page are described in Section 3.4.5.

5. Once the training ends and the *.bytes* file is generated, go to the *Unity* editor. Now set the *Brain Type* to *Internal* and pass the *.bytes* file to the *Graph Model* variable to let the *brain* "play" with the reached knowledge.

A *TAJ* training with 50,000 *steps* takes around 40 minutes to finish by using the *NVIDIA GTX 770 4GB GDDR5*.

A boolean constant *IS_TRAINING* has been entered in *TAJ* just to avoid any required user input for the training to be correctly executed (it also disables music and sounds). Another constant called *AVOID_RENDERERS* controls if the *Mesh Renderer* components have to be disabled, to speed up a little bit the training process and reduce the GPU load by avoiding to draw scene 3D models.

The final *Agents* obtained were trained (*external* mode) with 50,000 and 200,000 *steps*, respectively. All of them were trained against the random NPC, which gave a nice result.

Some previous training tests were made incrementally. So, an *agent* was first trained around 100,000 steps against the random NPC, this same *agent* was then trained against its own generated model, and so on.

Another test was made by training new *Agents* against old and different ones which were taking *actions* quite well but, due to the little mistakes they had in the code (with *observations* and *rewards*), they were inconsistent. To train *Agents* against previously generated models has returned an unsatisfactory result, while *Agents* only generated against the random NPC have a vastly wider victory percentage. The main reason an *agent* has been trained against previous ones was to provide it more different situations because they should have learned something and they should be able to apply some kind of logic in their decisions. As just mentioned, result is unsuccessful and the only thing that was achieved was limiting trained *agent* observations to the ones these have learned so far.

From the project experience, when *agent actions* depend on the enemy's ones (as in TAJ), a previously generated model is not useful to train future improved Agents. The better way is to train always against a random NPC. Bad results have been obtained this way.

Using several *agents* with the same *brain* would have been a nice option to accelerate the learning process. The way of using several *agents* is doing a lot of battles at the same time, one per *agent*. Every *agent* would learn from its NPC enemy, but a lot more *decisions* would be made in the same period of time.

There is a training method called *Imitation learning* [70] which consists of letting the *agent* learn from a human while he plays. This method has not been tried out in *TAJ*. A nice approach

could be making the first training sessions with Imitation learning and the following incremental ones, in External mode (random actions).

3.4.4 Understanding the *trainer_config.yaml* file

The configuration file is responsible for specifying the parameters (hyperparameters), the training method and other values to be used during training.

By default, the values of the *Default* section of the *trainer_config.yaml* are taken, but for each *brain*, these values can be specified to adapt the way the *agent* learns.

The main configuration parameters for training sessions with *PPO* [49] can be seen below:

- *Gamma* indicates if the *agent* has to act depending more on distant future rewards (larger value) or depending on more immediate rewards (smaller value). The typical range of values are between 0.8 and 0.995. The default value is 0.99 and 0.85 has been used in *TAJ* because this game requires *actions* to be done depending a lot on the current observations.

- *Lambda* is the same *lambda* used when calculating the *Generalized Advantage Estimate (GAE)* [71]. *GAE* first objective is reducing the variance of policy gradient estimates at the cost of some bias, to better manage the incoming data. Low values make the *agent* rely more on the current value estimate (higher bias), and higher to rely more on the actual *rewards* received (high variance). The typical range is between 0.9 and 0.95. The default value is 0.95 and 0.94 was appropriate for this project.

- *Buffer Size indicates how many experiences (observation, action, rewards loop) should be collected before updating the model. Larger buffer size corresponds to more stable training updates. The typical range is between 2,048 and 409,600. The default value is 10,240, correct for TAJ since a battle can easily fit into these number of steps.*

- *{Batch Size is the number of experiences used for one iteration of a gradient descent update, which really tries to minimize the error of the predictions [72]. The typical range for continuous action space is between 512 and 5,120, for discrete action space is between 32 and 512. The default value is 1,024, used in TAJ because training gave nice results, although Unity recommends 512 as the highest.*

- *Number of Epochs is the number of passes through the experience buffer during gradient descent. The typical range is between 3 (more stable updates) and 10 (faster learning). The default value is 3 and it worked fine in this project.*

- *Learning Rate is the strength of each gradient descent. It should be decreased if training is unstable and the reward does not consistently increase. The typical range is between 0.000001 and 0.0003. The default value is 0.0003 and 0.0005 has been used.*
- *Time Horizon* is the number of *steps* to collect before adding them to the experience buffer. The typical range is between 32 (in the case the *agent* receives frequent rewards or the project has extremely large episodes) and 2,048 (or larger, to make sure all agent's *actions* in an experience are collected). The default is 64 and 128 was used.
- *Max Steps* indicates how many *steps* the current training process has to do. It has to be increased when loading previous models from which it is wanted to continue learning. The typical range is between 500,000 and 10,000,000 (more complex problems). The default is 50,000. The values 50,000 and 200,000 were used.
- *Beta* has the objective of regularizing the *entropy*, which ensures the *agent* properly explore all the possible *actions* during training. *Entropy* should slowly decrease. The typical range is between 0.0001 and 0.01. *Beta* should be increased if *entropy* drops too quickly and decreased otherwise. The default is 0.005 and 0.0001 was used.
- *Epsilon* indicates the acceptable difference threshold between old and new policies during gradient descent updating. The typical range is between 0.1 (more stable updates) and 0.3 (faster training). The default is 0.2 and it was used.
- *Normalize* is a boolean which indicates whether normalization is applied to the vector observation inputs. Is useful for complex continuous observations problems and, maybe, inadequate for discrete problems. The default is false and it was used.
- *Number of Layers* corresponds to how many *hidden layers* are present after the observation input. More layers may be necessary for complex control problems. The typical range values are between 1 and 3. The default is 2 and it was used.
- *Hidden Units* correspond to how many units are in each fully connected layer of the neural network. For problems where the correct *action* is a simple combination of the *observations*, this should be small. The typical range is between 32 and 512. The default is 128 and 64 was used.

Lots of tests are needed to find values which return a nice *agent* behavior result. There is not a rule which determines exact values for each project and a lot of tests will be needed in most cases.

3.4.5 Understanding *Tensorboard* summaries

What *Tensorboard* does is to read the folders and files inside .*unity_project* *python_summaries* that *Tensorflow* writes. Eight plots are displayed:

- *Lesson Plot* shows the progress from lesson to lesson in the case *curriculum training* is used (a way of training an *agent* by increasing the hardness of the challenges little by little, for example in a game with different environments and tasks). This plot is not relevant for *TAJ* since the challenge of the *agent* is always the same (win the battle) and *curriculum training* has not been used. As Figure 3.5(a) shows, only one lesson has been given in that training.

- *Cumulative Reward* shows the *mean cumulative reward* for all *agents* involved in the training. It should increase during a successful training session because the objective is to make the *agent* find the highest reward possible. Figure 3.5(b) shows a bit of up and downs, but the tendency is to increase the value.

- *Entropy* talks about the randomness of the model decisions. In other words, the higher the entropy, the harder to draw any conclusions for the given information. So, a nice behavior for a successful training is to see the *entropy* slowly decreasing, because conclusions have to be more accurate during the training. In Figure 3.5(c) can be seen that the *TAJ* training slowly decreases its *entropy*.

- *Episode Length* just reports the mean length of each episode in the environment for all agents. It is related to the resets of the Academy and Agents, in the case of TAJ, made both at the same time manually. In Figure 3.5(d) shows *TAJ* result.

- *Learning Rate* should decrease over time. It notifies how large a step the training algorithm takes as it searches for the optimal policy. At the beginning of each training, the *external brain* tries to do random *actions* so it learns from new statuses. As the training progresses, it tries to choose the best *action* and it makes learning rate decrease. See Figure 3.5(e).

- *Policy Loss* values will oscillate while training. It refers to the changes in the process of deciding an action. Figure 3.5(f) shows one obtained result.

- *Value Estimate* should increase during a successful training since it shows the mean value estimate for all states already visited. Figure 3.5(g) shows values increasing as expected.

- *Value Loss* should decrease during a successful training (when reward becomes stable). This plot tells how good the model is to predict the value for each state. Figure 3.5(h) shows values decreasing as expected.

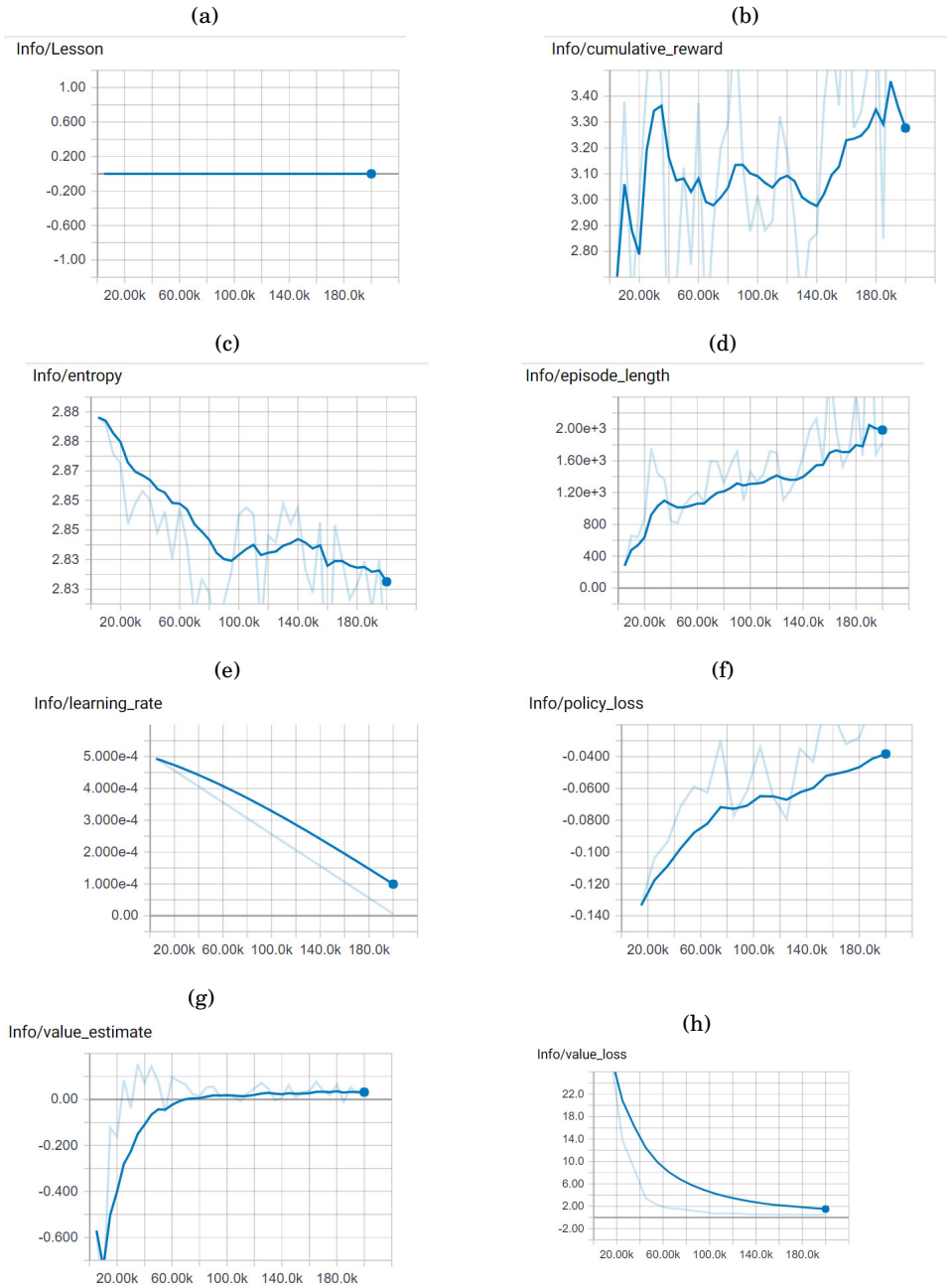

FIGURE 3.5. (a) *Lesson Plot* (b) *Cumulative reward* (c) *Entropy* (d) *Episode length* (e) *Learning rate* (f) *Policy loss* (g) *Value estimate* (h) *Value loss*

Getting plots outside this guideline say the generated model graph has not consistently trained.

3.5 Levels of difficulty

50,000 *steps* have been required to get an *agent* which really knows how to play. This one corresponds to the level "MEDIUM". There are other two levels which correspond to the random not trained NPC ("EASY") and the 200,000 *steps* model graph ("HARD"). They can be selected by using the *faction and difficulty selector* (see Figure A.3 in Appendix A).

The way the different levels are managed in this project is having different *agent GameObjects* disabled in the scene and, depending on the user selection, the corresponding ones are enabled. By using the *agent GiveBrain()* function, the corresponding *brain* can be set (which has to include its model graph predefined) but it was easier to make it by enabling and disabling previously configured *GameObjects*.

Not only the *brain* changes between difficulties, but also the time between decisions. To create a more realistic feeling, the frequency of the coroutine is not constant. The seconds before every new call of the coroutine are calculated randomly within a pre-established range. This range changes depending on the difficulty chosen when a trained model is used (while training every NPC and *agent* have the same decision frequency) :

- EASY, Random NPC. The range of seconds to wait for new decision requests is between 1.5 and 2.5 seconds. This makes *agent* slow and it will be easier for the player to win.
- MEDIUM, 50,000 *Steps agent*. The range of seconds to wait for new *decision* requests is between 1 and 2 seconds.
- HARD, 200,000 *Steps agent*. The range of seconds to wait for new *decision* requests is between 0.5 and 2 seconds. Not only will make better *actions*, but also will make them faster.

This decision frequency could also have been specified by disabling OnDemandDecisions check of the *agent* and overriding the default Decision class. Remember that disabling it makes the *agent* do an *action* every *step* and request *decision* every *DecisionFrecuency steps*, that's why overriding the *Decision* class would be needed. Doing it with coroutines also allows managing the randomness of time between decision requests without editing the Decision class.

So, decision time-frequency (however it is managed) is also important for getting differentiated levels of difficulty.

3.6 Writing the report

In the beginning, *Google Drive* was the tool used to write the *Technical Proposal* and part of the *Game Design Document*. Since then, *LATEX* has been used through the *Overleaf* online tool, with the main purpose of giving a more formal and correct design to the document.

It was a little tough to start with *LATEX* but, after getting used to it, working with it is really quicker than other tools because every design relative detail is determined by a template and that allows the user to only focus on the content of the document.

3.7 Version control

The repository used for saving every modification made into the *Unity* project (scripts, images, 3D models, *Unity* component values, etc) is *Collaborate* [73], by *Unity*.

This tool is totally integrated into the *Unity* game engine, so it takes care of keeping a history of changes enabling restoring old versions without losing any data.

The latest version of the project has been uploaded to *GitHub* [74] for allowing public access since *Unity Collaborate* repository needs to be linked with a *Unity* account.

Chapter 4

Results

Results have been measured by facing up several times different brains with *TensorFlowSharp* and getting the mean percentage of victories of each one. Table 4.1 show these clashes. To make this process faster, both *brains* has been set as *Internal*, its *.bytes* model has been passed and *delta-time* has been increased.

Tensorflow allows to execute itself without training the model, in *inference* mode, if *–train* variable is not passed to it. The problem is that only one generated model can be loaded (with *–load* and passing the corresponding –run-id). Since is needed to compare two models each time, it is a better option to use *TensorflowSharp*.

Table 4.1: Facing the *Brains*. Percentages of victories.

Facing the *Brains*			
-	**Random**	**50,000 *Steps***	**200,000 *Steps***
Random	-	4% - 96%	0% - 100%
50,000 *Steps*	96% - 4%	-	32% - 68%
200,000 *Steps*	100% - 0%	68% - 32%	-
Facing the *Brains*			

Clearly, the Table 4.1 shows that both *agents* have learned how to win the random NPC. 50,000 and 200,000 *agents* have trained well and their results are logic because they always have better victory percentages with less trained NPCs. So, the more the *agent* trains, the better the results. However, it should be considered that training too much can produce overfitting and, therefore, models with unwanted behaviors. A lot of training data does not directly mean

overfitting is going to happen, but as the data this project gives to the neural network comes from random actions, sometimes it cannot be that relevant and scattered as wanted. Overfitting appears when too much data information is inserted into a small region (and it is not in a wide range of values), so the neural network is too much precise with known data and it is bad at wandering best actions for a generalized problem (new data). In this project, an *agent* was trained with 1.500.000 *steps* and it constantly repeated same actions (it inserted totems in the same path). So, overfitting can lead the *agent* to exploit too much the best-known strategy and to avoid looking for better options.

Playing against the *agents* is another way of checking they have learned as expected. The experience changes depending on the tester (human player), but the result is that trained *agents* are hard to defeat.

One detail to highlight is that the actions taken by the *agents* at the beginning of the battles are always very similar because they always perform the best action given an observation. So, as observations of the first seconds of battles will be quite similar in all cases, they would be similar, but after that, every battle will be different and the *agent* will be more unpredictable.

4.1 Gameplay and executable

Two video gameplays of *TAJ* are published in Youtube [75].

The gameplay with comments [76] about the behavior of the *Medium* NPC can be found in this link: `https://www.youtube.com/watch?v=tzLsVMB_9tI`

The gameplay without comments [77] can be seen in the following url: `https://www.youtube.com/watch?v=ZgA-_ekphI8`

The executable [78] can be downloaded from the following url: `https://drive.google.com/file/d/1x3s695mTPUXNVycYMEdHF9AqzFCUomkZ/view?usp=sharing`

4.2 Mistakes setting up *observations* and *rewards*

Section 3.4.1.1 incorporates information related to the *TAJ* video game, but is important being aware that all of them have changed a lot from the first draft. This work requires a lot of try and error tests. In the *Unity's ML Agents beginners guide* [46] the next sentence can be found.

"This isn't about compile-time errors or whether you are a good programmer or not, the agents will find a way of exploiting your algorithm, so make sure that there is no logic gap.", Alessia Nigretti, 11 December 2017 [46]

Maybe *observations* are the easiest part of machine learning in a game like *TAJ* because they simply have to represent the game status in the current moment. Anyway, after first trainings, the *agent* was not learning properly and it has been tried to modify them. The way to edit them was to use fewer *observations* (from 44 to 28), trying to avoid passing not relevant information. Simplifying or unifying these *observations* was not a nice option as they were already quite simplified and they represented the game environment quite well. After all, they were reverted to 44 again, because this was not the problem why the *agent* did not learn. So, the main problem with observations has not been which ones to consider, but testing them and making sure they were being passed correctly with a bit of debugging.

Mistakes made with *actions* were to treat them as *continuous* (instead of *discrete*). Another fault was to consider two actions instead of one. At the beginning, these actions were understood as the human plays. A human will first click the *action* to make (action 1, to select the totem or power-up to use) and, afterwards, he/she will click the gap where this *action* has to be carried out (action 2, where action 1 has to be carried out). So, the first approach was using two *actions*. Once the machine takes a decision it needed to have a right combination of action 1 and action 2 to do something useful. It would have been surprising to see the machine coordinating itself to select an available totem/power-up in the desired gap/path. After fighting a bit with the code and making some tests, the best thing was to make things easier for the machine by turning *continuous actions* into *discrete* and unifying the two *actions* into one (see 3.2).

Mistakes made with *rewards* were to penalize too much the *agent*, and to reward it for constant acts with less relevance than initially could seem. For example, rewarding the *agent* for getting a *spirit* or killing an enemy have no direct effect on the main objective which is to win the battle. Another mistake later noticed was setting this constant *rewards* too high (although it seemed low, but repeated a lot of times they get bigger and bigger). The key was reducing a lot this constant rewards the way the *agent* will never find a hole to get *rewards* with unwanted actions. In this case, "1" was assigned to win the battle, and "0.2" every time a totem was inserted. It did not care to win the battle because just inserting 4 totems gave it the same reward as winning a battle. After transforming this "0.2" to a maximum of "0.1", and only rewarding for inserting totems in the appropriate situations, the change was drastic.

About resetting the *academy* (environment), another failure was not resetting it after each battle. The *agent* learned quite well although only the *agent* was reset, but not the environment. Having *Academy Max Steps* to 0 and not resetting manually the *Academy* produced an infinite episode, and it is a much better practice to reset *Academy*. So, for this project, resetting the environment after the *agent* (as seen in Section 3.4.2) improved the training results.

A little mistake that made me spend some more time was disabling *On Demand Decisions*. This configuration makes the *agent* to do an action every step and a decision every *DecisionFre-*

cuency, so, this way, actions were made that fast that made the game unplayable.

Some models have been generated with 1,500,000 *steps* and the result was not good because of overfitting. Depending on the project, the ideal number of steps needed will change.

Main problems about learning have been getting the right hyperparameters in the *trainer_config.yaml* file.

4.3 Unexpected problems

Setting up the environment gave some problems about permissions (remember executing programs with the right permissions) and about versions. Version 8 of *CUDA* was installed and an update to version 9 had to be done because of incompatibility with *CUDA* 8.

Another incompatibility problem was found further in the progress, just when passing into *Unity* the fist generated *.bytes* file which includes the model graph. It was necessary to downgrade *Tensorflow* from 1.7 to version 1.4 (compatible with *Unity*).

The method of training the *Unity* build with *Tensorflow* changed the 15th of March of 2018 because of a major update to version 0.3. Early versions used *Jupyter notebooks* to guide the process and allow stopping and generating the model from any *step*. Current versions (0.3 and newer) use the *learn.py Python* script and getting used to this new way of executing trainings took a bit of time. A little problem that appeared was that *Ctrl+C*, which should stop the training and auto-generate the *.bytes* model, only stops the training but does not generate the *.bytes* file in this current version. After some research, it has been noticed that modifying *max_steps* hyperparameter in *training_config.yaml* is the answer (although from this moment, trainings need to wait for the *max_steps* to be completed and to get a *.bytes* model). The positive thing in the new version 0.3 of *ML Agents* is the option of making decisions *On demand*, which has been required in *TAJ*.

These problems with versions made search alternative solutions to *ML Agents*. A little project with the library of an anonymous youtuber [79] has been trained. Then, move back to *ML Agents* was the taken decision, because of the efficiency and the facilities to get trained models *Tensorflow* gives.

4.4 Project files

Unity files, class diagram, art images and 3D models, music and so on can be found in the project repository [74] `https://github.com/al291708/TFG_al291708`.

4.5 Sections not developed

In terms of art, any animation has been done because the project has been much more focused to get an optimal neural network. 3D Models have been created to give the game a beautiful appearance but other animations (apart from the one that makes totems float and the one that makes the water move) has not been done.

Another aspect not fully developed is the 2 player's mode. Since the game is prepared to play by clicking buttons of both teams on screen with the mouse, it is not the best way to make to humans play in the same pc with the same mouse (or by using the keyboard). Tower defense games are more directed to have online or NPC opponents. The ranking of winners has not been developed either.

4.6 Dedicated hours

Table 4.2 shows the time this project has required. Initial planning was quite accurate, although there are certain aspects that required more time than previously thought (art, where hours have been exceeded and animations have not been done) and others that require less (video game programming). The Google Spreadsheet [80] with the tasks and times breakdown can be found in the following url [81]: `https://docs.google.com/spreadsheets/d/1qqb8tHH7cPAOKcuFzqvrL8TuKzHOcIwq4Q5mlOyySpO/edit?usp=sharing`.

Table 4.2: Planning of the Final Degree Project.

Dedicated hours		
Dedicated(hours)	**Estimated(hours)**	**Task**
26.75	20	Neural networks algorithm's research
27	20	Modeling and animating video game components
66.5	80	Video game programming (gameplay, mechanics, HUD, insert 3D models and animations)
57	50	Adding neural networks into the video game
26.5	30	Creation of the analysis and design document
74	70	Report of the project
16.25	30	Prepare presentation
5	10	Creation of videos about the project and the gameplay
TOTAL **3**00		
End of Dedicated Hours		

CHAPTER 5

CONCLUSIONS

The purpose of the project was creating a Tower Defense video game and integrating Neural Networks in it. The main objectives have been met:

- Creating a tower defense video game. This objective has been achieved since the game is entertaining and fits into the Game Design Document (Section 2) requirements.
- Integrating neural networks to a tower defense video game. *ML Agents Unity* plugin has been used to achieve this goal, and 2 game difficulties have been created by using reinforcement learning.
- Create a hard NPC by using neural networks. The *agents* trained with neural networks are hard to defeat, they behave correctly and they are competitive. The method presented in this project shows a way to get different difficulty levels without hardcoded behaviors, as rubber banding (commented in Section 1.1) does, making less evident the manipulation of the difficulty in the attempt to keep users desire to keep playing.
- Giving the game a nice looking appearance. 3D models created with *Blender*, sprites made with *Photoshop*, animations with *Unity Particle Systems* and camera movements offer a great aspect to the video game.
- Learning about reinforcement learning. This project has allowed me to better understand how reinforcement learning works and to face problems that appear in the process.

Making an *agent* learn in a quite wide game, with several different actions, has been a challenge. About the time dedicated to the project, neural networks require a lot of tests and perseverance and it has been sometimes frustrating to see that, despite making many changes, poor results continued to be obtained. However, the time devoted to the project has been adjusted

to the initial planning.

One of the main aspects demonstrated during this project is that the best way to train an *agent*, at least in a game as the *TAJ* Tower Defense, is to face it up against an NPC which makes it see as many observations as possible. Testing against previous models is not a good idea because lots of possible new observations can be discarded.

This project has talked about training an *agent*, getting a model and keeping it as it is once it plays in inference mode. Training the *agent* against a random NPC has the limitation that its actions are made with a logic not comparable to the one of a human. As an improvement proposal, it would be nice to make the best trained model learn from each game it plays against humans (with Imitation Learning), so it becomes invincible because all observations and studied strategies of experimented players would be given to the *agent*.

Playing against an invincible NPC is not challenging for any human player, so another field which can be explored, after having an *agent* which always win, is to adapt the *agent* difficulty during the game, depending on the way the human plays. This would ease the human how it learns to play the video game, it would avoid players to leave because of a high initial hardness and would allow to adapt online games to each player level or make them win/lose only when the game wants to keep their desire to play.

In terms of game, leaving apart neural networks, an online or 2 players mode would be interesting to launch the game and making it more addictive.

Summarizing, this project has been a nice experience for me since it has taught me many things about neural networks and reinforcement learning, the correct way to layout a document and the importance of setting goals and meeting them within the deadline.

APPENDIX A

ART APPENDIX

This appendix shows inspiration art, captures during the gameplay, sketches of characters, and particle animations of TAJ.

FIGURE A.1. TAJ logo.

FIGURE A.2. Intro menu screenshot.

FIGURE A.3. Screenshot of faction and level difficulty selector.

FIGURE A.4. (a) *Kaapo* of *Nerta* faction. (b) *Kaapo* of *Ittla* faction. (c) *Rad* of *Nerta* faction. (d) *Rad* of *Ittla* faction. (e) *Vasu* of *Nerta* faction. (f) *Vasu* of *Ittla* faction.

(a) (b)

FIGURE A.5. (a) Donkey Kong Country Returns Bosses [82]. (b) Nastia Polska Magic Stone [83].

FIGURE A.6. *Vasu* (*Nerta* faction), *Kaapo* (*Nerta* faction) and *Rad* (*Ittla* faction) models (left to right).

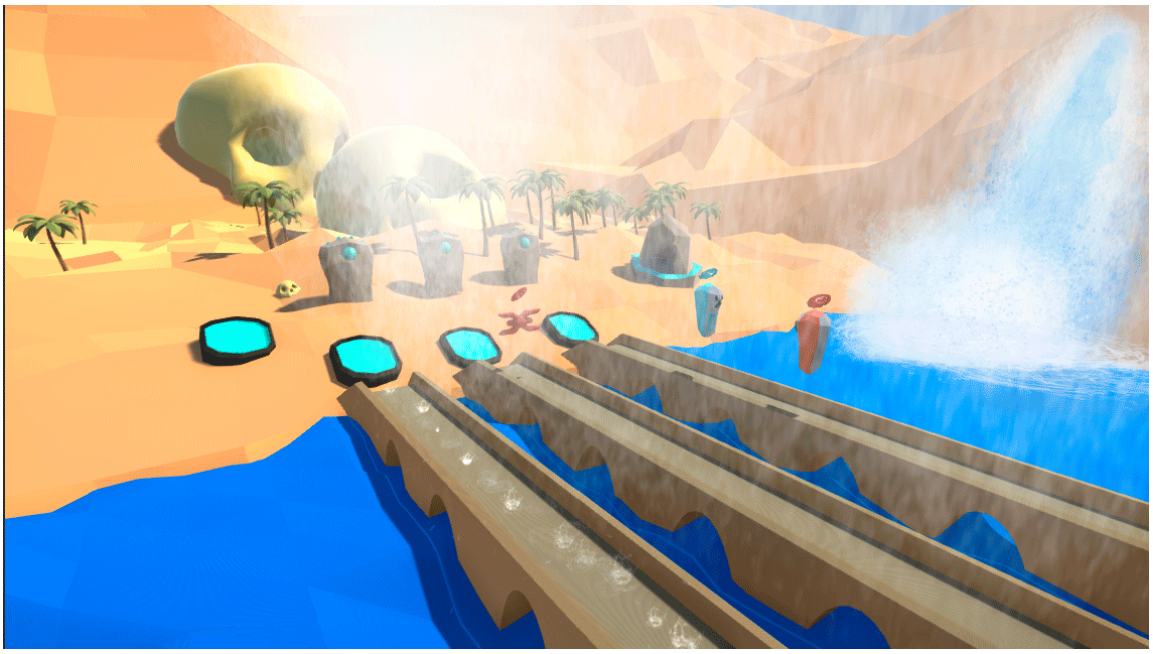

FIGURE A.7. Faith Force Power-up.

FIGURE A.8. Ice Force Power-up.

FIGURE A.9. Sun Force Power-up.

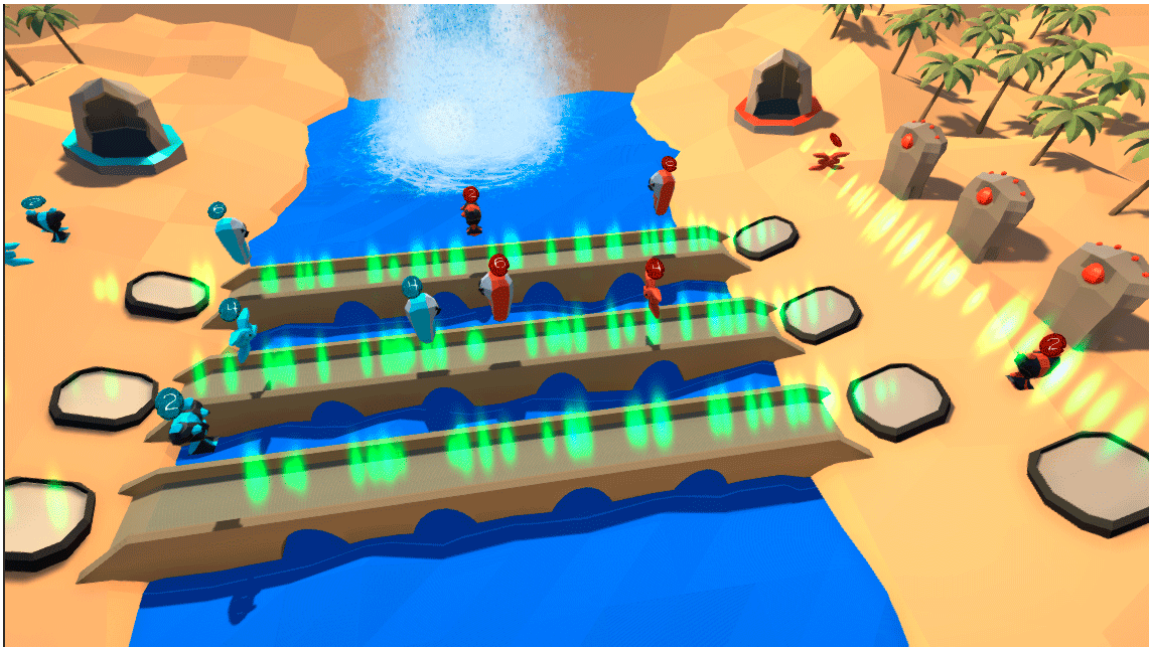

FIGURE A.10. Wind Force Power-up.

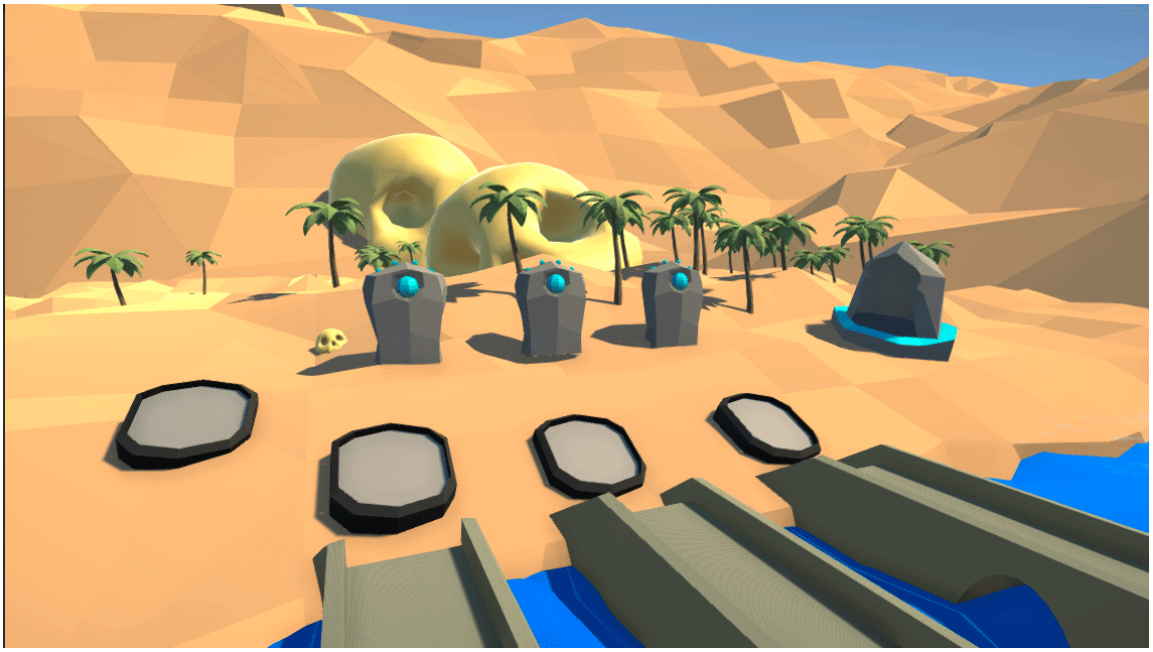

FIGURE A.11. Ittla Power Stones.

FIGURE A.12. Nerta Power Stones.

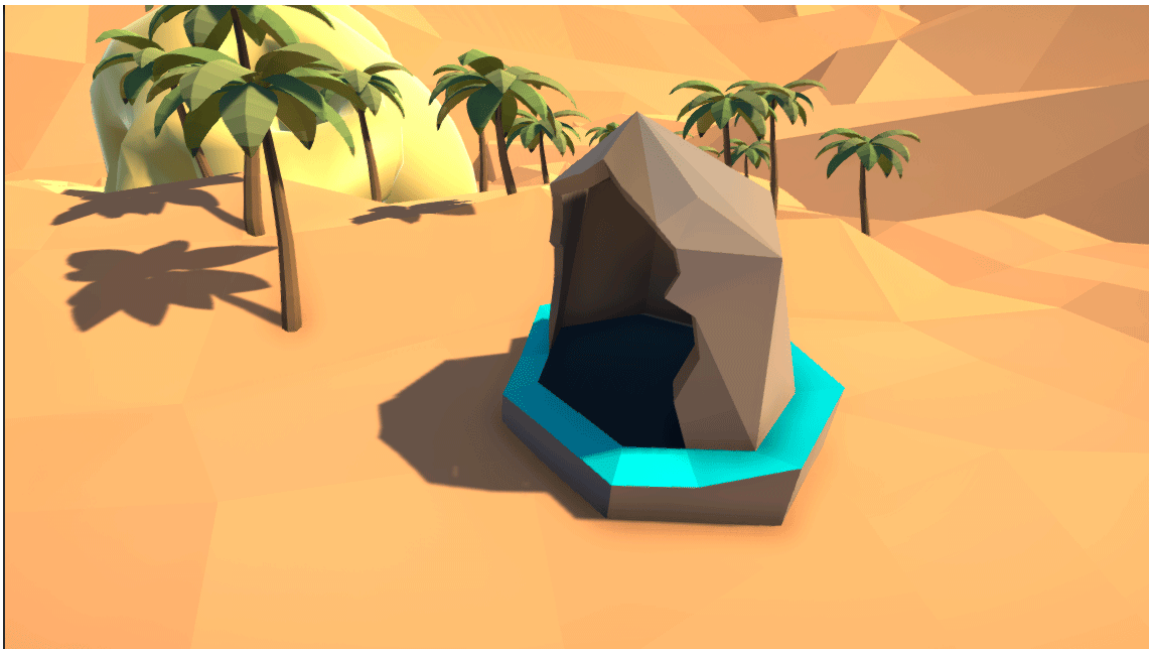

FIGURE A.13. Ittla Shrine.

Bibliography

[1] UNIVERSITAT JAUME I.
Diseño y Desarrollo de Videojuegos.
https://www.uji.es/estudis/oferta/base/graus/actual/videojocs/?urlRedirect=https://www.uji.es/estudis/oferta/base/graus/actual/videojocs/&url=/estudis/oferta/base/graus/actual/videojocs/, 2018.
Online; accessed 28 May 2018.

[2] WIKIPEDIA.
Tower Defense.
https://en.wikipedia.org/wiki/Tower_defense, 2018.
Online; accessed 28 May 2018.

[3] WIKIPEDIA.
Artificial Neural Network.
https://en.wikipedia.org/wiki/Artificial_neural_network, 2018.
Online; accessed 28 May 2018.

[4] UNITY TECHNOLOGIES.
Unity Machine Learning Agents.
https://github.com/Unity-Technologies/ml-agents, 2017.
Online; accessed 2 April 2018.

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov.
Proximal Policy Optimization Algorithms.
https://arxiv.org/abs/1707.06347, 2017.
Online; accessed 25 April 2018.

[6] WIKIPEDIA.
Reinforcement Learning.
https://en.wikipedia.org/wiki/Reinforcement_learning, 2018.
Online; accessed 28 May 2018.

[7] WIKIPEDIA.

Machine Learning.
https://en.wikipedia.org/wiki/Machine_learning, 2018.
Online; accessed 28 May 2018.

[8] WIKIPEDIA.
5 Games Totally Ruined By Rubberbanding AI.
https://en.wikipedia.org/wiki/Dynamic_game_difficulty_balancing, 2016.
Online; accessed 28 May 2018.

[9] ADRIÁN GONZÁLEZ RAMÍREZ.
Personal Website.
http://www.adriangonzalez.com.es/, 2017.
Online; accessed 28 May 2018.

[10] UNITY TECHNOLOGIES.
Unity 3D Website.
https://unity3d.com/es, 2018.
Online; accessed 28 May 2018.

[11] GOOGLE PATENTS.
US7278913B2 - US Grant - Racing game program and video game device.
https://patents.google.com/patent/US7278913, 2004.
Online; accessed 30 May 2018.

[12] WIKIPEDIA.
Mario Kart: Double Dash!!
https://en.wikipedia.org/wiki/Mario_Kart:_Double_Dash, 2004.
Online; accessed 30 May 2018.

[13] PASTEMAGAZINE.
5 Games Totally Ruined By Rubberbanding AI.
https://www.pastemagazine.com/articles/2016/10/5-games-totally-ruined-by-rubberbanding-ai.html, 2016.
Online; accessed 30 May 2018.

[14] EUROGAMER - WESLEY YIN-POOLED.
EA says no, your FIFA matches aren't rigged.
https://www.eurogamer.net/articles/2017-06-20-ea-says-no-your-fifa-matches-arent-rigged, 2017.
Online; accessed 30 May 2018.

[15] WIKIPEDIA.

FIFA 18.
https://en.wikipedia.org/wiki/FIFA_18, 2017.
Online; accessed 30 May 2018.

[16] WIKIPEDIA.
Need for Speed (2015 video game).
https://en.wikipedia.org/wiki/Need_for_Speed_(2015_video_game), 2004.
Online; accessed 30 May 2018.

[17] WIKIPEDIA.
Microsoft Windows.
https://en.wikipedia.org/wiki/Microsoft_Windows, 2018.
Online; accessed 30 May 2018.

[18] WIKIPEDIA.
Android (operating system).
https://en.wikipedia.org/wiki/Android_(operating_system), 2018.
Online; accessed 30 May 2018.

[19] WIKIPEDIA.
Windows 7.
https://en.wikipedia.org/wiki/Windows_7, 2018.
Online; accessed 30 May 2018.

[20] WIKIPEDIA.
OS X Mavericks.
https://en.wikipedia.org/wiki/OS_X_Mavericks, 2018.
Online; accessed 30 May 2018.

[21] WIKIPEDIA.
SSE2.
https://en.wikipedia.org/wiki/SSE2, 2018.
Online; accessed 30 May 2018.

[22] WIKIPEDIA.
Direct X.
https://es.wikipedia.org/wiki/DirectX, 2018.
Online; accessed 30 May 2018.

[23] GOOGLE.
Tensorflow.
https://www.tensorflow.org/, 2018.
Online; accessed 30 May 2018.

[24] ANACONDA, INC.
Download Anaconda Distribution.
https://www.anaconda.com/download/, 2017.
Online; accessed 2 April 2018.

[25] PYTHON.ORG.
Python.org Website.
https://www.python.org/, 2018.
Online; accessed 30 May 2018.

[26] ANACONDA INC.
Conda documentation.
https://conda.io/docs/, 2018.
Online; accessed 30 May 2018.

[27] PROJECT JUPYTER.
Jupyter Notebook Website.
http://jupyter.org/, 2018.
Online; accessed 30 May 2018.

[28] NVIDIA CORPORATION.
CUDA Toolkit Archived Releases.
https://developer.nvidia.com/cuda-toolkit-archive, 2017.
Online; accessed 2 April 2018.

[29] NVIDIA CORPORATION.
NVIDIA cuDNN GPU Accelerated Deep Learning.
https://developer.nvidia.com/cudnn, 2017.
Online; accessed 2 April 2018.

[30] BLENDER.
Blender Websie.
https://www.blender.org/, 2018.
Online; accessed 30 May 2018.

[31] ADOBE.
Adobe Photoshop.
https://www.adobe.com/es/products/photoshopfamily.html, 2018.
Online; accessed 30 May 2018.

[32] GOOGLE.
Google Drive Slides About.

https://www.google.es/intl/es/slides/about/, 2018.
Online; accessed 30 May 2018.

[33] MICROSOFT.
Visual Studio IDE.
https://www.visualstudio.com/, 2018.
Online; accessed 30 May 2018.

[34] WRITELATEX LIMITED.
Overleaf.
https://www.overleaf.com/, 2018.
Online; accessed 30 May 2018.

[35] UNITY ASSET STORE - ERVIS LILAJ.
Low-Poly Skull.
https://assetstore.unity.com/packages/3d/characters/low-poly-skull-111786, 2018.
Online; accessed 28 May 2018.

[36] UNITY ASSET STORE - DARTH_ARTISAN.
Free Trees.
https://assetstore.unity.com/packages/3d/vegetation/trees/free-trees-103208, 2018.
Online; accessed 28 May 2018.

[37] UNITY TECHNOLOGIES.
Unity Asset Store.
https://assetstore.unity.com/, 2018.
Online; accessed 28 May 2018.

[38] BLENDCRAFT SOLUTIONS.
Unity Tutorial - How to Make Low Poly Water.
https://www.youtube.com/watch?v=3MoHJtBnn2U, 2018.
Online; accessed 17 May 2018.

[39] FREE SOUND.
Huge collaborative database of audio snippets, samples, recordings, bleeps, etc.
https://freesound.org/, 2018.
Online; accessed 17 May 2018.

[40] YOUTUBE.
Audio Library.

https://www.youtube.com/audiolibrary/music, 2018.
Online; accessed 17 May 2018.

[41] UNITY ASSET STORE -UNITY TECHNOLOGIES.
Unity Particle Pack.
https://assetstore.unity.com/packages/essentials/asset-packs/unity-particle-pack-73777, 2018.
Online; accessed 28 May 2018.

[42] UNITY ASSET STORE -UNITY TECHNOLOGIES.
Unity Particle PackPost Processing Stack.
https://assetstore.unity.com/packages/essentials/post-processing-stack-83912, 2018.
Online; accessed 28 May 2018.

[43] WIKIPEDIA.
Fast approximate anti-aliasing.
https://en.wikipedia.org/wiki/Fast_approximate_anti-aliasing, 2018.
Online; accessed 28 May 2018.

[44] WIKIPEDIA.
Ambient Occlusion.
https://en.wikipedia.org/wiki/Ambient_occlusion, 2018.
Online; accessed 28 May 2018.

[45] WIKIPEDIA.
Artificial intelligence.
https://en.wikipedia.org/wiki/Artificial_intelligence, 2018.
Online; accessed 28 May 2018.

[46] ALESSIA NIGRETTI.
Using Machine Learning Agents in a real game: a beginner's guide.
https://blogs.unity3d.com/es/2017/12/11/using-machine-learning-agents-in-a-real-game-a-beginners-guide/, 2017.
Online; accessed 29 March 2018.

[47] WIKIPEDIA.
First Person Shooter.
https://en.wikipedia.org/wiki/First-person_shooter, 2018.
Online; accessed 28 May 2018.

[48] MICHAEL NIELSEN.

Neural Networks and Deep Learning.
http://neuralnetworksanddeeplearning.com/, 2017.
Online; accessed 03 May 2018.

[49] UNITY TECHNOLOGIES.
Training with Proximal Policy Optimization.
https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md, 2017.
Online; accessed 25 April 2018.

[50] WIKIPEDIA.
Tensor processing unit.
https://en.wikipedia.org/wiki/Tensor_processing_unit, 2018.
Online; accessed 30 May 2018.

[51] STACKOVERFLOW.
How to run Tensorflow on CPU.
https://stackoverflow.com/questions/37660312/how-to-run-tensorflow-on-cpu, 2018.
Online; accessed 30 May 2018.

[52] GOOGLE.
Tensorboard.
https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard, 2018.
Online; accessed 30 May 2018.

[53] .NET.
https://www.microsoft.com/net/, 2018.
Online; accessed 30 May 2018.

[54] UNITY TECHNOLOGIES.
GameObject.
https://docs.unity3d.com/ScriptReference/GameObject.html, 2018.
Online; accessed 30 May 2018.

[55] UNITY TECHNOLOGIES.
Canvas.
https://docs.unity3d.com/ScriptReference/Canvas.html, 2018.
Online; accessed 30 May 2018.

[56] UNITY TECHNOLOGIES.

Box Collider.
https://docs.unity3d.com/ScriptReference/BoxCollider.html, 2018.
Online; accessed 30 May 2018.

[57] WIKIPEDIA.
A* search algorithm.
https://en.wikipedia.org/wiki/A*_search_algorithm, 2018.
Online; accessed 30 May 2018.

[58] UNITY TECHNOLOGIES.
NavMesh.
https://docs.unity3d.com/ScriptReference/AI.NavMesh.html, 2018.
Online; accessed 30 May 2018.

[59] UNITY TECHNOLOGIES.
NavMeshAgent.
https://docs.unity3d.com/ScriptReference/AI.NavMeshAgent.html, 2018.
Online; accessed 30 May 2018.

[60] ADRIÁN GONZÁLEZ RAMÍREZ.
TAJ Class diagram.
https://drive.google.com/open?id=1UXq8Opy5zAJARaXroWeZIMcMFPYY61UJ, 2018.
Online; accessed 30 May 2018.

[61] UNITY TECHNOLOGIES - ARTHUR JULIANI.
Introducing: Unity Machine Learning Agents.
https://blogs.unity3d.com/es/2017/09/19/introducing-unity-machine-learning-agents/, 2017.
Online; accessed 30 May 2018.

[62] MICROSOFT.
Windows 10.
https://www.microsoft.com/es-es/windows/features, 2018.
Online; accessed 30 May 2018.

[63] JASON WEIMANN.
Unity3D Machine Learning – Setting up the environment & Tensorflow for AgentML on Windows 10.
https://unity3d.college/2017/10/25/machine-learning-in-unity3d-setting-up-the-environment-tensorflow-for-agentml-on-windows-10/, 2017.
Online; accessed 2 April 2018.

[64] UNITY 3D COLLEGE.
Unity3D Machine Learning Setup for ML-Agents on Windows 10 with Tensorflow.
https://www.youtube.com/watch?v=qxicgknzUG8, 2017.
Online; accessed 28 March 2018.

[65] WIKIPEDIA.
Integer.
https://en.wikipedia.org/wiki/Integer, 2018.
Online; accessed 30 May 2018.

[66] UNITY TECHNOLOGIES.
Making a New Learning Environment.
https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Create-New.md, 2018.
Online; accessed 25 April 2018.

[67] UNITY TECHNOLOGIES.
Getting Started with the 3D Balance Ball Environment.
https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Getting-Started-with-Balance-Ball.md, 2017.
Online; accessed 1 April 2018.

[68] UNITY TECHNOLOGIES.
Example Learning Environments.
https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md, 2017.
Online; accessed 29 March 2018.

[69] WIKIPEDIA.
Markov decision process.
https://en.wikipedia.org/wiki/Markov_decision_process, 2018.
Online; accessed 30 May 2018.

[70] UNITY TECHNOLOGIES.
Imitation learning.
https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Imitation-Learning.md, 2018.
Online; accessed 30 May 2018.

[71] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, Pieter Abbeel.
arXiv:1506.02438 High-Dimensional Continuous Control Using Generalized Advantage Estimation.

https://arxiv.org/abs/1506.02438, 2018.
Online; accessed 16 May 2018.

[72] WIKIPEDIA.
Gradient descent.
https://en.wikipedia.org/wiki/Gradient_descent, 2018.
Online; accessed 16 May 2018.

[73] UNITY TECHNOLOGIES.
Collaborate, Create Together Seamlessly.
https://unity3d.com/unity/features/collaborate, 2017.
Online; accessed 20 January 2018.

[74] ADRIÁN GONZÁLEZ RAMÍREZ.
Neural networks applied to a tower defense videogame.
https://github.com/al291708/TFG_al291708, 2018.
Online; accessed 17 May 2018.

[75] GOOGLE.
Youtube.
https://www.youtube.com/?hl=es&gl=ES, 2018.
Online; accessed 30 May 2018.

[76] ADRIÁN GONZÁLEZ RAMÍREZ.
TAJ Gameplay With Comments - Neural Networks Applied to a Tower Defense Videogame.
https://www.youtube.com/watch?v=tzLsVMB_9tI, 2018.
Online; accessed 3 June 2018.

[77] ADRIÁN GONZÁLEZ RAMÍREZ.
TAJ Gameplay Without Comments- Neural Networks Applied to a Tower Defense Videogame.
https://www.youtube.com/watch?v=ZgA-_ekphI8, 2018.
Online; accessed 3 June 2018.

[78] ADRIÁN GONZÁLEZ RAMÍREZ.
TAJ Executable.
https://drive.google.com/file/d/1x3s695mTPUXNVycYMEdHF9AqzFCUomkZ/view?usp=sharing, 2018.
Online; accessed 3 June 2018.

[79] THE ONE.
Evolving Neural Networks NEAT With 3D Cars + Tutorial (minute 16:15.

https://www.youtube.com/watch?v=X8ieBGMjtzM, 2017.
Online; accessed 16 March 2018.

[80] GOOGLE.
Google Drive Spreadsheets.
https://docs.google.com/spreadsheets/u/0/, 2018.
Online; accessed 30 May 2018.

[81] ADRIÁN GONZÁLEZ RAMÍREZ.
Dedicated time to the project (Google Drive Spreadsheet) .
https://docs.google.com/spreadsheets/d/1qqb8tHH7cPA0KcuFzqvrL8TuKzHOcIwq4Q5mlOyySp0/edit?usp=sharing, 2018.
Online; accessed 30 May 2018.

[82] 3D JUEGOS.
Donkey Kong Country Returns contará con la Súper Guía.
https://i11c.3djuegos.com/juegos/6209/donkey_kong_country_returns/fotos/maestras/donkey_kong_country_returns-1393828.jpg, 2010.
Online; accessed 25 February 2018.

[83] NASTIA POLSKA ARTSTATION.
Magic Stone.
https://cdnb.artstation.com/p/assets/images/images/001/679/533/large/nastia-polska-016sm.jpg?1450721321, 2015.
Online; accessed 25 February 2018.