CONTROLLING VIDEOGAME ELEMENTS USING PRE-TRAINED NEURAL NETWORKS IN UNITY.

Written and developed in 2020.

Student

Bermudo Bayo, Miguel

DNI: 29509441K

Tutors

Hernández Salmerón, Inmaculada

Borrego Diaz, Agustín

Department

Lenguajes y Sistemas Informáticos.

Dedicated to

- My girlfriend, for being the most supportive person I can have in my life and always making me inch forward even when times get hard.

- My family, for always supporting whatever it is that I want to pursue in life.

- My friends, for being brutally honest whenever I make something that I believe to be grandiose and in reality, it might be subpar (to put it lightly).

- The teachers that were there for me whenever I needed them to be.

- My classmates, for teaching me how to have ungodly amounts of patience whenever it is necessary.

To all these people I owe a great deal and I would not be where I am today if it weren't for them, to all of them from the bottom of my heart, thank you.

Special thanks

To my tutors, Inmaculada, for providing helpful insight and questioning the choices made to help me better the results of this project and Agustin, for always finding the time to help me with the multitude of difficulties found along the way and providing the infrastructure needed for the smooth deployment of the project in a different media that originally planned for due to the current situation we find ourselves to be going through.

Table of contents

Table of Figures

Table of Tables

Table of Graphs

SUMMARY

With this Project we aspire to answer to the need game developers have to develop AI for their intelligent Agents in a faster, less costly way.

we chose this project as it combines my passion for developing videogame with my curiosity for state-of-the-art technology.

We prove how intelligent agents can replace manually written AI and perform jobs with minimal supervision.

For this we will be using Proximal Policy Optimization algorithms and machine learning packages provided for our Game Engine of choice, unity in this case.

We used SCRUM methodology to guide our development and time scheduling needs. As well as several tools for development such as IDE, source control etc.

We've concluded in this study that these agents although intelligent are very costly, needing very strong hardware to cover its enormous computing needs, as well as the time taken to train them.

1. INTRODUCTION

First we'd like to start with a question; what is Artificial Intelligence? Well "*In video games, artificial intelligence (AI) is used to generate responsive, adaptive or intelligent behaviors primarily in non-player characters (NPCs) similar to human-like intelligence. Artificial intelligence has been an integral part of video games since their inception in the 1950[16]"*

"Game AI/heuristic algorithms are used in a wide variety of quite disparate fields inside a game. The most obvious is in the control of any NPCs in the game, although "scripting" (decision tree) is currently the most common means of control." [18]

"These handwritten decision trees often result in "artificial stupidity" such as repetitive behavior, loss of immersion, or abnormal behavior in situations the developers did not plan for" [19]

These quotes are pretty descriptive of the problem videogame AI faces nowadays, its either very straightforward which cause players to detect patterns and simplify them to the point of trivializing NPCs, or downright abuse them for unintended behaviors.

The repetitiveness that simple AI creates ins videogame affects replayability, once you've seen it once, you've seen it all. Even though this not exactly the case it is true that an enemy would respond similarly to external stimuli unless we introduce a random element in it, which is often done via a random number generator causing alterations in how AI behaves to keep games "fresh" a little longer.

AI agents aren't limited to videogames, they can also be applied to simulation of real life events to investigate possible solutions that the neural networks come with simulating real life conditions as closely as possible using unity's prebuilt physics engine, nonetheless it is not the recommended approach even thought this engine provides quite a good approximation of drag, gravity etc, it is not perfect and others physics engines should be used for this line of work.

You can take for example an existing AI that analyzes images from satellites to prevent things like landslides, floods, volcanic eruptions, and tsunamis. This is not the same type of agent you would use in a videogame as it doesn't need to predict movement or actions and react accordingly but instead it takes in imagery of events that happened already and tries to predict future events of similar characteristics.

There is also the possibility of training agents to determine the topographical viability of a plot that's going to be used for a big building. Japan is a great example of this as they need to check the terrain for possible seismic failures.

On the topic of video Games there are currently there are very little examples of them that rely on neural networks to create the logic for their agents. But there have been cases of external developers that created neural networks to play the games.

Some of these examples come of the hands of massive corporations like googles' **AlphaGo** a deep learning agent trained to play **GO** a traditional Chinese game that had 10^170 possible board states compared to the 10^120 states that chess has.

Figure 1: Traditional GO Table

Another example more fitted to the subject of this project is **Alphastar**, an AI developed to play blizzard entertainments classic 1 v 1 game **Starcraft**. this game is notorious for having one of the most dedicated playerbases in the world and for being extremely difficult to play at the top level, not only for the insane amounts of knowledge required to play it at that level but also for the mechanical skill required to perform actions that fast.

It was first developed to have an advantage over players by being able to see the whole map at all times, but to make it fair developers made it so it had the same resources as a normal player would. even then it was able to develop a totally new strategy never heard of before for a specific matchup in the game.

This shows the real potential of neural networks as intelligent agents to be used in place of manually designed AI. Furthermore, in the following statement we can see how it can even pose too much of a challenge for human players surpassing even the most dedicated of them.

"*The developers have not publicly released the code or architecture of their model, but have listed several state-of-the-art machine learning techniques such as relational deep reinforcement learning, long short-term memory, auto-regressive policy heads, pointer networks, and centralized value baseline.* ***Alphastar*** *was initially trained with supervised*

learning, it watched replays of many human games in order to learn basic strategies. It then trained against different versions of itself and was improved through reinforcement learning. The final version was hugely successful, but only trained to play on a specific map in a protoss mirror matchup." [6]

There is also the factor of money. how much was invested into these rather complex AI agents? how manageable is it to take this into general game development cycles? How much impact will it take overall into the budgets of a big project.

BUDGET CATEGORY	PERCENT OF GOAL	AMOUNT (IN USD)
DEVELOPMENT		
PROGRAMMING	18%	$27,000
ART	17%	$25,500
GAME DESIGN	13%	$19,500
MUSIC & SOUND	12%	$18,000
QUALITY ASSURANCE & TESTING	2%	$3,000
DEVELOPMENT SUBTOTAL	**62%**	**$93,000**
CAMPAIGN FEES & EXPENSES		
TAXES	20%	$30,000
KICKSTARTER FEES & PROCESSING	10%	$15,000
BACKER REWARDS	8%	$12,000
CAMPAIGN SUBTOTAL	**38%**	**$57,000**
TOTAL FUNDING GOAL	**100%**	**$150,000**

Figure 2: Example of project costs

Looking at this example from "The Last Goddess" a rather art intensive and low programming game we can see how programming is mostly always the top spender for a videogame. Since AI is normally the biggest part of programming costs we can assume reducing the time spent programming AI will reduce the total costs for this type of budgets.

There is also the fact that AI agents are very expensive to develop effectively resource wide and take a considerable amount of trial and error since you can't predict whats going to happen until you see it. For example the aforementioned alphastar and google's GO Zero costs were, $12,976,128 to replicate AlphaStar Final and $35 million to replicate the paper for GO. [19]

In conclusion this project will cover how the usage of pre-trained neural networks made for intelligent agents can result in shorter development times for certain situations such as repeating enemies that must have variable patterns, where we can train the agents in multiple lines of work and obtain different approaches. As well as how it can provide important insight on machine learning as a whole in the field of video games.

1.1 MOTIVATION

My decision was made fundamentally based on my love for developing videogames and my desire to eventually be able to live from this and for the fascination I've always had for the field of AI. This project combines both of them in an attempt to provide a useful approach leading to a faster development cycle and general usefulness in my future line of work and learning more about what I believe to be a very useful skill that is the field of neural network training and deep learning technology.

The alternative was simply developing a more complex and complete videogame without the added value that is working with non-pre-written AI like A* pathfinding algorithms or random attack patterns like most videogames do right now.

1.2 OBJECTIVES

The objectives are pretty clear, as they were referenced beforehand:

- We want to **analyze** the results of agent training and determine if it is a feasible way to create AI for video game element to further the speed of game development.

- We want to **systematize** the creation of training scenes and analyze its validity as a real alternative to the creation of manual AI state-machines

- We want to **analyze** the data produced by the trained agents versus real players and determine if their training was successful based on the results obtained.

- We want to **analyze** the data produced by the agent training to corroborate if the environment used to train the agent was successful at doing so.

- We want to **compare** different training environments to each other to determine which approach is the best.

Note that we do not want to **replace** the need for manually created agent patterns that give a unique and more controlled feeling to agents since that gives uniqueness and more of a singular feeling to a game.

To summarize, we want to develop training-ready scenes that would give us a multitude of neural networks for agents to use in-game, check if these networks give us a competitive enough AI to beat players consistently enough and determine if the training scenes and algorithms used were successful at doing so.

2. METHODOLOGY:

2.1. DOCUMENT STRUCTURE

Normalmente esta sección describe explicitamente la estructura del documento, es decir: “En el capítulo 1 se explica esto, en el capítulo 2 se explica esto otro, …, finalmente, en el capítulo tal se resumen las conclusiones y se finaliza la memoria”. De todas formas, deja esto para el final, cuando tengas una versión terminada de la memoria

The document will follow a complexity approach, going from simple themes into more complex and detailed ones. Starting with the justification of the project we explained why we’ve chosen to overtake this challenge and the previous experiences that lead us to it.

Next, we will talk about how we will overcome the challenges the methods that were used for organization how much they will cost us and indicators to exemplify if the objectives were reached on time or if they overcame us instead.

Then there is the topic of what this project does for the domain it is going to be developed for, why is it necessary and who can make use of it, also focusing on what detrimental or slow steps of the process of development it can help or substitute entirely.

Afterwards we will describe our scenes and how they were made, what purpose the components on the scene serve, how the code works, etc… In summary all of the necessary things that allow the reader to understand how all ties together.

2.2. CONVENTIONS

- Direct quotes taken from another source will be in cursive times new roman and in between quotation marks. they will also be accompanied by a number in brackets that indicates the reference, this can be found in the bibliography section; *“here is an example” [N]*

- Source code will maintain Visual Studio Codes’ format, this is for code visibility and easy importing.

```
using UnityEngine;
public class Example: MonoBehaviour
{
    void Example()
    {
        DoSomething();
    }
}
```

2.3. DEVELOPMENT CYCLES

The development cycle that was used in this project is determined by agile methodologies. Even though agile methodologies are mostly team oriented since this was a solo project they were adapted to work as a solo developer strategy. The strategies used that came from agile methodologies are the following:

- Maintaining a simple **product backlog**, that is a list of items you want to complete for this development. In an agile methodology this is flexible and most likely subject to change, this has to be taken into account for any successful project.

- Keeping a **Sprint burndown** and a **Product burndown**. A sprint burndown is a list of elements of the product you want to finish for a certain step of development. Whenever something gets done it is marked as so, this way you "burn down" elements that need completion hence the name. A product burndown is just the same concept but applied to the whole project.

- **Relative estimation and velocity.** We estimate how much time certain tasks are supposed to take, and we assign points to each of these tasks based on this concept. The point of this is not to have foresight on how long a task is going to take but in being consistent with estimations. Velocity is a measure of how many points get done every sprint if points are consistent it can be used to estimate which tasks are taking longer than expected and which are taking less, with this knowledge we can revise how many tasks to tackle on upcoming sprints.

- Knowing our average velocity, we need to also **Limit work in progress**, we should not start working in all the tasks at the same time. This much seems obvious but it is essential to not performing an underwhelming job in the sprint.

- Keep **tasks** as **small** as possible to correctly distribute the work in between sprints they need to be divided in smaller work units, this way they are easier to manage. We also want to have the **ratio between** the smallest and the largest **tasks** relatively small, 4:1 or even 3:1 ratio are acceptable.

- **Perform sprint retrospectives**. It's important to see how sprints went by looking back on them; this way it can be clearer in further sprints what must be changed to perform better.

With all the points considered it is pretty obvious we are going to use Kanban to divide the work to be done into a multitude of tasks and keep an organized sprint and product backlogs. For this we are going to use **Trello** as it is a free tool and an acceptable one for its "price".

Since the project requires a timeframe of 300-360 hours to be developed, we will also use a time manager to determine, first, the amount of time taken to develop tasks versus the time estimated for them; second, the time left for the project to not surpass the established limit.

2.4. BACKLOG, BURNDOWN AND COSTS:

Here we can see the project backlog in Trello with all the task separated into 4 possible categories and with some points assigned to them; these points will make possible determining the velocity at which it was being worked at compared to the task estimation made.

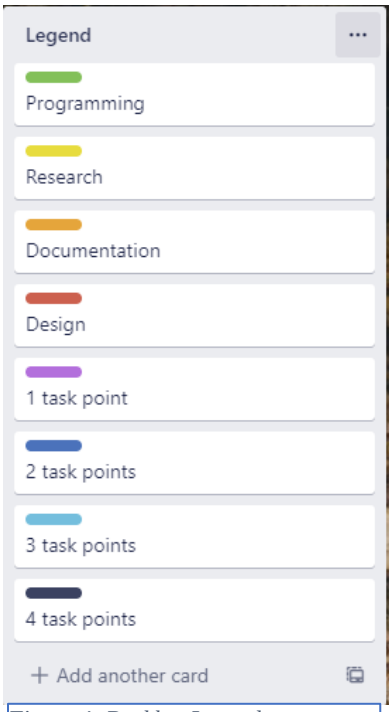


Figure 4: Backlog Legend

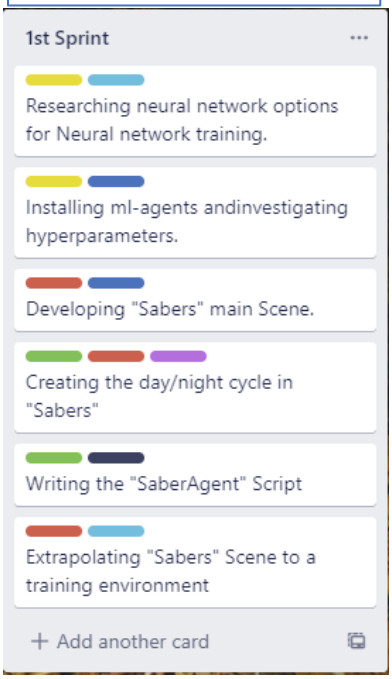


Figure 3: 1st Sprint Backlog

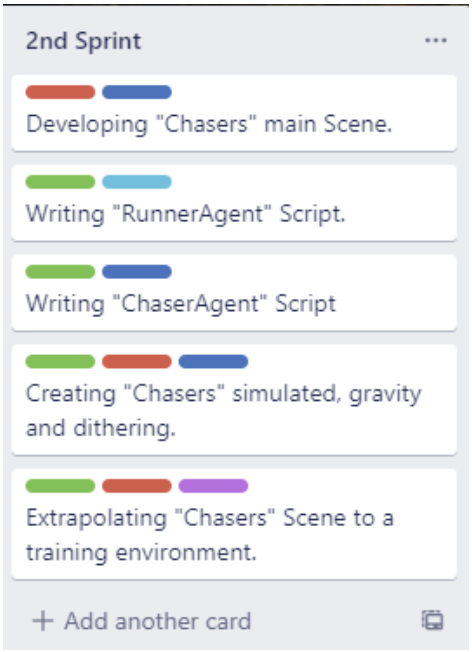


Figure 5: 2nd Sprint Backlog

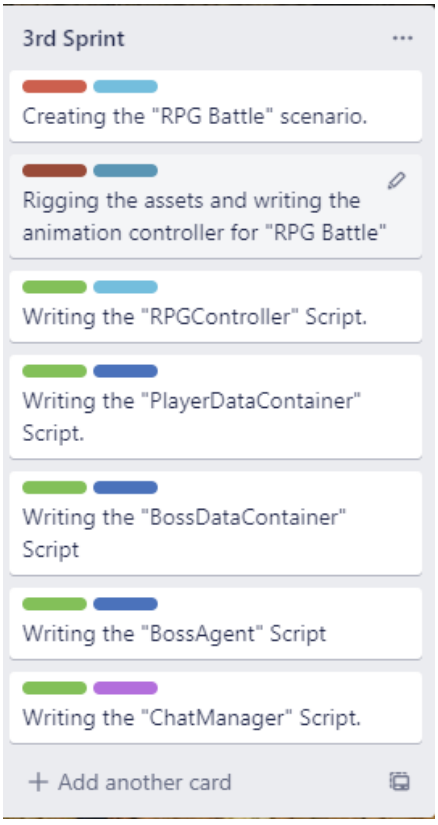


Figure 6: 3rd Sprint Backlog

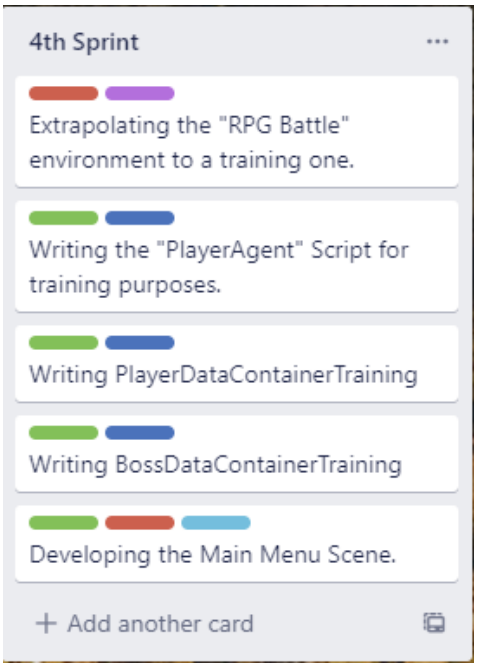


Figure 7: 4th Sprint Backlog

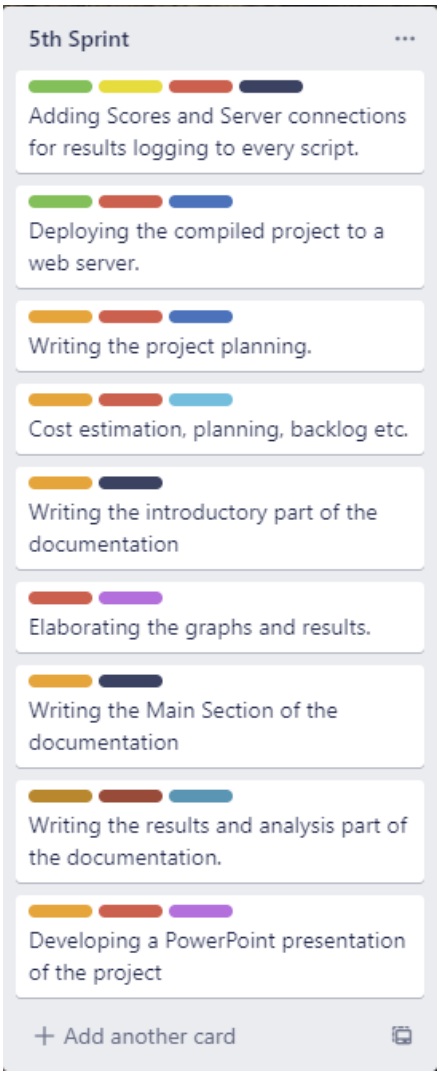


Figure 8: 5th Sprint Backlog

There is also costs of the project. as a baseline the main (and only) developer salary would be the bulk of the costs, hence we determined an approximate amount for this using a job salaries website to get an average for a software engineer in Seville.

Figure 9: Software Engineer salary in Seville [9]

As well as a tax calculator to know how much of the estimated amount would actually be the net value. This is not exact as this is an estimation for a company and this would probably be a freelancer job which would pay very differently but as an estimation, we consider it to be enough.

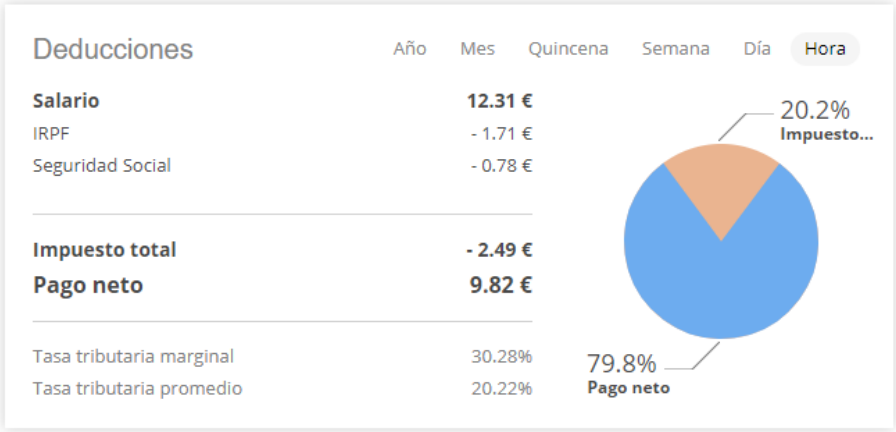


Figure 10: Salary unwrapped [10]

Down below we have added a detailed table with all logged tasks every sprint, the duration of the task (max 1 day, omitted start and end times, as well as, start and end date), and the cost attached to it.

Description	Sprint	Duration	Cost
Cost and plannings estimation	Sprint 5	9:08	112,43
Graph and results comparison	Sprint 5	3:16	40,21
Power Point presentation of the project	Sprint 5	4:00	49,24
Writing the main section of the abstract	Sprint 5	17:15	507,8
Compilation and deployment to webserver.	Sprint 5	8:39	106,48
Scores and Server connections	Sprint 5	18:15	224,66
Main Menu Scene and scripts	Sprint 4	11:29	141,36
Writing BossDataContainerTraining	Sprint 4	8:40	106,69
Writing PlayerDataContainerTraining	Sprint 4	10:38	130,89
Writing PlayerAgent Script for RPG	Sprint 4	6:34	80,83
Making the RPG training environment.	Sprint 4	10:40	131,3
Writing ChatManager	Sprint 3	5:35	68,73
Writing BossAgent Script	Sprint 3	7:16	89,45
Writing BossDataContainer	Sprint 3	5:20	65,65
Writing PlayerDataContainer	Sprint 3	8:51	108,94
Writing RPG Controller	Sprint 3	11:54	146,49
RPG asset rigging and animation controller writing	Sprint 3	10:10	125,15
RPG Battle Scenario	Sprint 3	12:41	156,13
Chasers simulated gravity and dithering effect	Sprint 2	12:09	149,56
Chasers training environment	Sprint 2	6:56	85,35
Chaser Agent Script	Sprint 2	10:33	129,87
RunnerAgent Script	Sprint 2	12:20	151,83
Chasers Main Scene	Sprint 2	10:58	135
Sabers Training Scene	Sprint 1	9:37	118,38
Day night cycle in Sabers	Sprint 1	3:10	38,98
Writing SaberAgent Script	Sprint 1	13:32	166,59
Sabers Main Scene	Sprint 1	9:38	118,59
Installing ML-Agents and investigating Hyperparameters.	Sprint 1	6:45	83,09
Researching Neural Network options for neural network training.	Sprint 1	11:22	139,93
TOTAL	5	305:37 h	3.762,53 €

Table 1: Total duration and cost of the project by task and day

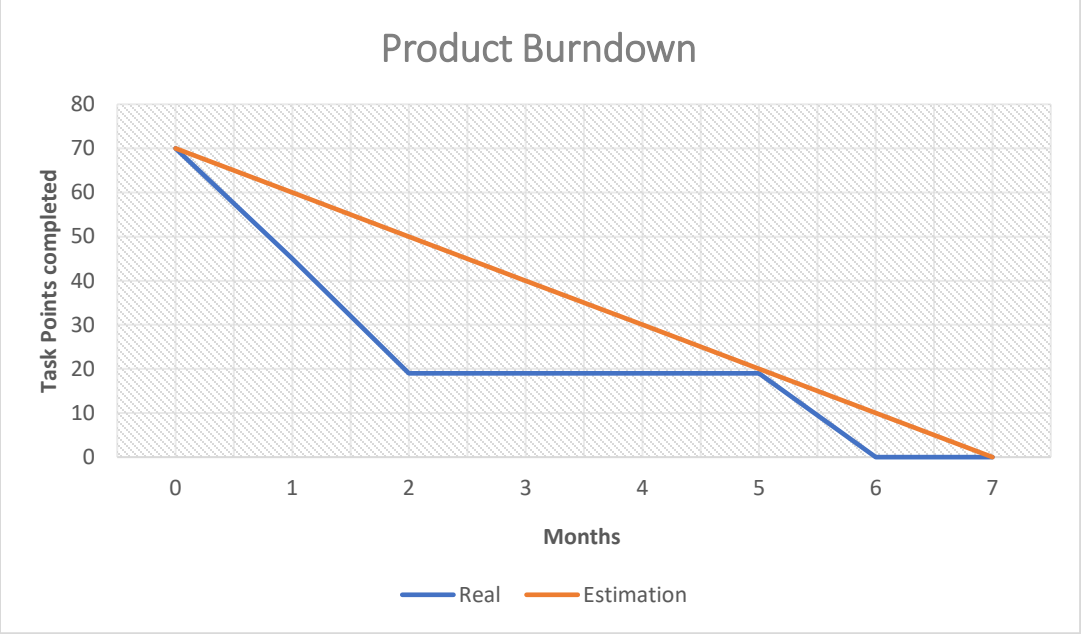
Thanks to the tools used we can conclude that the total salary costs are 3,762.53 €.

We should also add the direct cost of the resources consumed to develop the project which would be electricity, clocking at 0.08€ kWh and consuming on average around 0.6€ per 10h of work.

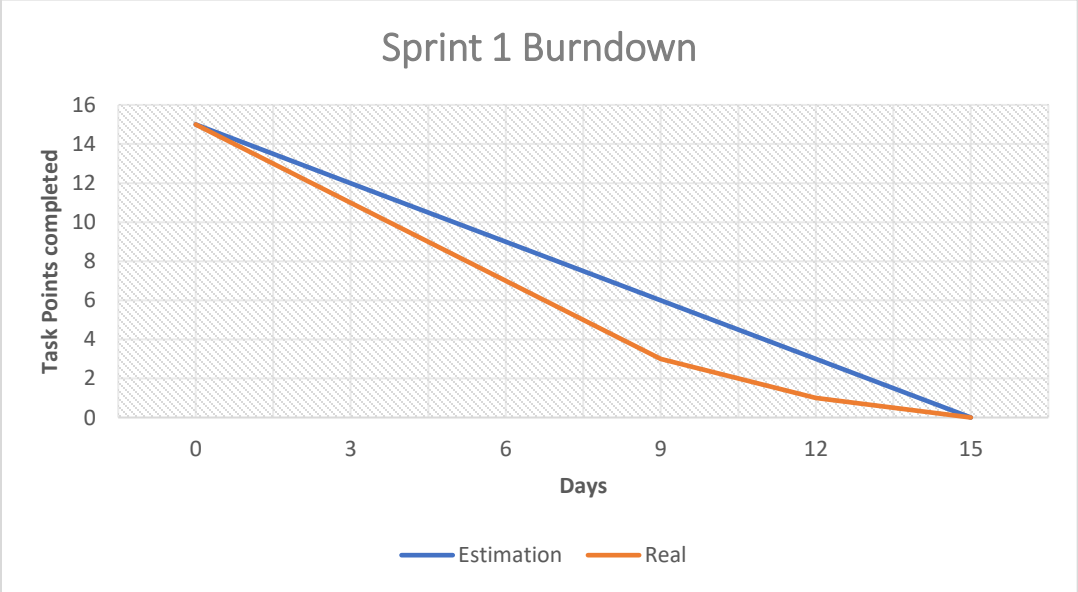
Considering we would also have to factor in the hours the agents take to train (which is around 200 since they were trained multiple times for testing purposes and failed attempts).

That would report us around 30,33 € of indirect costs, that added to the previous amount would result in **3.792,86** € total costs.

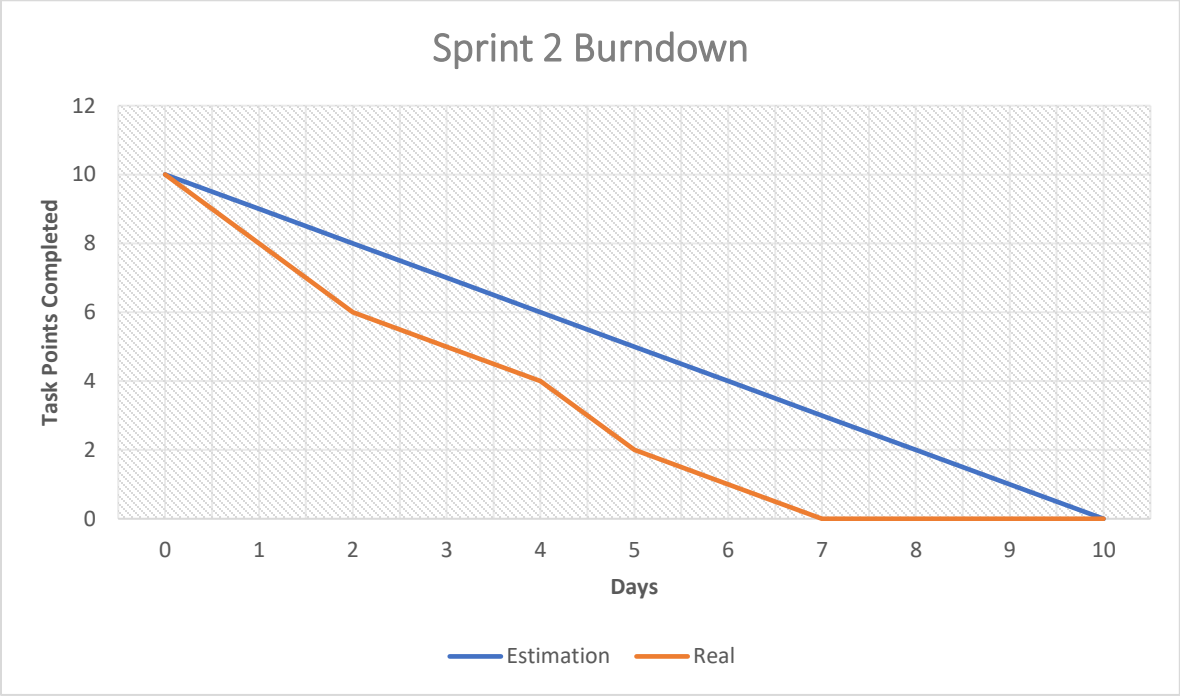
We also have detailed graphs that show the project burndown as well as the sprint burndowns calculated based on the task points that could be seen in the project backlog:



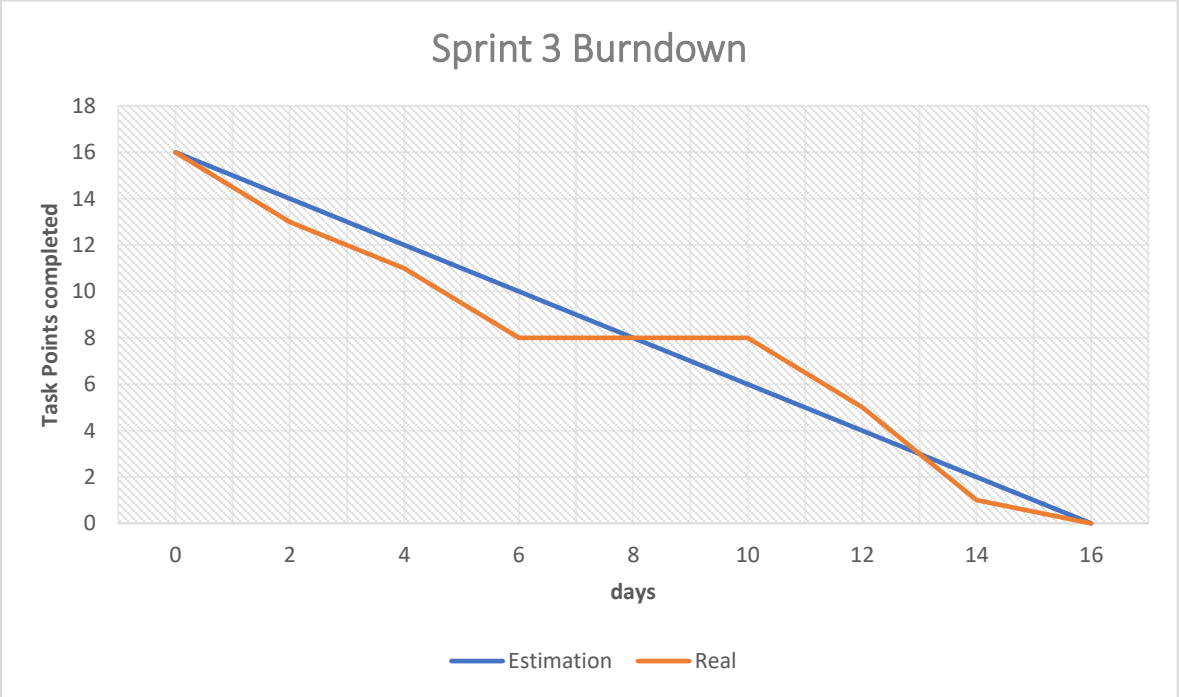
Graph 2: Product Burndown



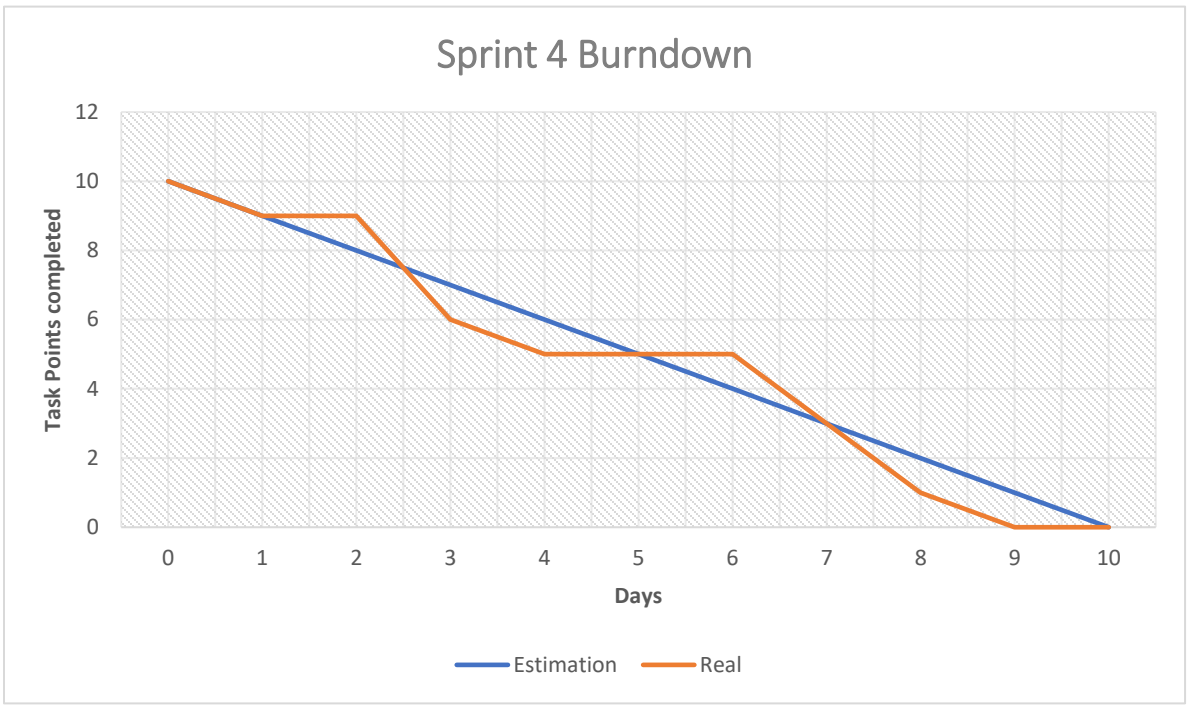
Graph 1: Sprint 1 Burndown



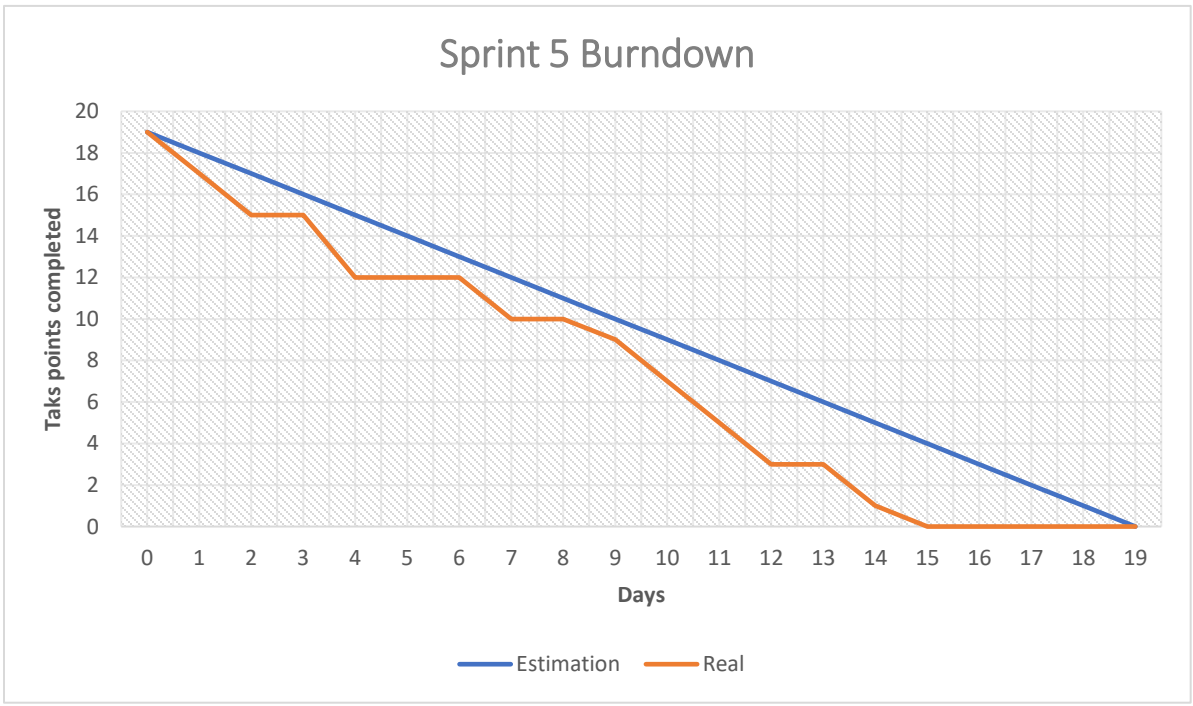
Graph 3: Sprint 2 Burndown



Graph 4: Sprint 3 Burndown



Graph 6: Sprint 4 Burndown



Graph 5: Sprint 5 Burndown

3. PROBLEM ANALYSIS

3.1. STATE OF THE ART

Current IA technology as we've mentioned before is done with scripting, its just a decision tree that takes into account its environment and general player parameters to evaluate how to act. The result is generally fairly predictable and, depending on how many reflexes it requires to beat, also quite unchallenging enemies.

Depending on the game engine used there are several coding languages that we can use for this, **Unreal Engine** for example uses C++ scripting with several tweaks of their own as their choice, **Unity** on the other hand uses C# as their choice, this language in particular is close to Java in syntax but closer to C++ in execution runtimes and general "closeness" to low level components.

There are a lot of game engines and multitude of scripting languages used but in general they all opt for lower level languages. This also limits the capacity of these languages to develop complex AI systems as they have to be built from the ground up instead of using tools that have been developed already.

There is also the factor of complexity in AI programming, decision trees can get cumbersome pretty fast. Here we can see an example of a state machine program which is not very easy to understand or follow at all.

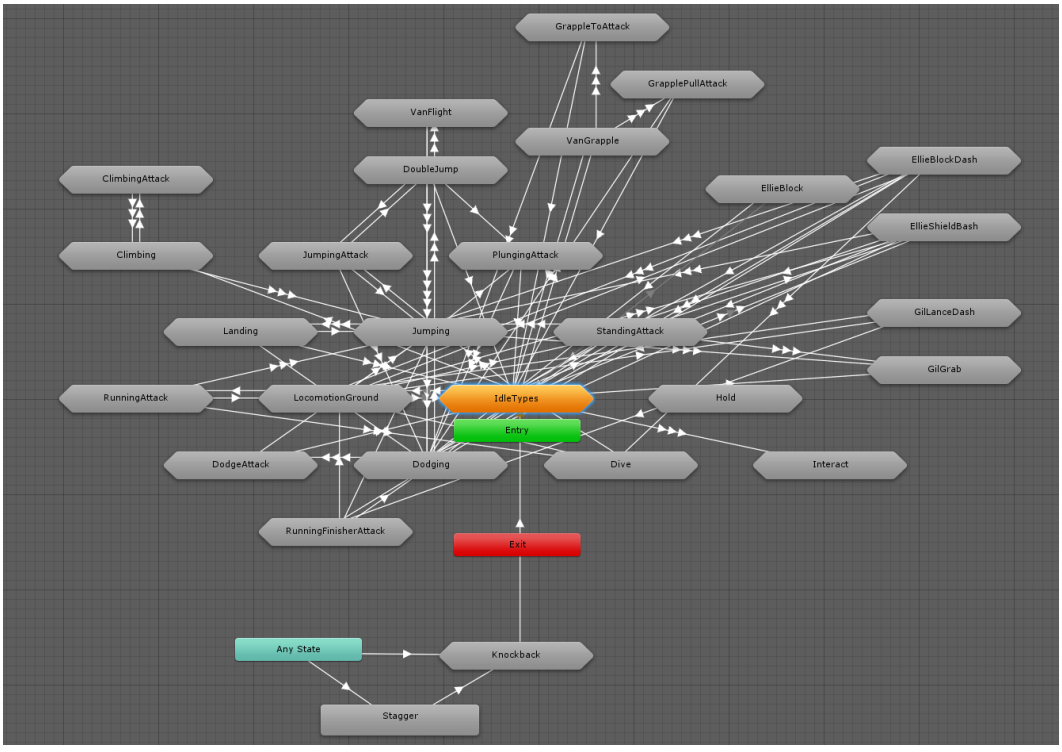


Figure 11: A "very simple" AI state machine [11]

Since this state machine diagram is only here to show how complex these systems can get it will not be explicitly explained, but you can imagine designing, creating and

debugging this type of AI is very cumbersome and error-prone. This example is actually an animation controller, but it applies to AI state machine programming too.

We can see in this tweet [12], how the tool used for this project was released just shy of two weeks before the projects deadline. This portrays how much the technology used to create the Agents of the project is actually state-of-the-art. To our knowledge unity is the only game engine with this type of support. Likewise, we don't believe this technology has been applied by any triple-A video game studio before.

This is where this project will try to make its dent, we want to prove how trained agents can serve as a faster and more efficient way to create AI elements for video game assets.

3.2. PROJECT REQUIREMENTS

We need to provide an algorithm that is capable of training multiple AI elements in different situations, allowing us to focus on different parts of game design and take charge of generating reliable AI.

To complete this task the project must provide:

- Multiple trained agents
- Multiple types of training scenarios
- Training scenes for all of the agents
- A fully developed game with a scene selection screen
- A way to log the results of the games.
- A way to analyze the results of the training.

To qualify, we will investigate how different training environments respond to the training. We will have 3 cases in particular:

- Physics based simulation of 2 agents in a frictionless environment to try and hit the other agent with a stick without getting hit themselves.

- Simulated gravitational pull towards an object where 2 agents chase and run away from each other respectively, this object is just a sphere simulating a small planetoid.

- Finally, a classic turn-based combat scene where there is just 1 agent training since the other one just performs actions at random. This training scene would simulate more closely what a classic scene of a videogame would be. A scene like this could be used with interchangeable enemies and create a bunch of neural networks to advance development speed by an enormous factor since it would only be necessary to program the actions the agent could take and rely on the training to perform the selection for us. You could go as far as to save the progress in different stages of the training to adjust for difficulty settings.

3.3. POSSIBLE SOLUTIONS:

To generate multiple trained agents, we can use a multitude of options;

We can train the agents by **playing against them** and let them learn from us, this is a very time-consuming approach since it requires a lot of playtime.

We can train agents **versus an AI** that is manually created, this is a medium effort approach since you have to define an AI that is a valuable opponent for the Agent. We've chose this option for one of the agents for various reasons that will be explained in further detail later on.

We can also train 2 agents with 2 different AI making them play against each other, this is the faster of the 3 approaches and the main one we are going to take. This option is referred to as **Self-Play.**

The above methods can be combined with any of the following to enhance the learning experience, even though these methods can cause the training to be slower.

- **Curriculum Learning:**
  "Curriculum learning describes a type of learning in which you first start out with only easy examples of a task and then gradually increase the task difficulty. We humans have been learning according to this principle for decades, yet we don't transfer it to neural networks and instead let them train on the whole data set with all its difficulties from the beginning on." [20]

- **Environment Parameter Randomization:**
  This method consists of modifying the different environmental parameters like gravity, mass or scale. This is only related to physics-based simulations that make use of Friction, drag and other physics related parameters. The paremeter modification happens every N number of steps and it allows the networks to learn better.

- **Recurrent Neural Networks:**
  We can specify if we want to use memory for our Network, in cases that need to know what to do in certain cases it is highly recommended to add memory to our networks. For example, if the solution needs the agent to remember a certain pattern, color etc.

- **Multi Instanced Training:**
  We can use multiple instances of the training simultaneously to generate various results from the same training session, this is a very important step that allows us to select which training has been more satisfactory and use that.

4. DESIGN

We've chosen unity as our game engine because of the ML-Agents package that is currently being developed for it. This alternative combines one of the best Machine Learnings solutions, Keras and Tensorflow, which are python packages, with the relatively low-level languages provided by the engine C#.

This connection will allow us to benefit from the best of both worlds. It allows us to create fast training methods provided by the python libraries, as well as, building environments and relationships within the game engine itself leaving a black box in between that takes care of the connection.

4.1. PROJECT ARCHITECTURE:

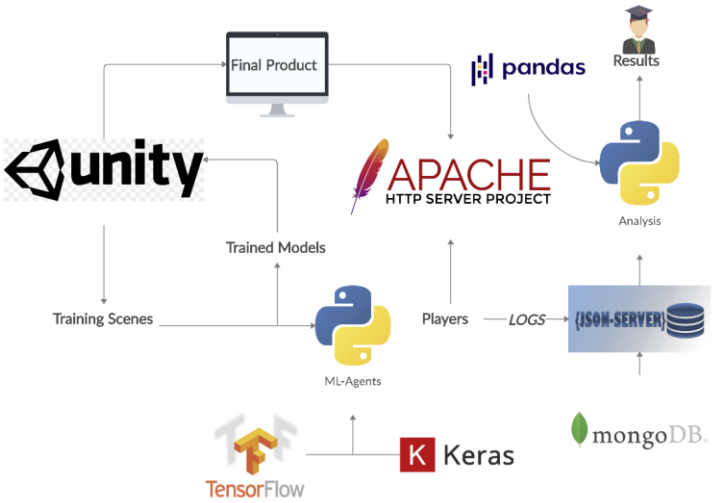


Figure 12: Project structure overview [13].

This diagram shows how the project elements will interact with one another, we will explain each of the elements below.

First Unity will be used to produce the environments (training scenes) for the agents to train on, this is the backbone of the projects structure. It will be described in detail further down the line.

Second, we will use the ML-Agents tools provided by the Unity development team to produce the neural networks that will then be fed back to the agents in its corresponding playing scenes. We will then connect of all of these in a fully developed game that make use of pre-trained neural networks.

This Game can then be compiled as an HTML5 element and deployed to an Apache webserver, there players could access it and play it. The game itself would be able to log the results of the sessions and log it into a database using the json-server. This technology is basically just an API wrapped around a mongoDB instance that can be entered as a any API would, via POST petitions.

There we would later on access them and get some result analysis with the use of python and the pandas package. This particular package allows us to manipulate big volumes of information such as the results from the training and create a more visual structure for them to correctly assess them.

4.2. UNITY, AN OVERVIEW:

We need to understand a little bit of the main tool of this project. Unity is a 3D game engine; this means that it allows us to create whatever we want to create as long as we know how to do so. We can modify physics, object properties, introduce surfaces, lighting effects, use different render pipelines, the possibilities are basically endless, but how can this be done?

We can use a plethora of methodologies but due to our background in programming in the university we will resort to **scripting**. Being one of the most powerful tools unity provides, it gives us the ability to attach a script to a **GameObject**, the main language for writing code in unity is C#, but it also allows for the use of JavaScript, in this case we opted for C# due to the similarities with object-oriented languages.

GameObjects are everything that can be placed in a **Scene**, essentially they make up everything that can be seen on screen, they can have multiple Components which determine how that GameObject will behave, there are multitude of components included by default and these can be written by ourselves as well, in this case they are called behavioral scripts.

Scenes are self-contained groups of GameObjects that interact with one another. It's an environment of whatever you want to join together.

To illustrate this a little better, Imagine the classic Super Mario. Every level in that game is most likely a different scene that ends whenever you touch the flag pole. GameObjects would be every block, enemy, coin, mushroom, etc… basically everything, and scripts attached to them would cointain the information on what they must do, goombas just walk and die whenever they get stomped by Mario, Koopas turn into a shell… all of that logic is contained into scripts.

Figure 13: Super Mario Bros

SOLUTION IMPLEMENTATION

5. CHASERS

5.1. Chasers Playing Scene.

It contains two main elements, the Game elements and the GUI elements:

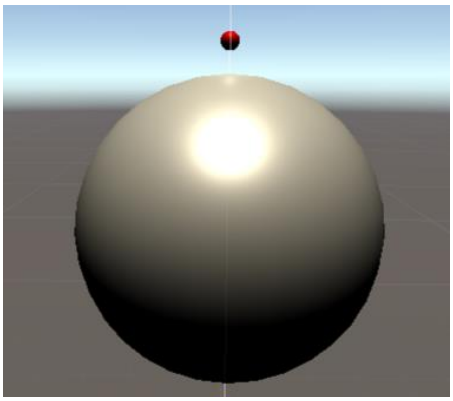


Figure 14:Game elements of chasers playing Scene.

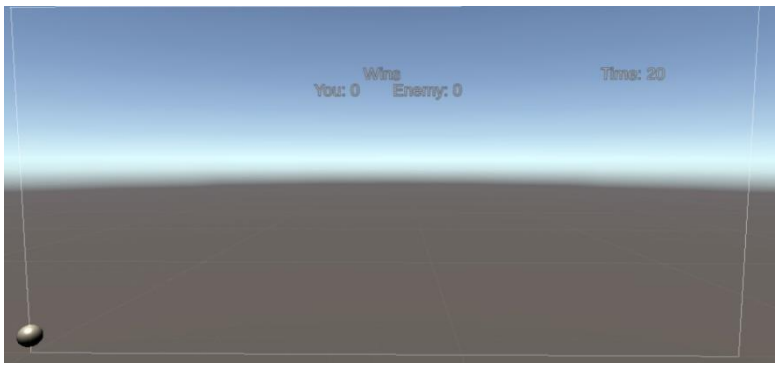


Figure 15:GUI elements of the chasers playing Scene.

Figure 16:Game Element in the menu.

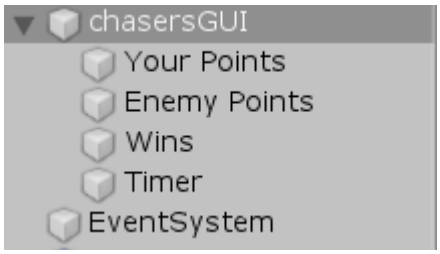


Figure 17:GUI element in the Menu.

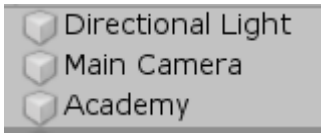


Figure 18: Chasers playing Scene miscellaneous elements.

First, we will note how the “Game” element (figure 16) is a Blue box in the menu, this indicates that this element is **Prefab** object, prefabs are a conglomerate of game objects that act as one, meaning you can put them on any scene and they will work as intended (if they were made correctly). This is used to build Scenes in an easier and more streamlined way.

5.2. The Game element

The first of the objects we are going to look at is the **planet** game object:

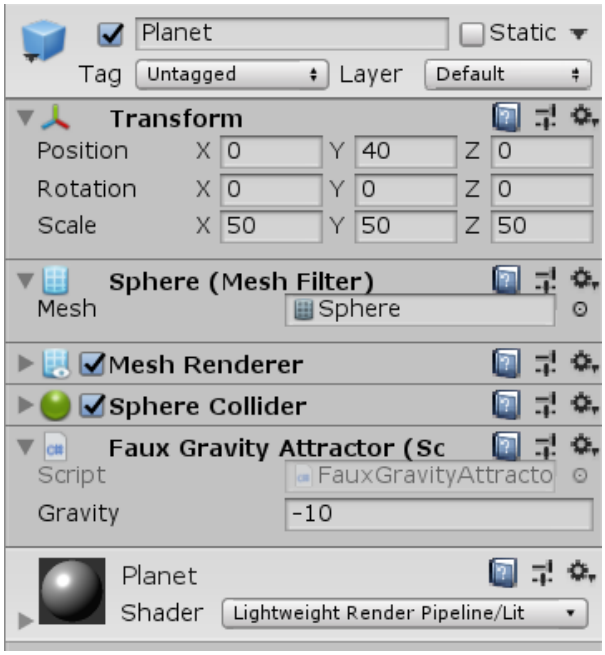


Figure 19: The planet Game Object

It's a simple sphere determined by the Mesh Filter, Renderer and Collider components we will not enter in too much detail about object rendering in this project as it is not the main focus.

The **transform** component is the parent of all GameObjects, it contains the information about the objects position rotation and scale, and it also contains multitude of different and very useful functions that will appear throughout the abstract.

The Shader component determines how light interacts with the object; in this case we are using the Lightweight render pipeline to optimize performance.

Finally, The Faux gravity attractor is a game object used to override Unity's physics engine gravity pull and make a simulated one towards the object. Let's see how the script works.

```
using UnityEngine;
public class FauxGravityAttractor: MonoBehaviour
{
  public float gravity = -10;
  public void Attract(Transform body){
      Vector3 gravityUp = (body.position - transform.position).normalized;
      Vector3 bodyUp = body.up;
      body.GetComponent<Rigidbody>().AddForce(gravityUp * gravity);
      Quaternion targetRotation = Quaternion.FromToRotation(bodyUp,gravityUp)*bod
y.rotation;
      body.rotation = Quaternion.Slerp(body.rotation,targetRotation, 50*Time.fixe
dDeltaTime);
}}
```

Code 1: FauxGravityAttractor

Unity preferred scripting language is C# and all the code will be in this language.

Unity uses **MonoBehaviour** as the parent class for most of its scripts.

In this script we can see that the gravity float value is -10, this indicates that objects influenced by this will try to move 10 units towards the planet. This happens whenever the Attract function is called.

gravityUp is the vector between the object being attracted and the planet normalized.

bodyUp is the vector going towards the Y axis from the attracted object.

The **getComponent** function gets the component we ask for in the GameObjects in this case it is the objects being attracted and it getting its **Rigidbody** which is responsible for physics based operations, this component is probably the most important component in any game engine.

We then use the **AddForce** function of the Rigidbody component in the direction from the object to the planet, meaning that the object is **attracted** to the planet.

Finally, the last 2 lines are responsible for rotating the game object in a manner that is consistent with the planets surface, this is made through **Quaternions**. These are basically X, Y, Z rotations with an extra W variable that indicates Euler angles. They avoid the Gimbal Lock problem in rotations, they are pretty complex, so we will not delve more into them.

In essence these lines use the objects up (Z) vector to maintain the same relation between them (the attracted objects up vector must be parallel to the gravityUp one), this is made with the Slerp function which "*Spherically interpolates between a and b by t. The parameter t is clamped to the range [0, 1].*"

We will now look at the **Runner** GameObject, this is one of the agents that were trained:

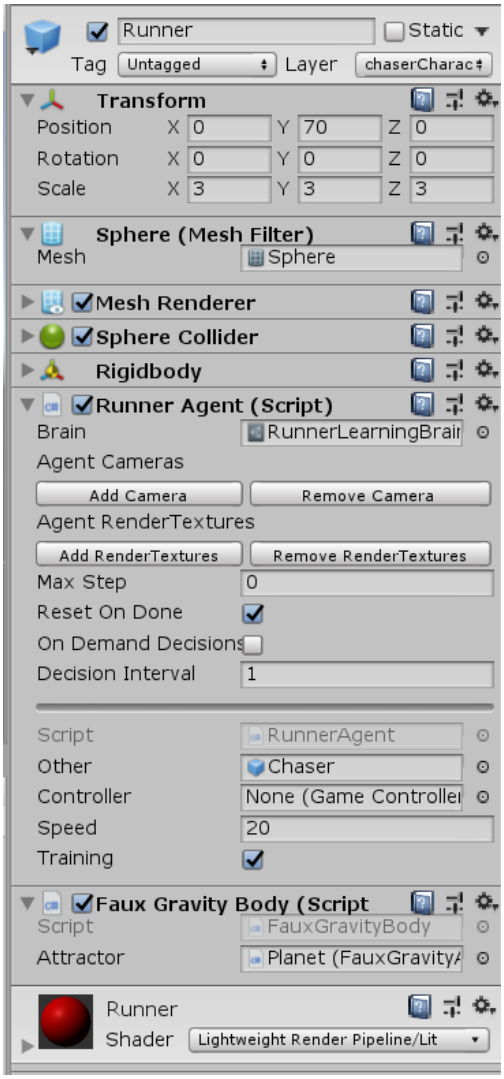


Figure 21:Runner GameObject

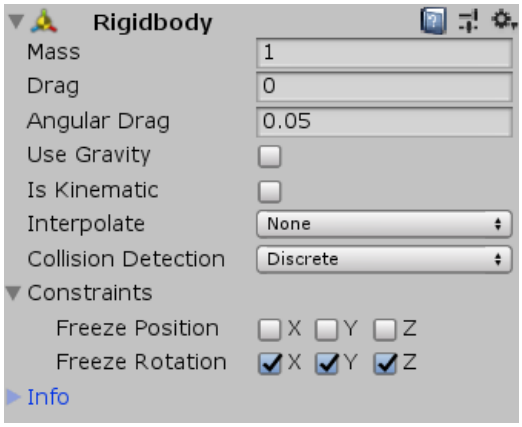


Figure 22: Runner GameObjects rigidbody component.

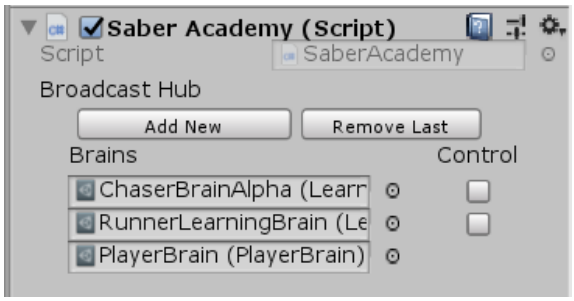


Figure 20: Academy Script in the editor

As we can see all the objects that need rendering and 3D models contain the same components, we can also see the Rigidbody component (figure 20) that allows physics calculations is also present. we won't get much into details apart that we froze its rotation to modify it only via Script.

This is the case for it not to get wrongfully updated. We can also see the RunnerAgent(figure 22) Script and Faux Gravity Body Script(code 2), we will get into the latter first, but before that lets also look at the academy GameObject (figure 21):

The academy GameObject must be present in every scene where there is an agent Script as it is responsible of the interaction between the GameObjects and the **neural networks** they use as their AI.

The Academy script is just an import of the MLAgents Academy class, it contains the connection elements for the python package to find the game elements it need to and train them.

As for the FauxGravityBody script it is very simple:

```
using UnityEngine;

public class FauxGravityBody: MonoBehaviour
{
    public FauxGravityAttractor attractor;
    void Update()
    {
        attractor.Attract(transform);
    }
}
```

Code 2: FauxGravityBody

There is only a reference to the FauxGravityAttractor which is made explicitly in the editor by moving the planet game object from the scene into the corresponding element. In this case one of the balls in the planet. Then we use unity **Update()** function that **executes every frame** to run the Attract functions code we saw earlier with a reference to this GameObjects transform.

Now for the RunnerAgent code we will break it down in sections for simplicity:

```
using UnityEngine;
using MLAgents;

public class RunnerAgent: Agent
{
  public GameObject other;
  public GameController controller;
  public int speed;
  Rigidbody rBody;
  Vector3 initialPosition;
  float timeAlive = 0;
  bool iVeBeenCaught = false;
  private int counter = 0;
  private float localreward = 0f;

  private float inner_timer = 20.0f;
  public bool training = true;

    void Update(){
        inner_timer -= Time.deltaTime;
    }
```

Code 3:Runner Agent Code variables.

We see just like in the academy case we also use something other than MonoDevelop **Agent** is the class that allows us to use machine learning in this engine.
The variables are the following.

other, is a reference to the enemy GameObject in the scene (chaser)
controller, will be a reference to the GameController, this object is not currently in the scene and will be talked about in further sections.

The rest of the variables are just flags that control various occurrences in the scene, except for speed that just determines how fast the GameObject can go.

In the Update () function we will only update the inner_timer of the object it will decrease just like a normal timer would thanks to **deltaTime**.

```
void Start(){
  rBody = gameObject.GetComponent<Rigidbody>();
  initialPosition = this.gameObject.transform.position;
  controller = GameObject.Find("Game_Controller").GetComponent<GameController>();
}
```

Code 4: RunnerAgent Start method.

The **Start()** function is also a Unity function it runs once at the start of the scene it is normally used to get references to GameObjects that are not in the scene at load time. For example, here we are getting a reference to this objects Rigidbody component, as well as its initial loading position in the scene.

We also get a reference to the mentioned GameController object, we don't need to worry about it yet.

```
    public override void AgentReset(){
        this.transform.position = initialPosition;
        this.rBody.angularVelocity = Vector3.zero;
        this.rBody.velocity = Vector3.zero;

        iVeBeenCaught = false;
        localreward = 0;
        timeAlive = 0f;

        inner_timer = 20.0f;
        if(!training){
            controller.timer = 20.0f;
        }
    }
```

Code 5: RunnerAgent AgentReset function

This function is MLAgents dependent, whenever a training scene reaches the point when it's considered **done**, this snippet of code will run, in this case it will reset this object transform to its initial value as well as the flags we can see here that indicate the reward, the time it has been alive and if it has been caught by the other agent.
We can also see how if the training flag is false, we also update the controller's timer.

```
public override void CollectObservations(){
    //this method contains the necesary inputs for the neural network
    AddVectorObs(gameObject.transform.position);
    AddVectorObs(other.transform.position);
    //Agents velocity:
    AddVectorObs(rBody.velocity);
    AddVectorObs(other.GetComponent<Rigidbody>().velocity);
}
```

Code 6: RunnerAgent CollectObservations function

This is another MLAgents dependent function it will add either, **vector3**, **quaternions**, **booleans**, **ints** or **floats** to the observation vector, this vector is basically what the neural network will receive as the input parameters and will determine the size of the **brain**, that we will need to use.

Before resuming with the code, we will touch on the subject of **Brains**, it is what MLAgents uses to determine how the agent will behave once its trained, there are 3 types of brains:

LearningBrain are the ones that we will use for Agents to learn from their environment.

PlayerBrains are used for testing, as they are player controller but can also be used to teach agents, we will not be using them for training, but we will use themas a player controller.

HeuristicBrains can be programmed to do whatever we want them to, just like normal AI but allowing easier interaction with other brains. We wont use these types of brains in the project.

We saw in the Academy GameObject there were 3 Brains referenced: ChaserBrainAlpha, RunnerLearningBrain and PlayerBrain

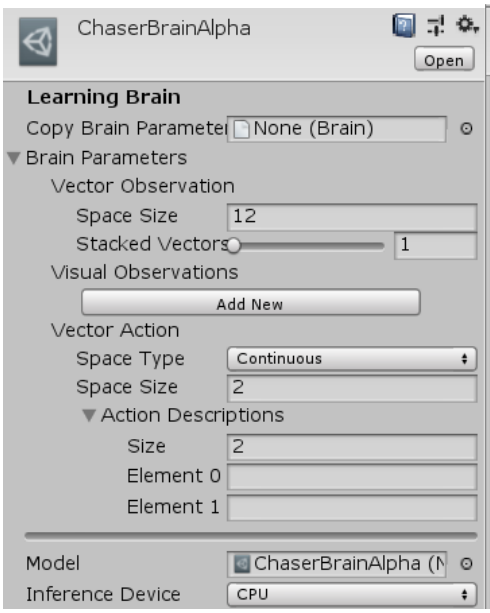


Figure 23: Chaser Brain

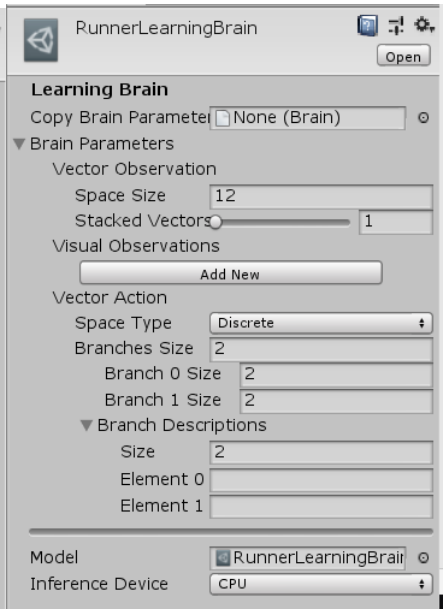


Figure 24:Runner Brain

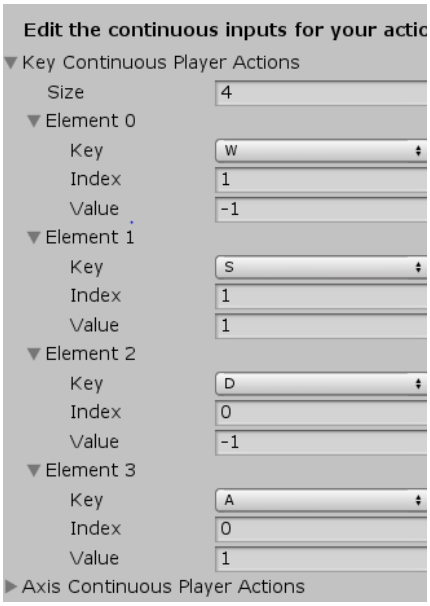


Figure 25: PlayerBrain Key actions

These Objects contain the information that **TensorFlow** receives to make and train the neural network, the Space Size is how many elements are derived from the vector observations this are determined by adding the elements that were introduced there. 3 for vector3, 4 for quaternions and 1 for any other.

The space type determines if the output of the network must be a integer array or a float array, in this case different types were used just to check what would happen, the discreet one made for a more interesting result as it always went as fast as possible.

The PlayerBrain is just a copy of the RunnerBrain but it includes the possibility of just modifying the output with keypresses (just like a normal player controller would do). In this case we are modifying the two possible outputs of the neural network manually, these control the players movement by either inputing force (pushing or pulling) in the objects X, Z axis.

The **Model** parameter of the brains is where we introduce the Trained networks that come from the training.

Now that we understand how brains work, we can see the AgentActions code.

```
public override void AgentAction(float[] vectorAction, string textAction){
        Vector3 controlSignal = Vector3.zero;
        controlSignal.x = vectorAction[0];
        controlSignal.y = 0;
        controlSignal.z = vectorAction[1];

        rBody.MovePosition(rBody.position + transform.TransformDirection(controlS
ignal)*speed*Time.fixedDeltaTime);

        CalculateReward();
        checkCollisions(gameObject, other);
        if(iVeBeenCaught){
            SetReward(-100);
            if(brain.name == "PlayerBrain" && !training){
                controller.IncreaseWin("enemy");
            }
            Done();
        }
        if(inner_timer <=0){
            SetReward(100);
            if(brain.name == "PlayerBrain" && !training){
                controller.IncreaseWin("player");
            }
            Done();
        }
    }
```

Code 7: RunnerAgent AgentAction function

This is the way to get the network outputs to act on the GameObject, as we saw from the player controller there are two outputs that we assign to the controlSignal Vector3, this Vector3 will then be used in the MovePosition function of the objects Rigidbody to move it in the specified direction.

We then calculate the current reward to submit to the network and check for collisions with the other GameObject.

Finally, we check if we've either been caught or if we've escaped for 20 seconds, in any of those cases happen we will mark the scene as done and reset it. In case that the game controller exists (we are actually playing the game) we also add the win to whoever deserves it counter.

```
private void checkCollisions(GameObject me, GameObject other){
       float distance = Vector3.Distance(me.transform.position, other.transform.p
osition);
       ChaserAgent chaser = other.GetComponent<ChaserAgent>();
       if(distance < 6 ){
            iVeBeenCaught = true;
            chaser.caughtOther = true;
       }
    }
```

Code 8:Runner Agents checkCollisions function

This function (code 8) controls if the distance between the 2 GameObjects is less than a certain value (6 in this case) and if that's the case then we activate the corresponding flags. It's important to note that unity contains a collision system that could be used for this, but we found it non-reliable for this particular case, so we decided against it.

```
    private void CalculateReward(){
        float distance = Vector3.Distance(gameObject.transform.position, other.tr
ansform.position);

        SetReward(distance/53);
        localreward = (distance/53);

        if(distance < 12){
            SetReward(-1);
            localreward = -1;
        }
    }
}
```

Code 9: RunnerAgent CalculateReward function

The main difference between the Runner and the Chaser Agent is found in the reward function. It's simple it calculates the distance between the agents and normalizes it to a

[0,1] value as 53 is the largest possible distance between them. It occurs at the beginning of the scene.

We've also included a "Danger Zone", whenever the other agent gets too close the reward immediately changes to -1, in hopes that the agent will start running away as to "tease" the enemy agent.

The SetReward function is a MLAgent function again and it tells the environment how it's doing.

We will now Look into the **Chaser,** GameObject.

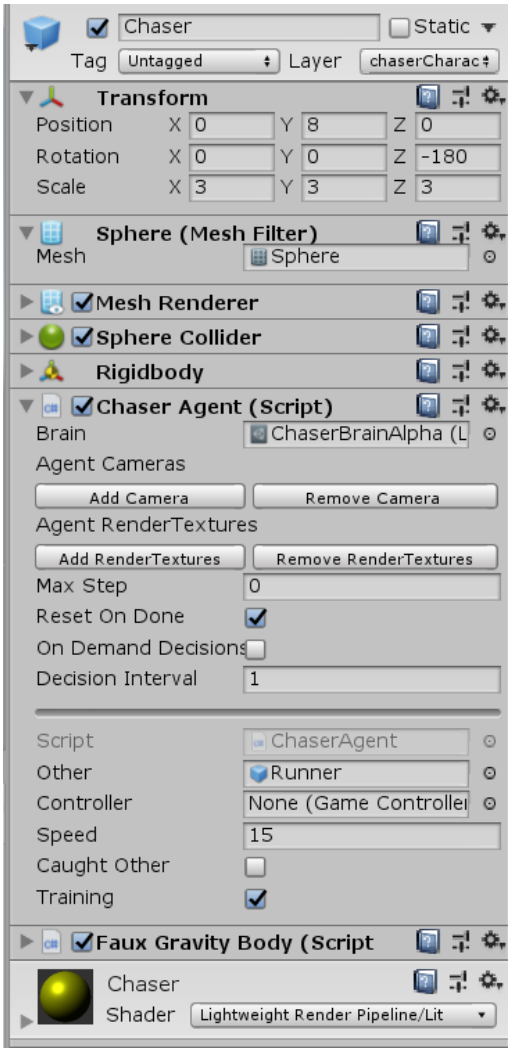


Figure 27: Chaser game object.

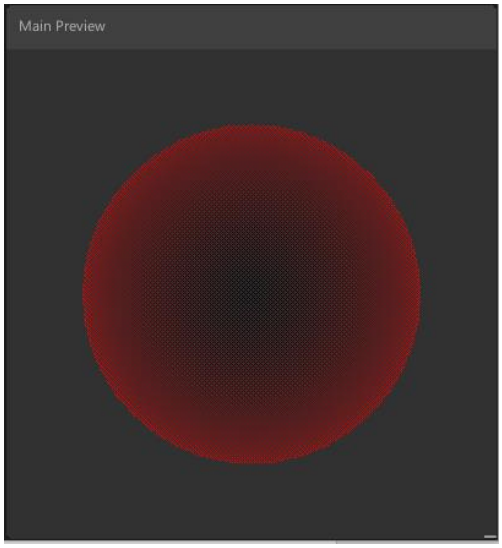


Figure 26:Dither effect preview.

This GameObject is very similar to the Runner GameObject, it contains the same elements as the other one except for the ChaserAgentScript.

This script is very similar to its counterpart, something that wasn't mentioned in the previous one is the reset on done flag, which is responsible of resetting the environment when the Done () function is called. This function belongs to the ML-Agent package and can only be called inside of the Agent class.

We also have to mention the custom **dither** effect that was applied whenever an object was rendered behind the planet.

This relatively small but necessary change took a lot of time because of the full texture update that the project had to undergo, we went from **standard render pipeline** to the **lightweight** one, making necessary to update every single object accordingly.

Here we can see how the dither effect is created in Unity render editor.

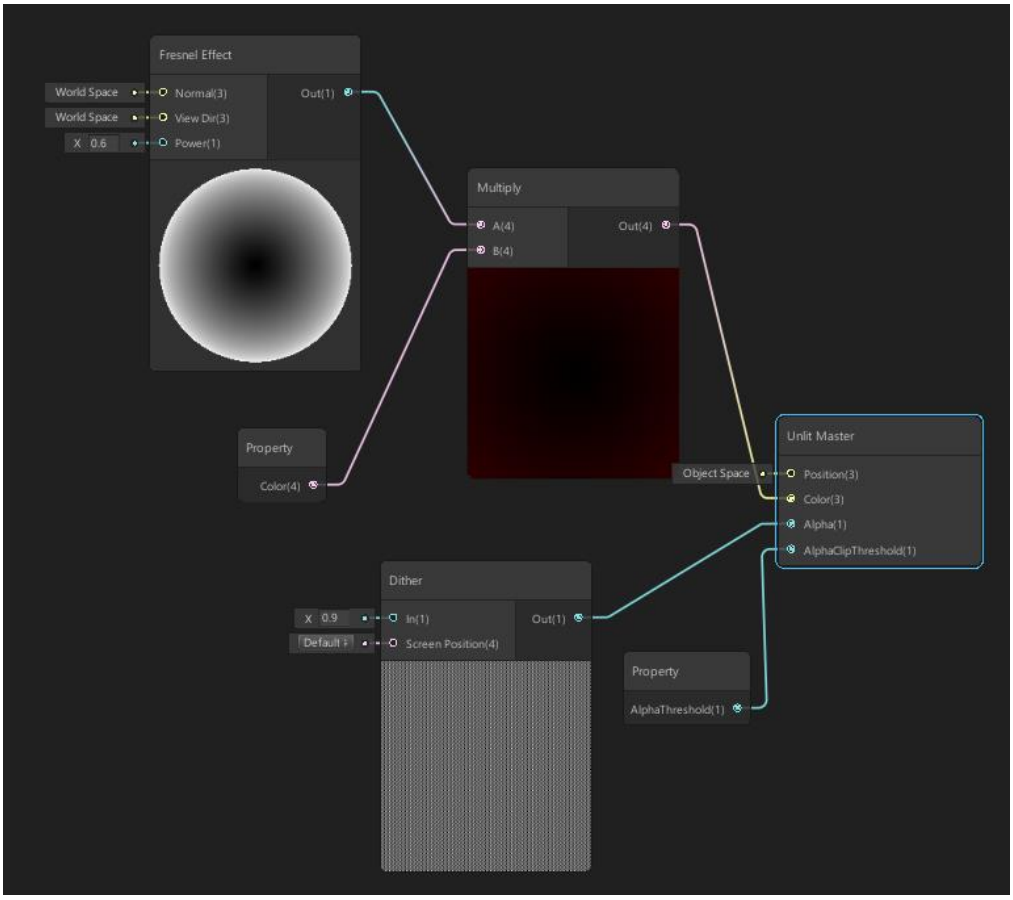


Figure 28:Dither elements in unity

And now with all the scenes kinks out of the way we will look at the Chaser Agent Script Since we have looked at the RunnerAgent Script in a lot of detail, a big part of this Script will be very similar, so we will not go too deep into it.

The only difference with the variables between the two is that this Agent uses the caughtOther flag instead of iveBeenCaught. Both of them serve very similar purposes, to indicate whether the environment should reset.

Update(), Start(), AgentReset() and CollectObservations() are very similar between both agents and will be omitted

```
    public override void AgentAction(float[] vectorAction, string textAction){
        [...]
        rBody.MovePosition(rBody.position + transform.TransformDirection(controlS
ignal)*speed*Time.fixedDeltaTime);
        if(speed < 35){
            speed += 0.005f;
        [...]
        }
    }
```

Code 10: Speed increase in the AgentAction script for ChaserAgent

In this case the way the Agent moves is the same with the exception that it will **progressively increase its speed** up to 35, this is also the only difference in the AgentReset function where the speed is reset to its initial speed of 25.
The rest of the function matches the first one but with the opposite logic, this agent will lose if 20 seconds passes and it hasn't caught the other one.

As for the CalculateReward function its very similar to the RunnerAgentss one (code 9) the only difference is that the reward is the opposite of the other agent, the closer it gets to the enemy the better.

5.3. THE GUI ELEMENT

This element is way simpler than the game element, it is a 2D plane that intersects the camera, it contains 4 sub-elements:

- Player points
- Enemy points
- Wins
- Timer

All of these elements are updated by the GameController which comes from the main menu, so in this scene its just some text on the screen waiting to be called, we will go into detail about this element when we talk about the GameController object.

5.4. Chasers Training Scene

Figure 31: Chasers Training Scene GameObjects.

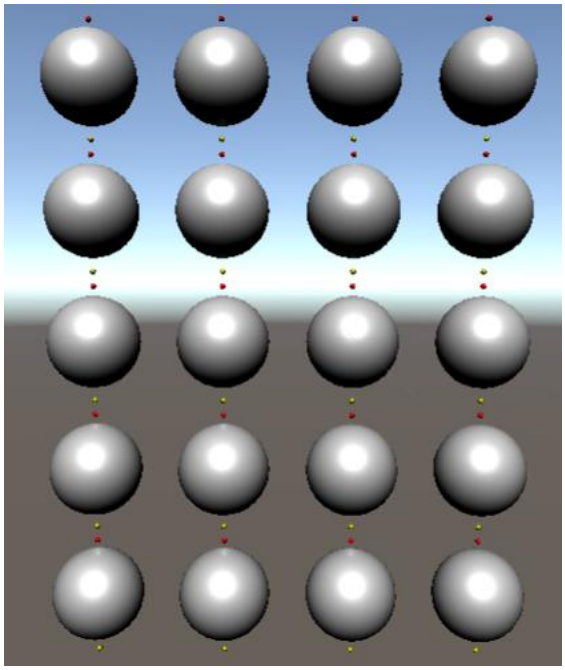


Figure 30: Chasers Training Scene

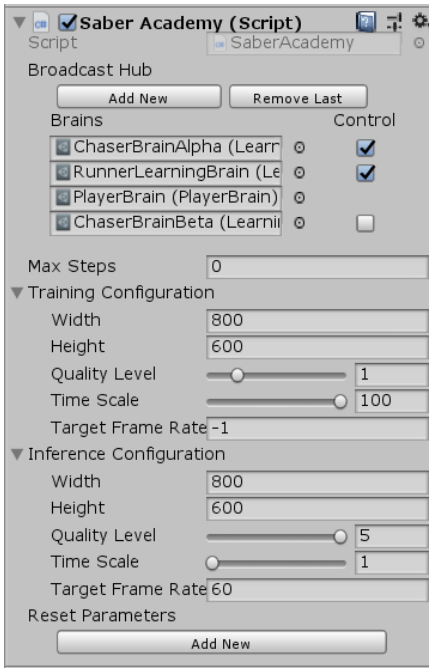


Figure 29: Academy controlling both brains for training

In this case the training Scene is pretty straightforward. It's just a multitude of planets each with its own set of players. This is so whenever the training generates initial randomized neural networks more than one is generated and the others can adapt according to the one with better results. Eventually they will homogenize and behave more or less the same.

This is also a good moment to comment on how these Scenes underwent the biggest change of them all and were for the most part remade from scratch. The original approach was to create really big planes in which the ball could run away. A lot of time went into this idea but the ball that was running away was always in a disadvantage since whenever they reached a wall the one chasing it would always catch up.

Several iterations and code went into trying to teach the ball that ran away that "walls are bad" and "try to stay close to the center". Some of its kind of worked but eventually the solution was to give them an "infinite" playfield, and, what is better than an infinite plane? Well a Sphere of course. Hence the final result of this Scene.

There is nothing to show from the previous Scenes but we thought it would be a good idea to mention where the planet and false gravity idea came from.

The GameObjects here are an exact copy of the ones we already saw. The only difference with the playing Scene is how the Academy game object now needs to control the brains for it train and not leave it to the attached neural networks that were trained.

6. SABERS

6.1. SABERS PLAYING SCENE

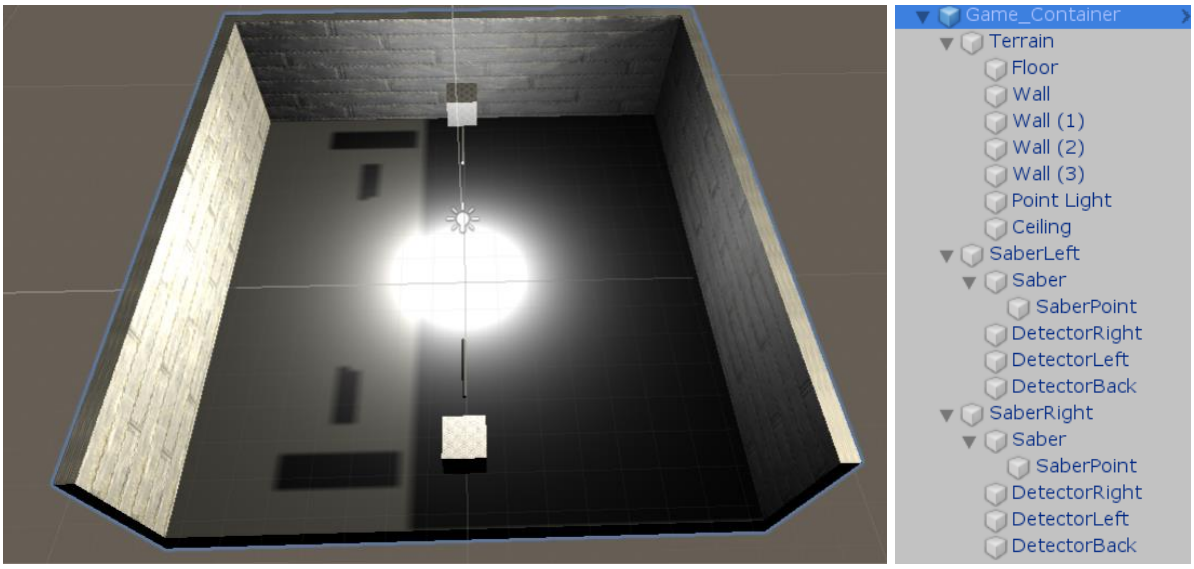
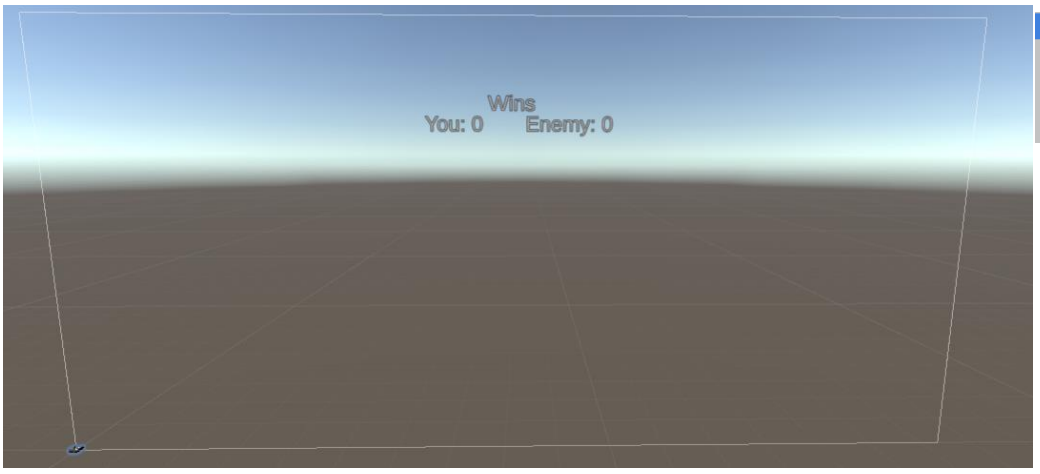


Figure 32: Sabers Game_Container and its menu objects.



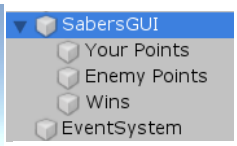


Figure 33:Sabers GUI element and GameObject

We can see, there are plenty of similitudes with the chasers Scene we will analyze the Game_container mostly as it contains the most notable differences as the GUI is virtually the same as we expect it to be for the purposes of score tallying.

There is also a day-night cycle in this game made possible by a very simple script in the Scenes directional light.

```
public class Light_Rotation : MonoBehaviour{
    void FixedUpdate(){
      transform.Rotate (5*Time.deltaTime,0,0);}}
```

Code 11: Light Rotation Script.

The behavior simply rotates by 5 degrees every second making the light point towards and away from the scene simulating sunrise and sunset the point light in the middle of the game container is so the game is visible at night too.

6.2. THE GAME_CONTAINER OBJECT

Starting with the **terrain** which is very simple, the are 3 noteworthy elements, there is a fourth wall with its mesh renderer turned off, there is also a ceiling which also has its Mesh Renderer component turned off, this is obviously so we can see the Scene and play without a wall obscuring the view.

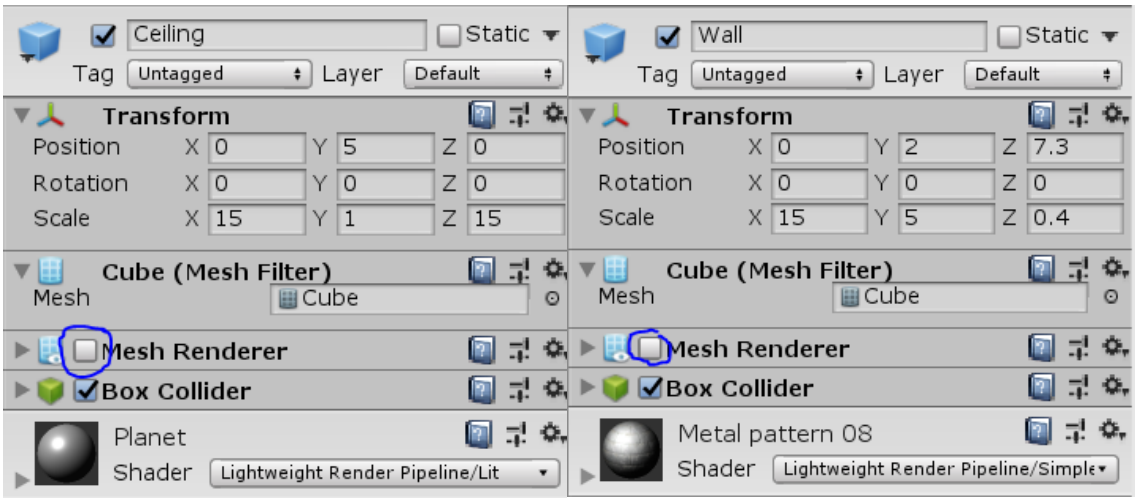


Figure 34: Game Objects without a mesh renderer are invisible

Also, none of the objects here have a RigidBody component that is why they are unaffected by gravity and are not detected by the unity collision system, this is so the Main GameObjects of the scene do not collide with the walls.

Continuing with the **Sabers**, they are both basically the same GameObject the only differences are in the Saber Agent component references.

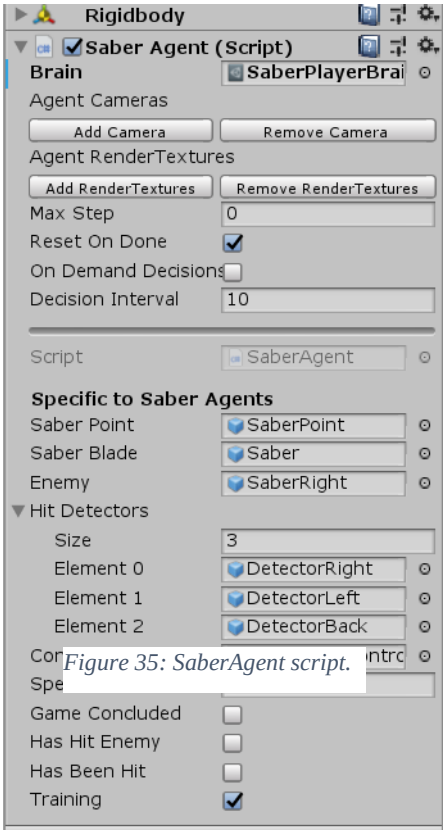


Figure 35: SaberAgent script.

Firstly, the Enemy parameter points to the other GameObject. Secondly, there are two brains, one for the right Agent and one for the left brain. The rest of the parameters are identical for both of them.

Also, both of the agents are made up by a Saber which is simply a cylinder with a ballpoint and some hit-detectors on each of the sides fo the cube except for the front one.

These detectors will allow the agents to determine if they are in danger and act in consequence.Generally getting away from the other agents saber.

We will now explore the SaberAgent Script, we will see there are a lot of similarities between this and both of the chaser scene Agent Script.

Variables are generally uninteresting we will try to skip them from now on. Just like we saw before on code 1 there are some references to Scene GameObjects and some flags.

At the **start** of the Scene we get the Rigidbody component of the object and the initial position and rotation for the environment reset. We also get a reference to the GameController object if its present in the Scene.

```
public override void AgentReset(){
    this.transform.SetPositionAndRotation(initialPosition, initialRotation);
    [...]
}
```

Code 12: SaberAgents reset function

We use the Rigidbody function SetPositionAndRotation to return the environment to its initial state whenever the Agents mark they are done.

The **observations** will be the enemies and ourselves position and rotation parameters as well as our velocity, since this is a totally physics based simulation, we will need to take into account the rotation as an observation because it determines the position of the saber.

```
public override void AgentAction(float[] vectorAction, string textAction){
    Vector3 controlSignal = Vector3.zero;
    controlSignal.x = vectorAction[0];
    controlSignal.y = 0;
    controlSignal.z = vectorAction[1];

    rBody.AddForce(controlSignal*speed);
    rBody.AddTorque(0,vectorAction[2]*10,0);
    [...]
}
```

In this case we can see how the **output** of the neural network is of **size 3**, outputs for X, Z linear movement and another to determine rotation. We use the AddForce and AddTorque to the cube object of the Agent to move it around the Scene. We then calculate its reward and check if any of the two have been hit by the enemy saber or if they have hit the enemy themselves.

To check if the enemy player has **collided** with our saber, we check if the distance is smaller than 0.65, since the distance is calculated to the center of the object (in this case the cube), and the its size is 1 we approximate 0.65 (because of the corners and some leeway) to detect a collision. If either the saberBlade or the saberPoint touch the cube we consider that this player has won and we mark it as such.

```
    private void CalculateReward(){
        float distance = Vector3.Distance(saberPoint.transform.position, enemy.tr
ansform.position);
GameObject enemyBallPoint = enemy.transform.GetChild(0).GetChild(0).gameObject;
        AddReward(-1f / 3000f);
        currentReward += (-1f/3000f);
        AddReward((9.2f-distance)/9.2f);
        currentReward += ((9.2f-distance)/9.2f);
        if(Vector3.Distance(hitDetectors[0].transform.position,enemyBallPoint.tra
nsform.position) < 1.5 ||
           Vector3.Distance(hitDetectors[1].transform.position,enemyBallPoint.tra
nsform.position) < 1.5 ||
           Vector3.Distance(hitDetectors[2].transform.position,enemyBallPoint.tra
nsform.position) < 1.5){
                AddReward(-5f/3000f);
                currentReward += (-5f/3000f);
        }
}
```

Code 13: Reward calculation for the SaberAgent

The reward function takes into account 3 things. Firstly, we add a **negative reward for existing**, this is so the agents want to take action fast, they will be more **aggressive** this way.

Secondly, we add a positive reward the closer the saber point gets to the enemy, this is so the agents' positive reinforcement is linked to the action of hitting the other agent.

Lastly, we use the aforementioned hit detectors located on the middle of each of the agents' sides to check if the other agents' saber is close to us, if so, we add a high negative reward as a penalty to try for the agent to play avoiding the enemy saber when attacking.

6.3. SABERS TRAINING SCENE

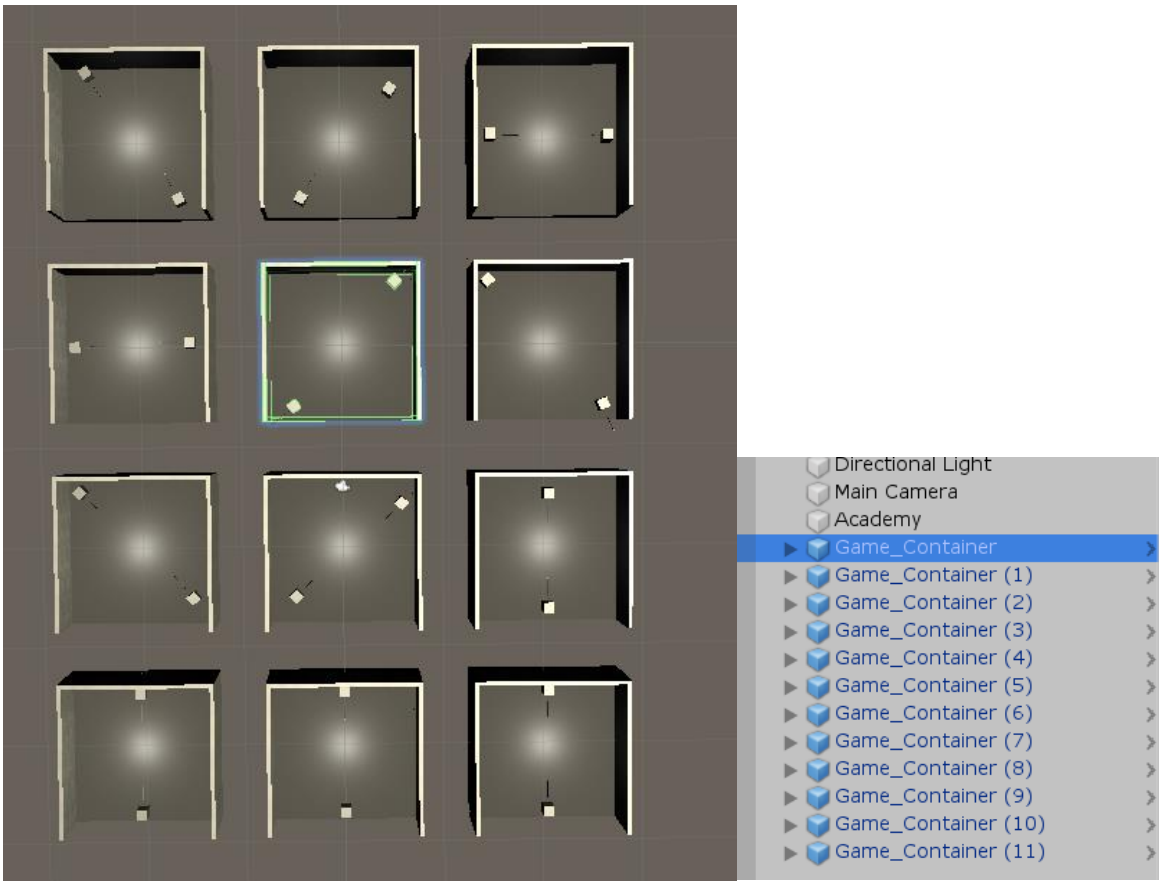


Figure 36: Sabers training Scene and gameobjects on the menu.

Similar to the previous training Scene where we've seen there are multiple copies of the main Game_Container object, here we have them for the same reason, this time we can appreciate that the initial position and rotation of the agents vary, this makes for a more robust training experience.

Same as before the academy GameObject is controlling both of the brains, as an important note even though both of these brains get trained, we only use one of them for playing purposes, unlike in the chasers playing scene.

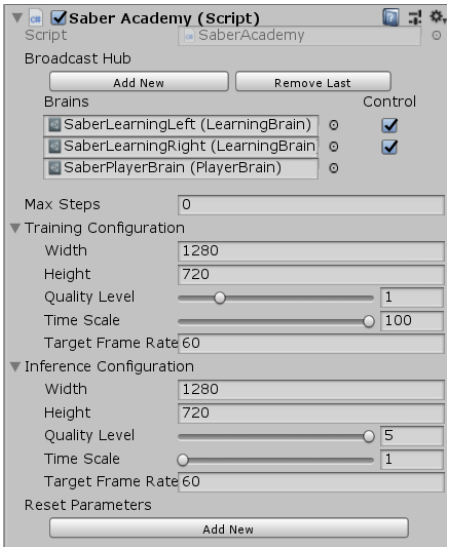


Figure 37: Academy gameobject controling both brains.

7. RPG MAIN SCENE

This is possibly the most complex of the 3 Scenes, as well as the most realistic in terms of how a real approach to a video game training Scene would look like. Firstly, we will explore its GameObjects.

7.1. THE TERRAIN.

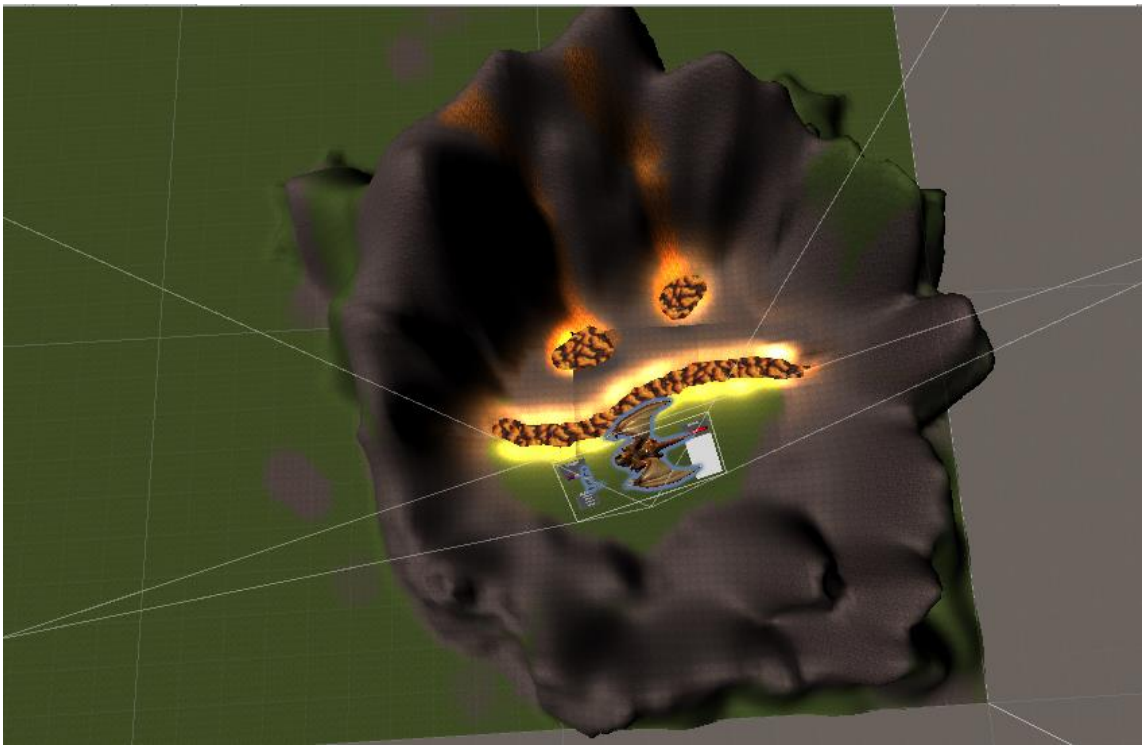


Figure 38: Birds eye view of the Scenes Terrain

Starting with the terrain we can see this time it's not just a box or a big ball, this sort of mountain ridge/volcano was made using unity **terrain brush** and element to give it a little bit more of a unique feel to it, just like in classic RPGs. It's just decorative but looks better than the other ones, even though a little texture work would be appreciated.

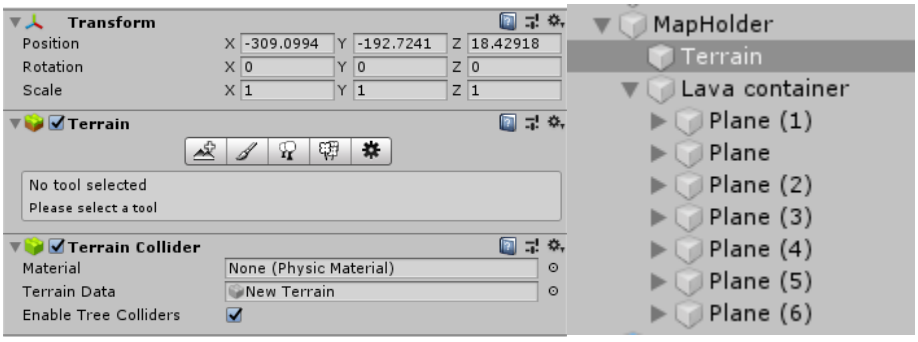


Figure 39: Terrain GameObject and menu references.

There are some planes added with a flowing lava texture that its offsetting constantly simulating flowing lava.

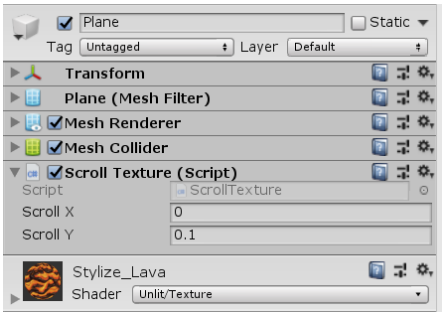


Figure 40: A lava plane

The Scroll Texture script is very simple:

We just get a reference to the renderer component and the materials mainTextureOffset and replace it by a value that increases with time, (textures with values bigger than 1 in the offset just go back to its original position).

This makes the illusion that the lava is flowing in the river giving it more of a "dangerous" feel, we also added some point light to the lava planes to make them give a little more light to the scene.

7.2. THE GUI

In the RPG case there is a particularity with the GUI, there are 2 of them.

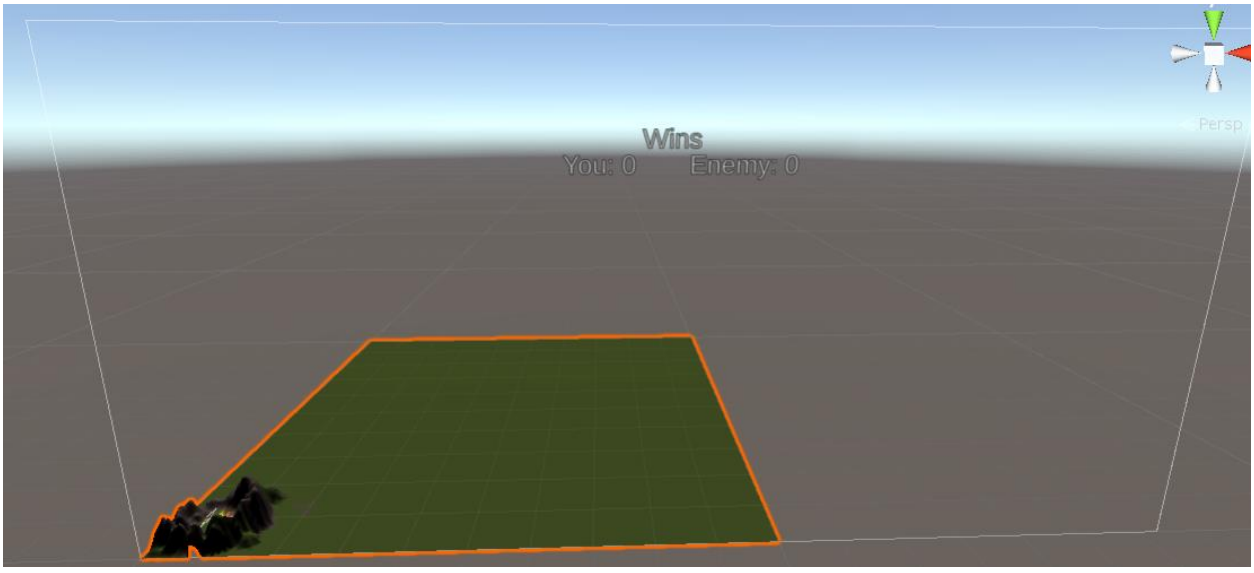


Figure 41: RPG external GUI

Fir
no

Figure 42: The RPG GUI

This is the RPG GUI which is a little more interesting it's made up of 4 main elements and it's the entry point for the games code via the menu.

Firstly, there are 2 HP bars one for the player and one for the Boss and also a mana bar for the player.

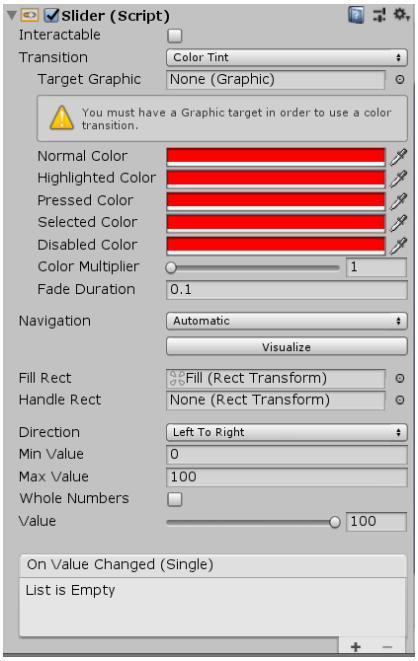


Figure 44: A slider object representing the players HP

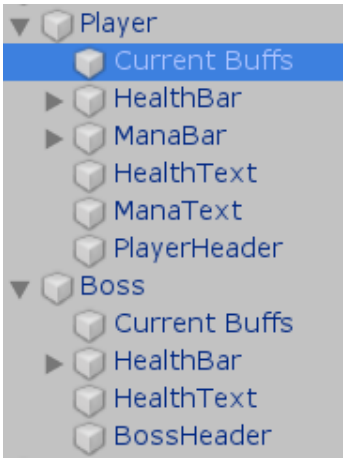


Figure 43: The player and Boss Healthbar window objects.

These are simply recycled slider objects (figure44), just like you would see in a volume or graphics slider but instead of using them as intended we programmatically modify the value to represent the current life or mana the entity has.

There is also some overlapping text that shows the numeric value of the HP and Mana total, just in case (figure 43).

Finally, there is a current buffs text (figure 43) which will show the active buffs for the player and boss.

Now for the chat window, its main use is to describe what is happening in the scene. With every action the player takes the boss takes another one. The chat takes care of letting the player know what is happening with every action taken.

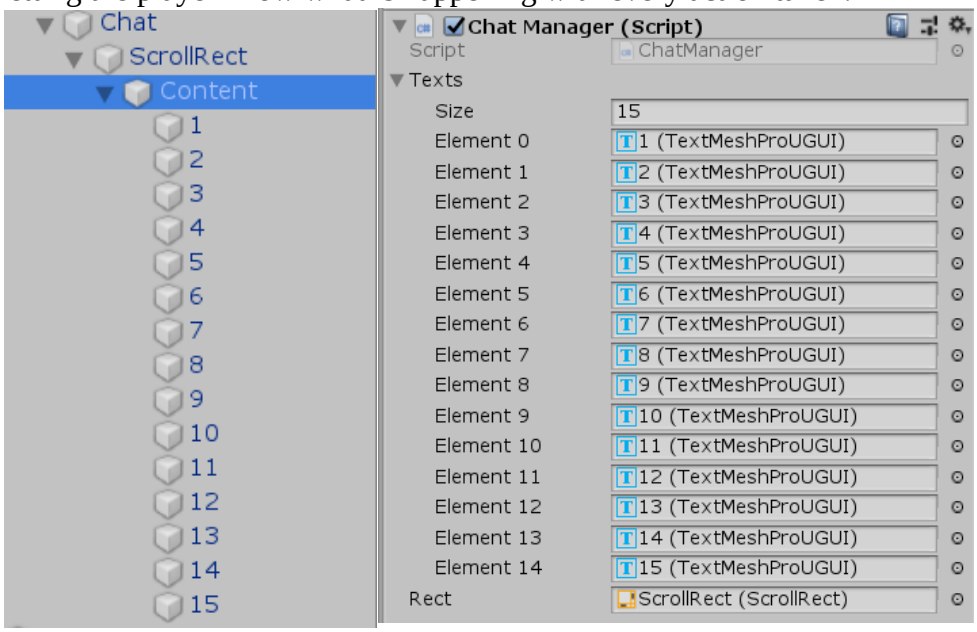


Figure 45: Chat GameObject and ChatManager.

The chat GameObject (figure 45) and manager are separated in this case but connected via references to the objects ScrollRect and the 15 possible entries that it has. Now lets explore the Chat Manager Script

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections.Generic;
using TMPro;

public class ChatManager: MonoBehaviour {
    [...]
```

Code 14: Chat Manager references.

are some imports we haven't seen before. **UI** is used to manage user interfaces; the chat is part of one so we need to have this here.
Collections is also necessary as we manage a collection of text objects (the chat entries). Finally, **TMPro** is a reference to text mesh pro, a text package that allows the creation of fancy text.

We also have a reference to the array of text objects, the scroll rect and a private list. At the start we set the chat to its highest possible value 1.

The addMesage function is responsible for adding new messages to the bottom of the chat. First, it checks if the total number of messages in the chat is 15, in that case it moves every message up 1 place and in this way “deletes” the oldest message and adds the new one to the bottom. If not 15 it simply adds a message to the list. Then it calls the **updateUI()** function.

This function is responsible for the chats position it moves the chat around to show the newest messages and leave older ones invisible out of the rendering square. This way we allow the player to scroll up and down the chat window to check older messages.

Now we will check the PlayerMenu object, which is made up of 3 different menus, the Main Menu, the MagicMenu and the ItemsMenu.

Figure 47:Player Menu GameObject

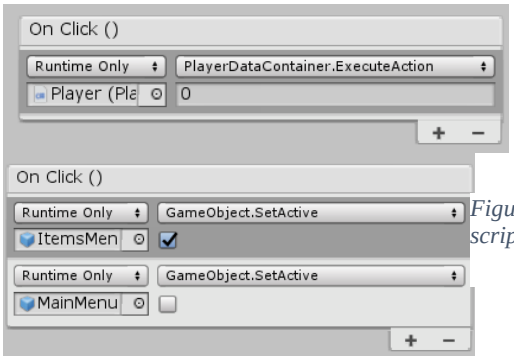


Figure 46:Buttons calling custom scripts and game object functions.

In the player menu (figures 47 & 48) the attack and defend options are actions per se and the button contains a call to the PlayerDataContainer to perform it.

Unity has Event Systems that are in charge of making calls to either GameObject functions, like setActive, or alternatively custom script functions like in this case. They can be detected by various sources like the press of a button or the change in a slider as. We can see some examples of this behavior in the figure 46

In the case of the Items and Magic options the outcome is simple. It set the current player menu to inactive (which renders it gone for all purposes but to set it as active again which brings it back) then sets the other menu (either magic or items) active and renders it visible again.

Figure 48:Main, Magic and Items menu

We will go through the options:

- Attack: A simple and free attack.
- Defend: A 2 turn buff that **increases defense** by 50%, also free.
- Fireball: A **damaging spell** that consumes a big chunk of the players mana.
- Magic Shield: A 4 turn buff that increases the defense **against magical attacks**, cost some mana.
- Ironclad: A 4 turn buff that increases the **defense against physical attacks**, cost some mana.
- Health Potion: A 1 time use item that heals the player to 100 hp.
- Mana Potion: A 1 time use item that heals the player to 100 mana.

All of these actions go through the ExecuteAction function of the player data container. We will check how it works now.

7.3. THE PLAYER GAMEOBJECT

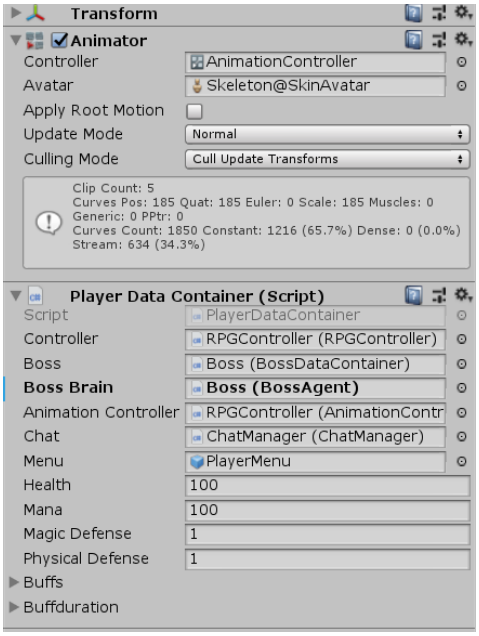


Figure 49:Player GameObject.

This GameObject is a little different to the other ones we've previously seen because it has animation!

Animating a 3D model is made outside of the unity editor, these models contain "bones" and "joints" which make movement possible.

We use an AnimationController which is responsible for the way in which the model will respond.

They also have custom 3D models obtained from the unit asset store.

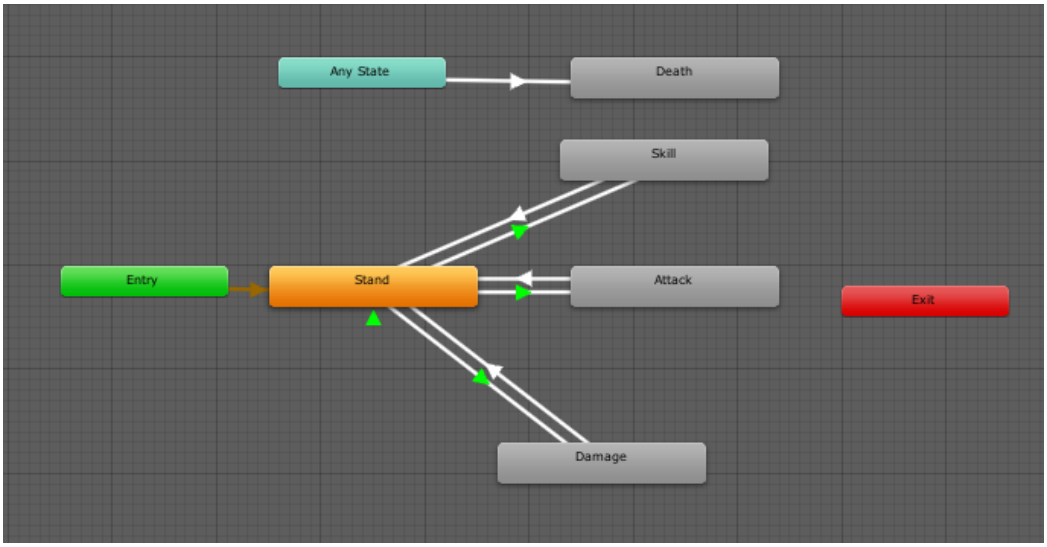


Figure 50:Player animation controller.

This is a simple animation controller; it is just a state-machine that controls the changes between the states of animation. How we change from one state to another is determined by the connections between them, here is how it works.

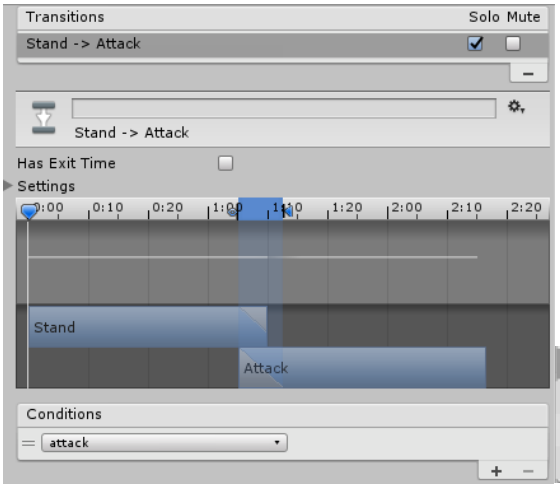


Figure 52:Animation controller transition.

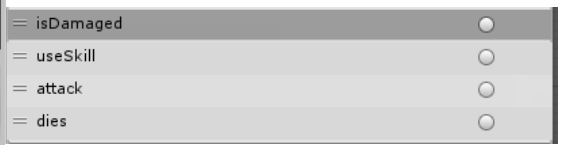


Figure 51:Animation triggers.

Firstly, the Entry box is where the animation starts whenever we load a scene or spawn this GameObject. The first connection is obligatory and the orange node is the default animation, it's a common occurrence to use an Idle or Stand animation for this.

The checkboxes are animation triggers (figure 51); we need to define them to be able to call them from a script later on, they are responsible for initiating transitions between states.

Here is an example of one, in this case we go from standing to performing the attack animation whenever the attack condition is met (the trigger is activated). It is a little more complex than this but it's not the purpose of this project to show how to animate 3D models.

We will now explore the PlayerDataContainer Script which is responsible for initiating the game flow.

```
using UnityEngine;
public class PlayerDataContainer : MonoBehaviour
{   [...]
    public string[] buffs = {"ironclad", "magic shield", "defend"};
    public int[] buffduration = {0,0,0};
    private string[] actions = {"attack", "fireball", "magicShield", "ironclad",
"defend", "healthPotion", "manaPotion"};
```

Code 15: PlayerDataContainer buff variables.

We've touched on imports and variables a lot so we'll just talk about the last 3.
Buffs is all the possible buff the player can have active.
The **buff duration** is responsible for counting how any turns are left of a certain buff.
Actions contains every possible action the player can take.

As we've seen in the previous section the ExecuteAction function is the entry point for the game flow, it receives the action to be executed and handles it. We see two functions that run first thing, **applyBuffs** and **reduceBuffDurations,** they contain the logic for buff handling.
We then print the action to the chat window and request an action from the boss. RequestDecision is an MLAgents function we will get to it when exploring the boss agent in detail, but in essence it will call this agents ExecuteAction function.

We then update the boss (in case it has buffed itself), and, Finally, we perform the action that the player chose, most of them are simple and are made up of the following sub actions; we ask the animation controller for the actions movement, apply damage to the boss or buff ourselves and spend mana or spend an item. We then call the update boss and update player functions to reflect the changes performed by the action.

```
    public void ExecuteAction (int actionId){
        applyBuffs();
        reduceBuffDuration();
        string action = actions[actionId];
        chat.addMesage("You used "+ action);
        bossBrain.RequestDecision();
        boss.updateBoss();
      switch (action){
            case "attack":
                animationController.PlayerAnimations("attack");
                boss.applyDamage(30,0);
            break;
            [...]
        }
        boss.updateBoss();
        updatePlayer();
    }
```

Code 16:PlayerDataContainer ExecuteAction function.

Now for the ancillary functions:
UpdateBuffs adds the indicated number of turns to the buffduration array for the selected buff.

applyBuffs updates the players defense values according to the current state of the buffDuration array. All defenses are increased by +0.5 adding to the base value of 1 we then will divide the damage taken by the player by this amount, depending if its magical or physical.

applyDamage reduces the health value of the player by the specified amount taking the players defense into consideration, in case the player dies it also calls the death animation if its health hits 0, then hides the menu and adds a message to the chat letting you know you've lost.

updatePlayer changes the values of the GUI texts, health bars, mana bars and buffs for the player, this is done via the RPGController, we'll touch on this GameObject later.

Finally, **reduceBuffDuration** is called once at the start of the turn to reduce the buff duration by 1 turn if it isn't already 0.

7.4. THE BOSS GAMEOBJECT:

Similar to the player GameObject there is a data container for the boss too, but now we can see there is also a **BossAgent**. This component, as it is the case for all the previous ones, contains the functions necessary for the agent to train.

There is also a noteworthy mention that I would like to make here, working with early version of tools as it is the case with this package has its risks, in this case the risk paid off but not without taking its toll.

This agent was supposed to be trained with a memory buffer component (recurrent neural network) to "remember" the player actions and its consequences. This way it would know how to respond if the player took X action by evaluating the context.

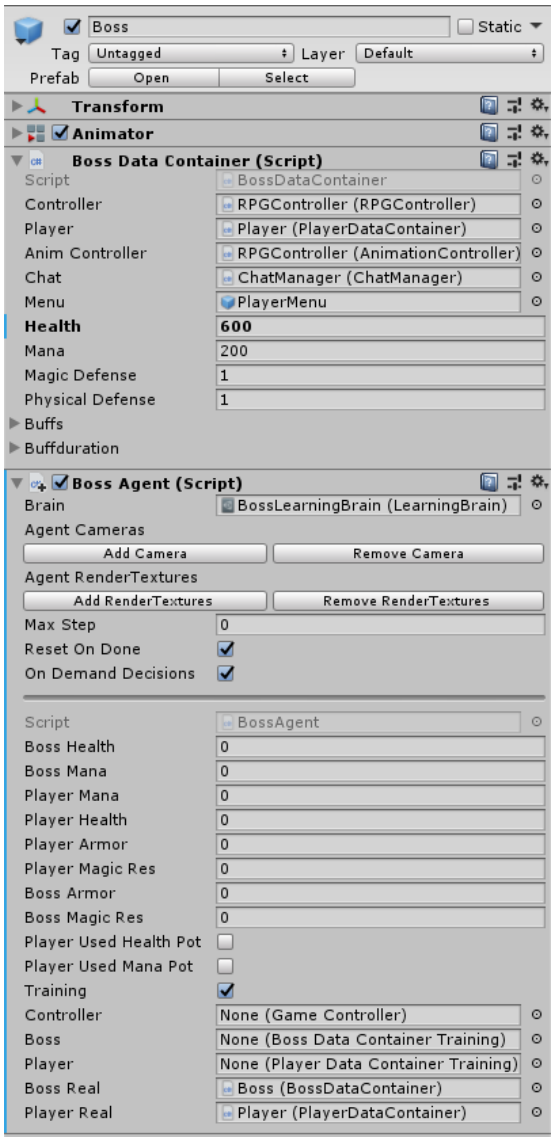


Figure 53:Boss GameObject

What happened was that whenever the memory component was active the environment was unable to train, causing the console to crash, thus leaving us with no option but to ditch this type of training altogether.

Anyway, with that out of the way, we can see that the boss data container (figure 53) is pretty similar to the one in the players game object.

We can see all the references to the different scene managers (chatManager, RPGContoller and animationController) as well as the buff and buffdurations etc…

As for the BossAgent you can see it has two empty references to the Boss and Player GameObejcts, this is used in the training Scene and it is a necessary evil.

We will now see the BossData Container Code as well as the BossAgent.

Note that On Demand Decisions is checked, this is because we want the boss to take action only when the playes has chosen theirs, this way it can benefit from knowing which action the player is going to take and act accordingly.

```
public class BossDataContainer: MonoBehaviour{
    [...]
    public string[] buffs = {"omnishield"};
    public int[] buffduration = {0};
    private string[] actions = {"attack", "magicBlast", "omniShield"};
```

Code 17: BossDataContainer buffs and actions.

Just like in the player container script we have some references to some controllers as well as the abilities, in this case there is only 1 buff and 3 options the boss can take.

The bosses **execute action** has 3 outcomes, “attack”, “magicBlast” and “omniShield”, they execute the corresponding actions. It also takes into account the amount of mana the boss has, depending on how much it has left it allows it only to perform certain actions.

It’s important to note is that the boss has 250 mana and not 100 like the player does. That is why its abilities are more expensive.

```
    public void ExecuteAction (int actionId){
        string action = actions[actionId];
        reduceBuffDuration();
        applyBuffs();
        if(mana >= 50){
            switch (action){
            [...]
        }
        chat.addMesage("Boss used "+ action);
        }else if(mana < 50 && mana >= 25){
            [...]
        }else{
            [...]
      }
        applyBuffs();
        player.updatePlayer();
    }
```

Code 18: Boss Execute action.

In case the boss selects an action that it can’t do because of mana limitations, we specify in the chat that the boss is performing a different action that the one it wants to. Since we can’t control the output of the network, we have to control it this way.

updateBuffs, **updateBoss**, **applyBuffs** and **reduceBuffDuration** have been omitted due to being the same as the players ones.

The main difference between the apply damage functions is that 3 out of 10 times the player can score a **critical hit** against the boss and deal double damage.

We’ll now check out the BossAgent Script.

```
public class BossAgent: Agent{
    //Variables
    [...]

    //training enviroment;
    public BossDataContainerTraining boss;
    public PlayerDataContainerTraining player;

    //playing enviroment;
    public BossDataContainer bossReal;
    public PlayerDataContainer playerReal;
```

Code 19: BossAgent initial distinction between scenes.

We have some variables that will be set in the Start method to mimic the ones in the data container, also as it was commented before there are **copies of the Data containers made for training** that weren't referenced in the GameObject since we are not in the training scene.

As for the **AgentReset** we have to make a distinction in case we are training or not here as well, checking which of the variables aren't assigned before resetting.

The same happens in the **CollectObservations,** we need to check which case we are in before assigning.

```
     public override void CollectObservations(){
        if(boss != null){
            [...]
        }
        if(bossReal != null){
            [...]
        }
             [...]
List<float> observations = new List<float> {bossHealth, bossMana, bossMagicRes, b
ossArmor, playerMana, playerHealth, playerArmor, playerMagicRes};
        AddVectorObs(observations);
        AddVectorObs(playerUsedHealthPot);
        AddVectorObs(playerUsedManaPot);
    }
```

Code 20: BossAgent CollectObservations function.

We add 8 floats and 2 Booleans as our observations containing the different parameters, health, mana, armor and magic resistance for both the boss and the player as well as if the player has used its health potion and mana potion or not.

```
public override void AgentAction(float[] vectorAction, string textAction){
    if(boss != null){
        boss.ExecuteAction(Mathf.FloorToInt(vectorAction[0]));
    }else{
        bossReal.ExecuteAction(Mathf.FloorToInt(vectorAction[0]));
    }
    if(playerHealth <= 0 || bossHealth <= 0){
        [...]
        Done();
    }else{
        SetReward(CalculateReward());
    }
}
```

Code 21: BossAgent AgentAction function

This code (code 21) is ran every time the MLAgents function **requestDesition()** is invoked for the bossBrain., in our case it will be called once every time the player takes action.

We then call the executeAction with the selected action given by the vectorAction, this determines that the output of the neural network will be 1 element that ranges [1,3] and we floor it to the nearest one.

We then proceed to check for the ending condition, that either the boss or the players HP reaches 0. If not, we calculate the reward.

The **reward calculation** is a pretty simple formula we get the normalized amount between the players hp and the bosses in range [-1,1].

```
((bossHealth/600) - (playerHealth/100)
```

7.5. RPG AND ANIMATION CONTROLLERS

We use external scripts (not attached to the gameObject that performs the action) to perform actions that aren't GameObject specific and more about the scene as a whole.

This type of game objects lacks components, they only have a transform and some scripts and are responsible for coordinating behaviours.

In the case of the **animation controller** its very simple it is just an entry point for executing animation triggers, we just pass the name of the trigger we want to run and it will call the animator for us.

This could be done just by calling the **SetTrigger** function inside of any of the other scripts directly by getting a reference to the animator component, but this way we keep everything related to animation in the same place and you can find how many animator components are in the scene.

There is also the **RPG controller** which contains some auxiliary functions that update the Scene to let the player knows about what happening while they play and controls that no inadequate actions are taken.

```
public Button fireballButton;
[...]
public Slider playerHealthBar;
[...]
public TextMeshProUGUI playerBuffs;
[...]
```

Code 22: RPG controller variables.

Firstly, we can see some references to the button objects that are limited resources, either mana dependent or one time use like health and mana potions. Then we can see some references to the health and mana bars and texts.

The **UpdateBoss** function is in charge of modifying the Boss health and buff values to whatever is passed to it. It is called from the main BossDataContainer function multiple times.

The **UpdatePlayer** function does a similar thing as the previous one but with the players data. It also checks if we are in the training or playing scene and if we are in the playing scene calls the **DisableButtons** function.

Which checks the players current mana and in case it is less than a certain threshold it disables the button rendering the player unable to use that ability.

The **DisablePotionButton** function receives which type of button it must disable and does so, it also changes the text to reflect this change.

8. RPG TRAINING SCENE

This particular case differs a quite a bit from the other ones we've seen previously, hence its own chapter.

The prefab object we use for training is not the same as the one in the playing scene, as you could've expected we use a variant of the **playerDataContainer** and **bossDataContainer** Scripts, even though they are pretty similar we need to use this approach because of the interaction with the academy and how it processes the "on demand" type of decisions.

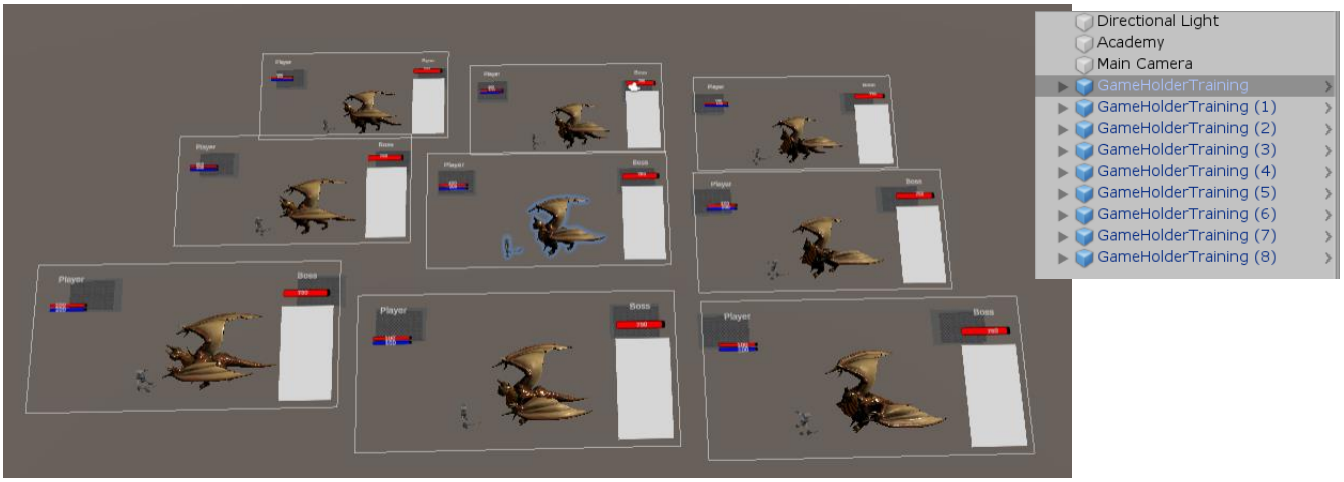


Figure 54: RPG training Scene.

Just like in every other training scene we start with multiple copies of the same Game for the initial randomization of the Neural network. The academy (figure 55) object will also have control of both Agents, but in this Scene the **player brain takes no part in the training** since it is just a fake and it takes a random action no matter what.

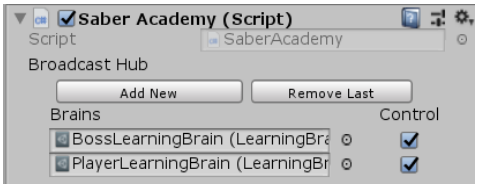


Figure 56: RPG training academy

Figure 55: GameHolderTraining

We will now go through the differences between the training and the playing data container scripts.

8.1. PLAYER DATA CONTAINER TRAINING:

The major difference is that the **ExecuteAction** function is not entered via menu and player input this time it is selected randomly in the PlayerAgent Script.

Every action the player can take is preceded by a check to see if it has enough mana to perform the action, in case it doesn't it performs the "attack" action instead (code 23).

```
      public void ExecuteAction (int actionId){

        [...]
        switch (action){
        [...]
            case "fireball":
            if(mana >= 30){
        [...]
            }else{
animationController.PlayerAnimations("attack");
chat.addMesage("You tried to use "+ action +" but no mana so attack instead");
        [...]
            case "healthPotion":
            if(!healthPotUsed){
                 health = 100;
                 healthPotUsed = true;
            }else{
chat.addMesage("You tried to use "+ action +" but you're out so attack instead");
        [...]
```

Code 23:PlayerDataContainerTraining ExecuteAction function

There is also a special check for health and mana potions since they can only be used once we had to add a flag that indicates if it had been consumed.

We don't limit how early or late it can be consumed or if it will be consumed at all, so the boss can kill the player without having consumed both potions or the player can consume them at full hp or mana.

The rest of the functions are an exact copy of the playerDataContainer.

8.2. PLAYER AGENT

We will now go through the **PlayerAgent** code and the similarities with the BossAgent we saw in the playing scene

The variables are the same except for the **nextActionReady** that determines when the Agent can make a new decision, albeit a random one in this case. GameController is also missing since this is a training scene.

```
    public override void AgentReset(){
        if(player != null){
            [...]
        }
            [...]
        playerUsedManaPot = false;
        playerUsedHealthPot = false;
}
```

Code 24: AgentReset function

The **AgentReset** function (code 24) resets the player stats to its initial state to restart the fight, pretty similar behaviour to the boss, except for the playerUsedXXXXPot flags that control if the player can use a certain potion or not.

collect observations has been omitted since it is an exact copy of- the boss one and it's irrelevant since we don't plan to train this agent.

```
    public override void AgentAction(float[] vectorAction, string textAction){
        if(!playerUsedHealthPot && playerHealth < 30){
            [...]
        }else if(!playerUsedManaPot && playerMana < 20){
            [...]
        }else if(Random.Range(1,100) >  ((bossHealth/6) - (playerHealth))){
            aggresiveApproach();
        }else
            defensiveApproach();
        SetReward(CalculateReward());
        if(playerHealth <= 0 || bossHealth <= 0){
        Done();
        }
    }
```

Code 25: PlayerAgent AgentAction

This function is in charge of making the player decisions. It can take one of two approaches depending on the current state of the game. If the player has more % HP than the boss does it always takes the aggressive approach, if that is not the case then the bigger the HP % difference between the boss and the player the more likely it is to take the defensive approach.

The **aggressive approach** casts fireball if the mana allows for it, otherwise it attacks.

The **defensive approach** is a little more complicated, it first checks if the defensive buffs are applied, in case they are not and the player has mana it first tries to buff its magic defense, then its physical defense, if it hasn't enough mana it defends, and if all the defenses are in place it just attacks.

Reward Calculation is the same as in the Boss and it doesn't matter to us since the Agent is not training.

We check every frame in the **Update()** function if the next action is ready, meaning that all other operations have concluded and the player would select its next action. In this moment we call the **RequestDecision** for the **PlayerAgent** which in turn would call the Execute action.

Even though the PlayerAgent is not meant for training its necessary to replace the menu entry point and enables the boss to train against a pseudorandom input.

8.3. THE BOSS GAMEOBJECT

The only difference for the BossAgent in the training Scene is that the references are updated for this scene.

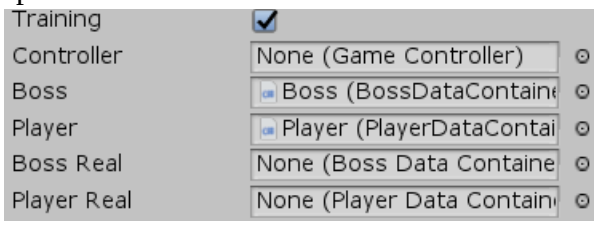


Figure 57: The BossAgent training GameObject Script dfferences

The rest of the Agent Script must be same since it needs to be able to perform in the playing scene. Now we will see the differences between the **BossDataContainers** for playing and training:

There's is one difference in the **Execute action** function which is that the **updatePlayer** function call is missing from the training data container. This is due to the fact that it updates the active buffs string, since it's only useful for player visuals we do not do so in this case.

In this case the reference to the menu.SetActive function is missing from the **apply damage** function, this again is due to the menu object being missing in the training scene.

updateDefenses, **updateBuffs**, **applyBuffs**, **updateBoss** and **reduceBuffDuration** are exactly the same as in the original version of the script and for that reason they will be omitted.

As we can see due to the menu object being missing some things had to be tweaked from the original playing scene main GameObject prefab, like the fact that there isn't a Main menu anymore, and the RPG controller calls to deactivate buttons do nothing anymore.

9. TRAINING AGENTS

Now that we have explained how everything was developed, we will show the ml-agents package interacting with our training scenes and what we get as a result from the training.

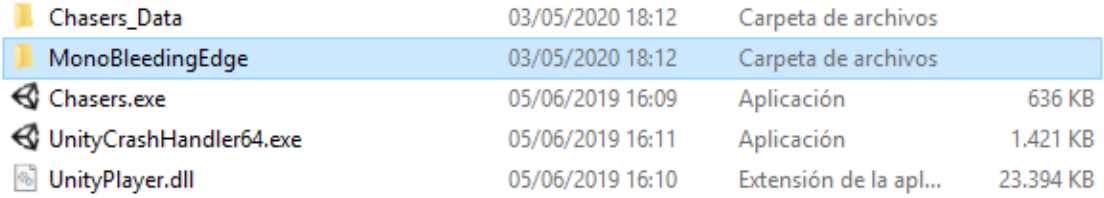


Figure 58: A compiled training Scene for windows

This is what a compiled scene looks like on a typical windows system, to interact with it we can use our python environment with the ML-Agents package to do so.

Figure 59: ML-Agents package.

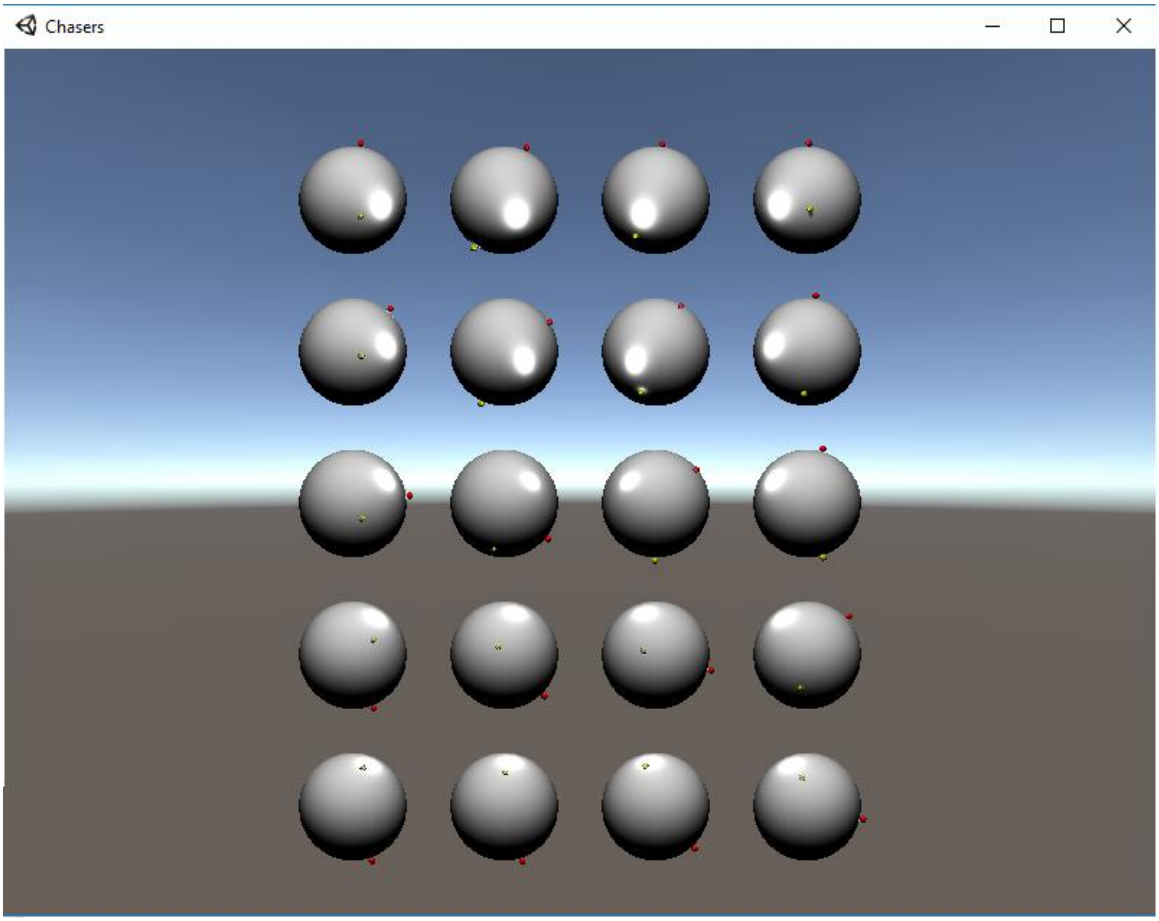


Figure 60: Chasers Scene training.

This is typically what the training looks like (figure 60), in this case the spheres are moving around the planetoid objects in a random fashion (since this is a new training scene) determined by the initially generated neural network, while they learn they will have more reasonable patterns.

This is what we can see while it trains:

Figure 61. ml-agents training screen.

First, let's analyze the command that was given for this particular environment to train:

```
mlagents-learn config.yaml --env=Chasers/Chasers  --run-id=chaser0 --train --slow
```

- **mlagents-learn** is the base command for the environments to train and its ml-agent package dependent.

- **config.yaml** is a file can be found in the root where the command was run, actually this statement must be a path to it.

- **environment** is again a path, in this case to the .exe that contains the training scene.

- **run-id** is a unique name that must be given to this particular training instance, its purpose is to identify it in case something fails or the need to continue training this instance arises.

- The **train** flag indicates that the environment must train and generate a .NN file containing the resulting neural network.

- Finally, the **slow** flag is used in this particular scene only, this is because the chasers scene doesn't run on the unity physics system, it instead uses a custom behavioral script to simulate gravity.

This affects the training because of the technique it uses to accelerate the flow of time making multiple physics calculations and applying them faster skipping frames. This technique allows the training to last significantly less time.

Next, we can see the unity logo art and some information telling us how the training is configured and past that it tells us the trainer has found the academy game object and the two brains that are the ones that are going to be doing the training. We can also see the vector parameters that we talked about before.

Further down we can see the **Hyperparameters** that were used in the training for each for each of the brains, since we used the same config.yaml file all of our networks are going to be trained using the same parameters, we will be going into detail for each one of them further on.

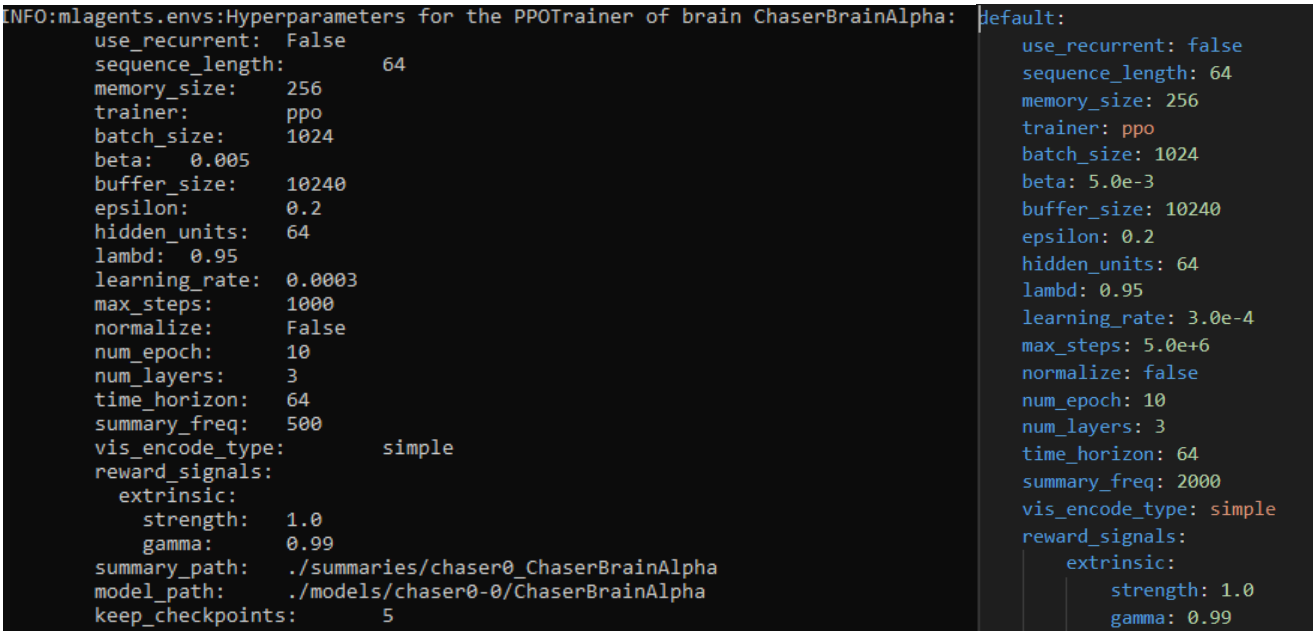


Figure 62: Hyperparameters used for training.

The parameters show here are chosen in the config.yaml file (right). We are using ppo that stands for **Proximal Policy Optimization**, according to OpenAI [21]:

"*Proximal Policy Optimization (PPO), which perform comparably or better than state-of-the-art approaches while being much simpler to implement and tune. PPO has become the default reinforcement learning algorithm at OpenAI because of its ease of use and good performance.*"

There is also a catch thought:

"*Policy gradient methods are fundamental to recent breakthroughs in using deep neural networks for control, from video games, to 3D locomotion, to Go. But getting good results via policy gradient methods is challenging because they are sensitive to the choice of stepsize — too small, and progress is hopelessly slow; too large and the signal is overwhelmed by the noise, or one might see catastrophic drops in performance. They also often have very poor sample efficiency, taking millions (or billions) of timesteps to learn simple tasks.*"

For this reason, we can justify the max steps chosen of 5 million. This also implies that the time taken to train the networks is going to be large, and more so if the training is made in real time and not accelerated one, the logs that were taken of the training make it pretty clear.

INFO:mlagents.trainers: chaser0: ChaserBrainAlpha: Step: 5000000. Time Elapsed: 217935.960 s Mean Reward: 156.024. Std of Reward: 37.530. Training.

INFO:mlagents.trainers: chaser0: RunnerLearningBrain: Step: 5000000. Time Elapsed: 217935.973 s Mean Reward: 14.176. Std of Reward: 38.739. Training.

60 and a half hours is the time that it took to train the chasers. That is two and a half days!

Knowing this let's explore the other two training scenes before going deeper into the hyperparameters.

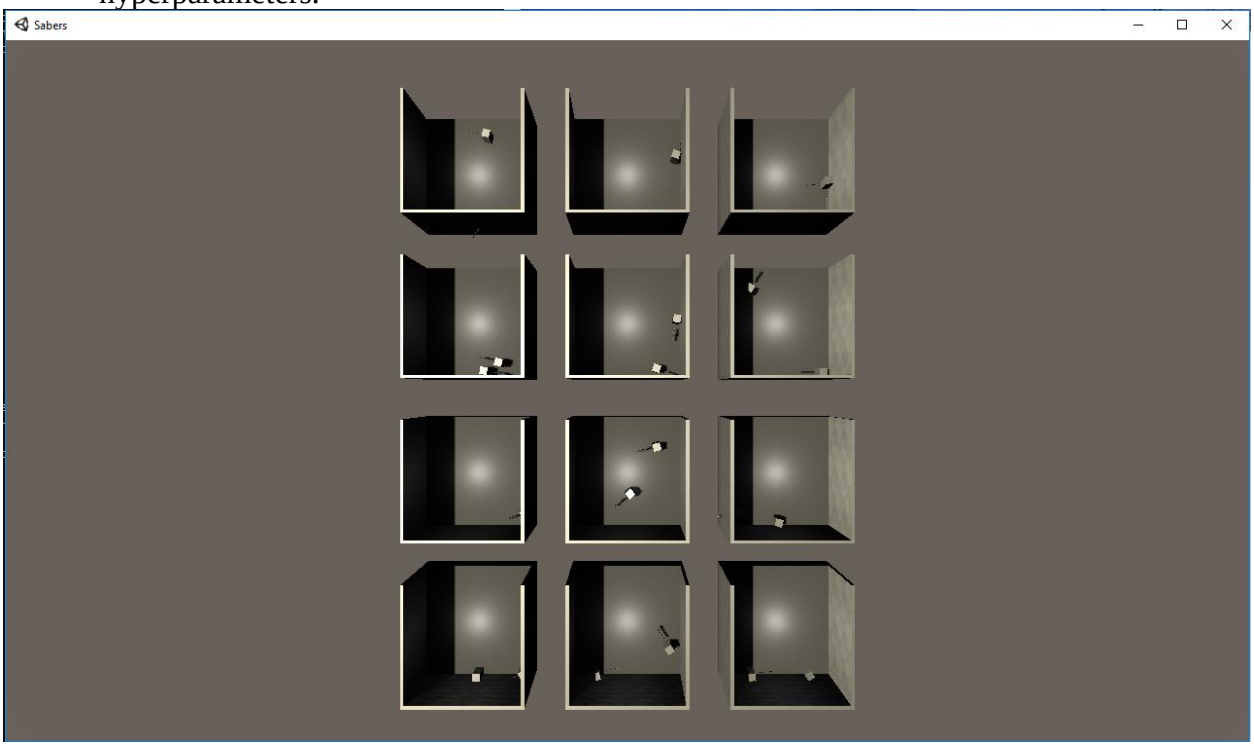


Figure 63:Sabers training Scene.

Figure 64: a training scene of 60 by 60 px of the RPG

```
(ml-agents) C:\Users\Magic Mike\Desktop\Training_Envs>mlagents-learn config.yaml --env=RPG/RPG
--run-id=rpg0 --train
```

```
(ml-agents) C:\Users\Magic Mike\Desktop\Training_Envs>mlagents-learn config.yaml --env=Sabers/Sabers
 --run-id=saber0 --train
```

We can see that neither one of the training statements make use of the slow training tag, we can see how much of a difference this make by checking the time it took to train them for the same number of steps.

INFO:mlagents.trainers: saber0: SaberLearningLeft: Step: 5000000. Time Elapsed: 64047.417 s Mean Reward: 9972.849. Std of Reward: 571.975. Training.

INFO:mlagents.trainers: saber0: SaberLearningRight: Step: 5000000. Time Elapsed: 64047.424 s Mean Reward: 9994.518. Std of Reward: 330.604. Training.

In the case of the sabers it took just shy of 18 hours to train them it's more than 3 times faster since it uses only physics and not so many calculations or intermediate actions. This is possibly the best-case scenario for the fast training mode.

INFO:mlagents.trainers: rpg0: BossLearningBrain: Step: 5000000. Time Elapsed: 110913.412 s Mean Reward: -1.504. Std of Reward: 1.791. Training.

INFO:mlagents.trainers: rpg0: PlayerLearningBrain: Step: 5000000. Time Elapsed: 110913.419 s Mean Reward: 1.122. Std of Reward: 2.351. Training.

In the RPG scene case, it took 31 hours. Even though not as good as the Sabers it's still took half the time than the Chasers did, and taking into account that this scene takes a lot more resources every frame since it has custom models, terrain, plenty of code execution, on demand decisions, etc… to take care of it's a rather necessary time save.

Now that we know how the scenes train let’s check the hyperparameters and their use.

Setting	***Description***	***Applies to Trainer****
batch_size	The number of experiences in each iteration of gradient descent.	PPO, SAC, BC
beta	The strength of entropy regularization.	PPO
buffer_size	The number of experiences to collect before updating the policy model. In SAC, the max size of the experience buffer.	PPO, SAC
epsilon	Influences how rapidly the policy can evolve during training.	PPO
hidden_units	The number of units in the hidden layers of the neural network.	PPO, SAC, BC
lambd	The regularization parameter.	PPO
learning_rate	The initial learning rate for gradient descent.	PPO, SAC, BC
max_steps	The maximum number of simulation steps to run during a training session.	PPO, SAC, BC
memory_size	The size of the memory an agent must keep. Used for training with a recurrent neural network.	PPO, SAC, BC
normalize	Whether to automatically normalize observations.	PPO, SAC
num_epoch	The number of passes to make through the experience buffer when performing gradient descent optimization.	PPO
num_layers	The number of hidden layers in the neural network.	PPO, SAC, BC
pretraining	Use demonstrations to bootstrap the policy neural network. See	PPO, SAC
reward_signals	The reward signals used to train the policy. Enable Curiosity and GAIL here	PPO, SAC, BC
sequence_length	Defines how long the sequences of experiences must be while training. Only used for training with a recurrent neural network.	PPO, SAC, BC
summary_freq	How often, in steps, to save training statistics. This determines the number of data points shown by TensorBoard.	PPO, SAC, BC
time_horizon	How many steps of experience to collect per-agent before adding it to the experience buffer.	PPO, SAC, (online)BC
trainer	The type of training to perform: "ppo", "sac", "offline_bc" or "online_bc".	PPO, SAC, BC
use_recurrent	Train using a recurrent neural network.	PPO, SAC, BC

Table 2: Hyperparameters used for PPO

We will explain some of these more in detail:

- **hidden_units**: This is the number of nodes in each of the intermediate layers for the neural networks, in our case we chose 64, meaning each of the hidden layers contain that many neurons.
- **max_steps**: This is in essence how long the training is going to take and it is directly related to how well the agents are going to respond.
- **num_layer**: The number of hidden layers in the network.
- **summary_freq**: This is used by TensorBoard to draw the result of the training, we will see more later.
- **use_recurrent**: This allows us to use recurrent neural networks, these types of networks allows us to use memory to map result of actions taken by the network. This network was intended to be used for the RPG training scene but sadly the fact that this build was in alpha caused an unexpected bug to occur, which was promptly reported and never answered by the development team.

This is what a typical network would look like in our case:

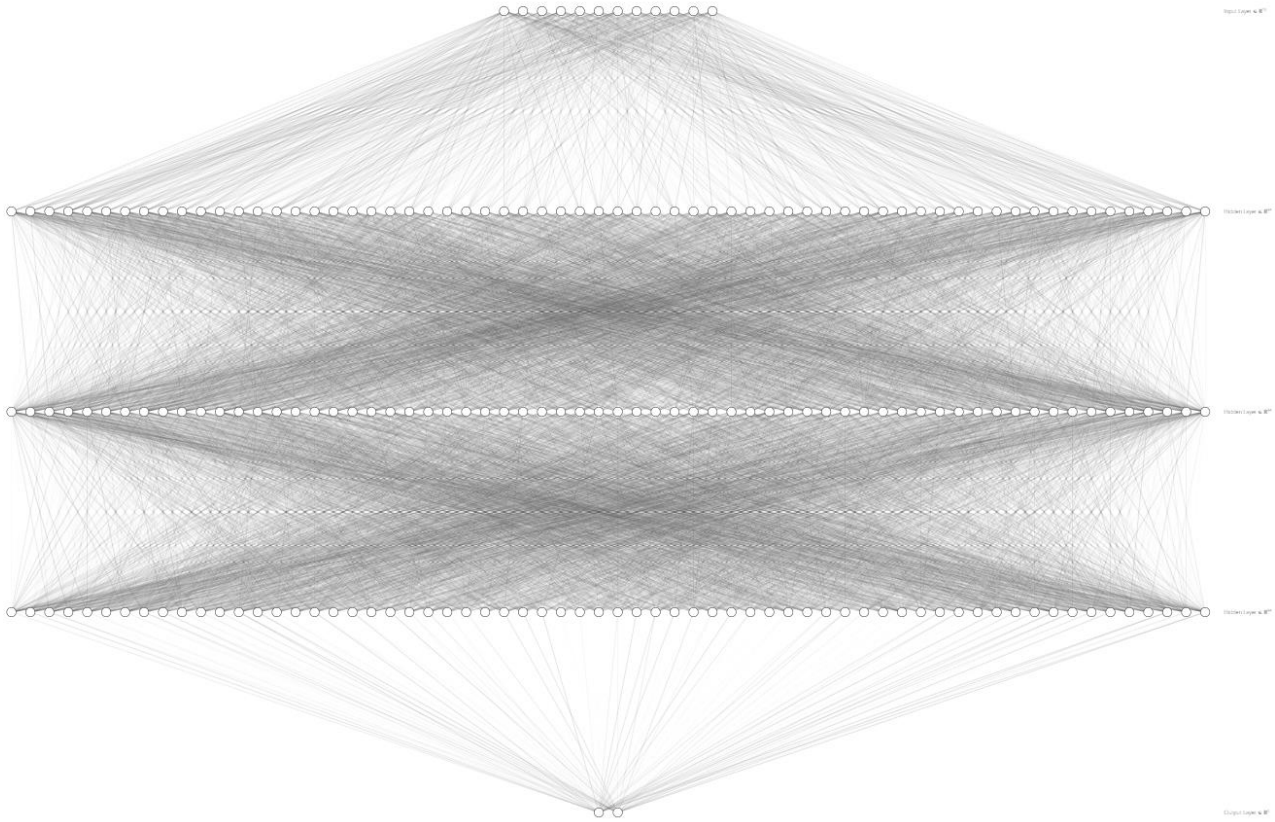


Figure 65: A 12 input, 2 output networks with 3 hidden layers and 64 neuros per layer. [14]

10. TYING IT ALL TOGHETHER

10.1. THE MAIN MENU SCENE.

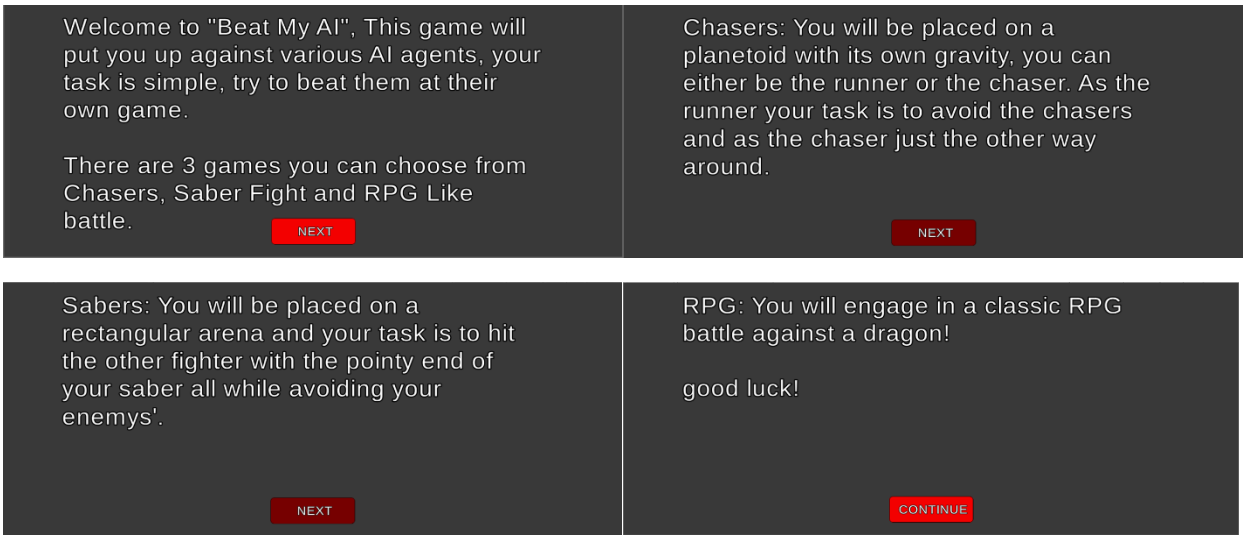


Figure 66: The Main menu introduction

Here we can see how the Game will start out, first it explains a little of what you're going to find in it in general, then scene by scene and it then shows you the scene selector.

Figure 67: Scene selection screen and button logic

Each one of these buttons will call a function to load a new scene using the menu options script (figure 67, right), this interacts with the game controller (yes the same we've been referencing along the project), whenever a Scene loads.

As we can see there is also a **DontDestroyOnLoad** Scene (figure 68), this is a special scene unity uses to preserve data between scene loads, the game controller object is the only thing that will persist in between scenes.

There is also an Info Button that just shows information about the developer.

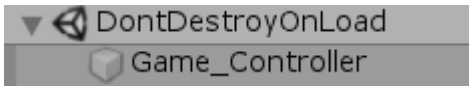


Figure 68: DontDestroyOnLoad Scene with Game_Controller object

Figure 69: The info container on main menu.

10.2. THE MENU OPTIONS:

```
      public class MenuOptions: MonoBehaviour

{
  public GameController controller;
  void OnEnable(){
  controller = GameObject.Find("Game_Controller").GetComponent<GameController>();
  }
    public void ChaserLoad(){
        controller.game_to_load = "Chaser";
        SceneManager.LoadScene(2);
    }
    [...]
}
```

Code 26:Menu options Script.

We use OnEnable (another unity function) instead of Start because the object is already in the scene when we load it. Since we can't get references to it while it's disabled we need to wait until its active to call the Find function to retrieve the game controller.

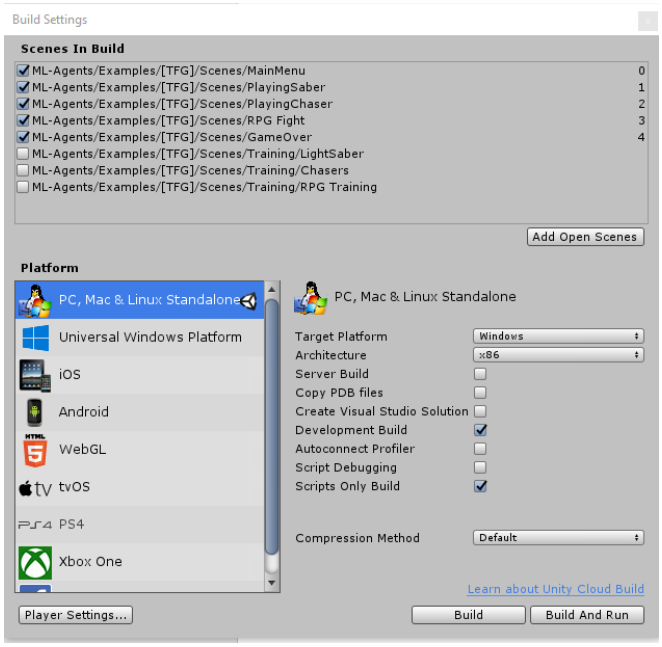


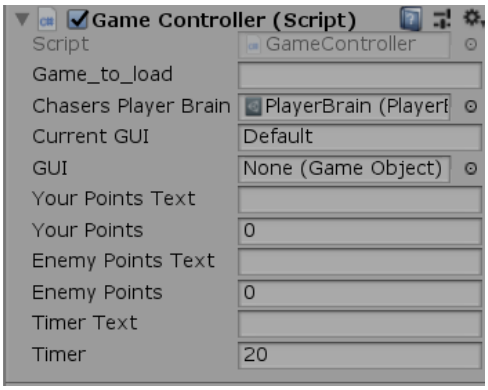
Figure 70: Unity Build Settings.

Whenever we select one of the scenes the functions XXXLoad() will tell the controller which scene is going to be loaded and then call a reference to the scene number. This is determined by the unity Build Settings.

The numbers on the right determine which number corresponds to which scene. It has multiple other options, like which platform to create a build for, the name of the game, resolution render pipelines…

10.3. THE GAME CONTROLLER

A game controller is one of the most important elements in a game, it controls everything that makes a game what it is, interaction between game scenes, controlling game data, passing info between game objects in different scenes, etc.



Here we can see some elements that are going to passed in between scenes. Such as the game to be loaded a reference to the player brain for the saber's scene, some points a timer object…

We need to use some new imports like **Networking**, **SceneManagement** and **System**, there is also a camera reference, this is so we can center the camera on the player in the chaser's scene. This is due to the possibility to play as chaser or as runner in this scene.

```
void Awake(){
    UnityEngine.Object.DontDestroyOnLoad(this.gameObject)
    SceneManager.sceneLoaded += OnSceneLoaded;
}
```

Code 27: GameController Awake function

The **Awake** function is another unity function, it does the same as the Start() function, but it runs right before it, this is useful in case you want to do some operations before some other start functions. In this case we are adding the game object to the don't destroy on load scene (and in this case creating it) and adding the **OnSceneLoaded** flag to the scene manager, this is necessary as it is used to control whenever we change scenes.

We use the **update** function that executes every frame to check if we are in the chases scene, in that case we reduce the timer by the **deltaTime** amount (the time that has passed from the previous frame) every frame and update the timer text that we've seen in the chasers playing Scene.

On chaser or runner **scene load** we Firstly, **find the main camera** in the scene as well as the corresponding GameObject (either chaser or saber), then they set the camera as a child of this game object, this means that the camera performs the same movement as the parent effectively following them.

We also set the localPosition to 10 units above the parent game object, giving us a top down view of the scene. Finally, we make use of the player Brain to control the main game object, and set the flags and timer to the correct values.

Loading the other 2 games is much easier, as they have no variable player selection, we just setup the training flags to false and get a reference to the GUI we are going to be updating. In case of the RPG we also setup a reference to this GameController object in the boss GameObject as some updating operation come directly from this script.

```
    void OnSceneLoaded(Scene scene, LoadSceneMode mode){
        switch (game_to_load){
            case "Chaser":
                LoadChaser();
                   [...]
            case "Main Menu":

GameObject[] controllers = GameObject.FindGameObjectsWithTag("GameController");

foreach (GameObject controller in controllers){
      if(controller.GetComponent<GameController>().currentGUI == "Default"){
              Destroy(controller);
        }
  } [...]
```

Code 28: OnSceneLoaded in GameController.

This is the function that ties all together. **OnSceneLoaded** is once again a Unity specific function that is called whenever a scene has finished loading. As we saw before the Menu options script calls this controller and tells it which scene is about to be loaded.

This script just calls the corresponding function and also resets the GUI.

It is also responsible for making sure that there aren't multiple instances of the Controller, for this it checks whenever we reach the main menu once again (after finishing a game), finds the game controllers in the scene that have the "Default" GUI meaning it was just created and deletes (destroys) them. This is also the reason why we don't reference the controller directly in the menu options and find it on enable.

```
    private void SetupEndScreen(){
        string endText = "";
        if(yourPoints > enemyPoints){
            endText = "Congratulations you win!";
        }else{
            endText = "You lose, better luck next time >:)";
        }
GameObject.Find("Score text").GetComponent<TextMeshProUGUI>().text = "Your final
score was\n" + yourPoints.ToString()+" - "+ enemyPoints.ToString()+"\n"+endText;
    }
```

Code 29: SetupEndScreen function from GameController

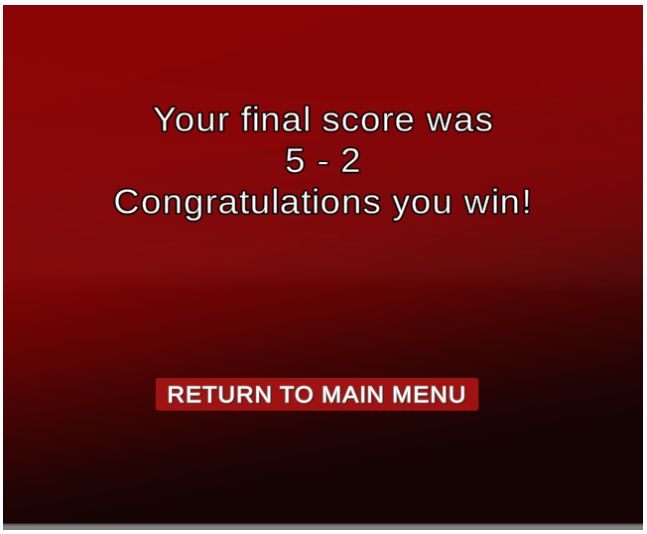


Figure 71: GameOverScene.

This function sets up what is going to be shown in the game over scene, here is an example of a finished game of sabers.

There is also a function in the **GameOver scene** that calls the reset to main menu in the controller, we will omit the script since we've seen plenty of similar examples already.

The **GUI selection** is just getting a reference to GUI object depending on the scene, in case of the RPG it has a particularity, since the scene resets every time it's **done()** we need to find it and also set up the correct values for the tally.

```
    public void IncreaseWin(string who){
        if(who == "enemy"){
            enemyPoints++;
GUI.transform.Find("Enemy Points").GetComponent<TextMeshProUGUI>().text = "Enemy:
 "+ enemyPoints.ToString();
        }
        if(who =="player"){
            [...]
        }
        if(GUI.name == "chasersGUI"){
            if(enemyPoints >= 3 || yourPoints >= 3){[...]}
        }else if(GUI.name == "SabersGUI"){
            if(enemyPoints >= 5 || yourPoints >= 5){[...]}
        }else if (GUI.name == "rpgGUI"){
            if(enemyPoints >= 3 || yourPoints >= 3){[...]}
        else{
                SceneManager.LoadScene(3);
            }}}
```

Code 30: IncreaseWin function from GameController.

This Script (code 30) is responsible for increasing the points to the correct person (the who variable) and updating the GUI, as well as, controlling if the game is over and calling the gameOver scene in that case

```
   private void SendResultToServer(int enemyPoints, int yourPoints, string game){
        SerializedData data = new SerializedData();
        data.enemyPoints = enemyPoints;
        data.game = game;
        data.yourPoints = yourPoints;
        if(yourPoints > enemyPoints)
            data.result = "win";
        else
            data.result = "lose";
        string url = "https://agube.lu/jserver/games";
        string jsonData = JsonUtility.ToJson(data);
        var request = new UnityWebRequest(url, "POST");
        byte[] bodyRaw = Encoding.UTF8.GetBytes(jsonData);
        request.uploadHandler = (UploadHandler) new UploadHandlerRaw(bodyRaw);
        request.downloadHandler = (DownloadHandler) new DownloadHandlerBuffer();
        request.SetRequestHeader("Content-Type", "application/json");
        request.Send();
    }
}

[Serializable]
public struct SerializedData
{
    public string game;
    public int yourPoints;
    public int enemyPoints;
    public string result;
}
```

Code 31: SendResultToServer function from game controller.

This is the communicator function between the game and the server where we log every result of every game played. On the bottom we see a struct of data that contains what is going to be sent to the server. We then encode the data using Json and send them via POST request to the Json server.

10.4. THE JSON SERVER

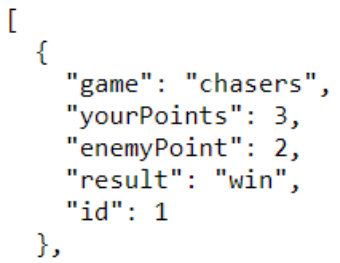


Figure 72:Json Server data [15]

Here is an example of how the json server stores the data we sent.

The json server offers a wrapper for a mongoDB database where we just send petitions via POST request to perform actions on the database.

These can be one of the main API calls:

```
GET    /games
GET    /games /1
POST   /games
PUT    /games /1
PATCH  /games /1
DELETE /games /1
```

These do what you would expect of them:

GET, returns the information on either every game if none is specified or, the one with the corresponding ID.

POST, creates a new entry for the corresponding item.

PUT and PATCH update the referenced object.

DELETE, erases the referenced object.

POST, PUT and PATCH need an application/json header to perform its operation.

10.5. THE DEPLOYMENT SERVER

We've explained how the Unity Build function works, we should also note that it allows for a HTML 5 build which we can the upload to any apache server

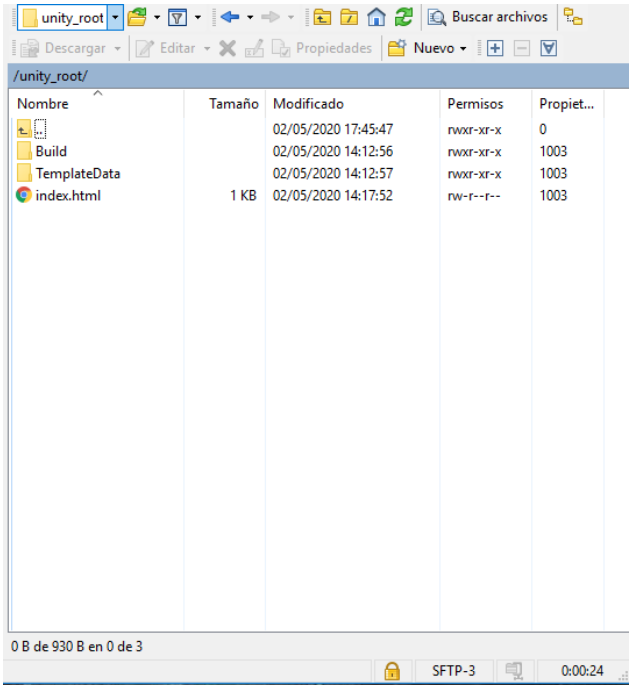


Figure 73:winSCP connecting to the server

Using a typical SCP connector we can access our servers folder, and using the Unity provided build files deployment is as easy as uploading, it even comes with an index.html file where the necessary connections to the game can be found.

The url where the project can be found is: https://agube.lu/unity/

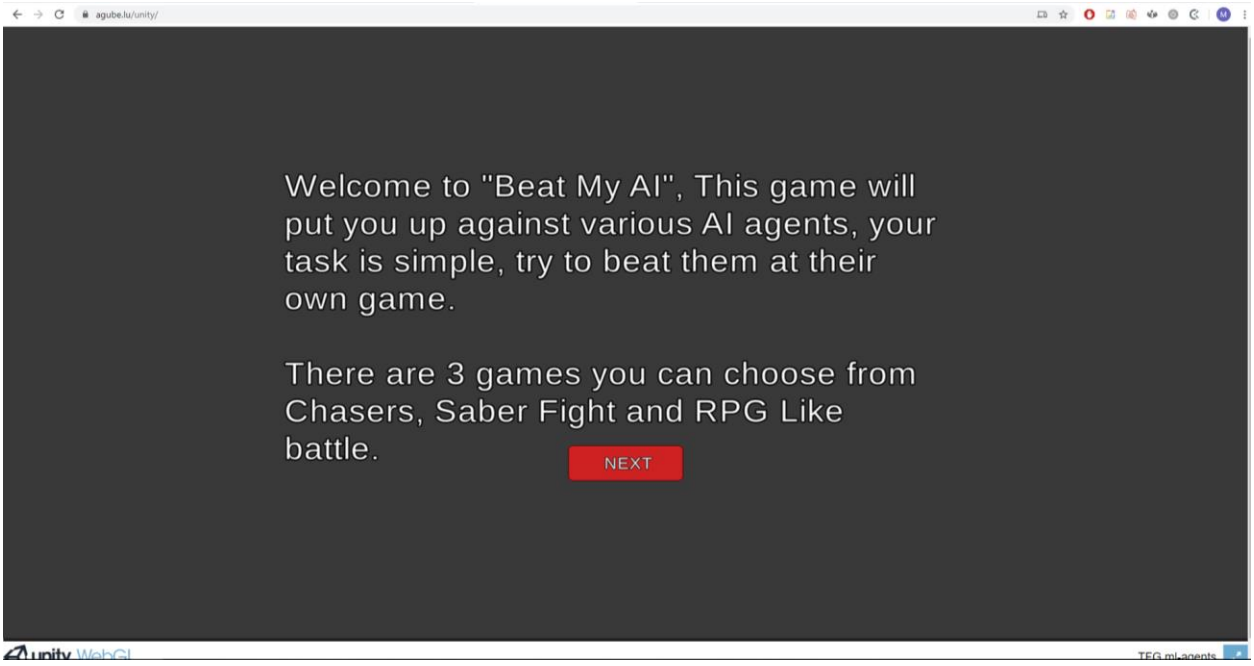


Figure 74: The game running in on a browser.

11. FIRST TRAINING RESULTS

11.1. AGENT BEHAVIOUR

As a first impression we concluded that the first training was a failure in the chaser scene, mediocre in the saber Scene and pretty successful in the RPG scene. This is probably due to the fact that the first two were training against another untrained agent, this makes the result a bit random and dependent on luck more than on the environment. We'll now explain what happened for each of the scenes.

For the **Chasers**, two things occurred, the runner agent learned to remain static and just take the loss, this is most likely due to the fact that the chaser agent actually learned properly how to intercept it and the runner just surrendered, this caused the chaser agent to actually unlearn what it probably learned and just go towards the runner initial position over and over always winning, and whenever the runner moves now it just doesn't expect it.

For the **Sabers** we tested both of the networks that come out of the training and even though they are not very good they can be considered actually a successful result as they try to defend themselves and orient its saber towards the other cube.

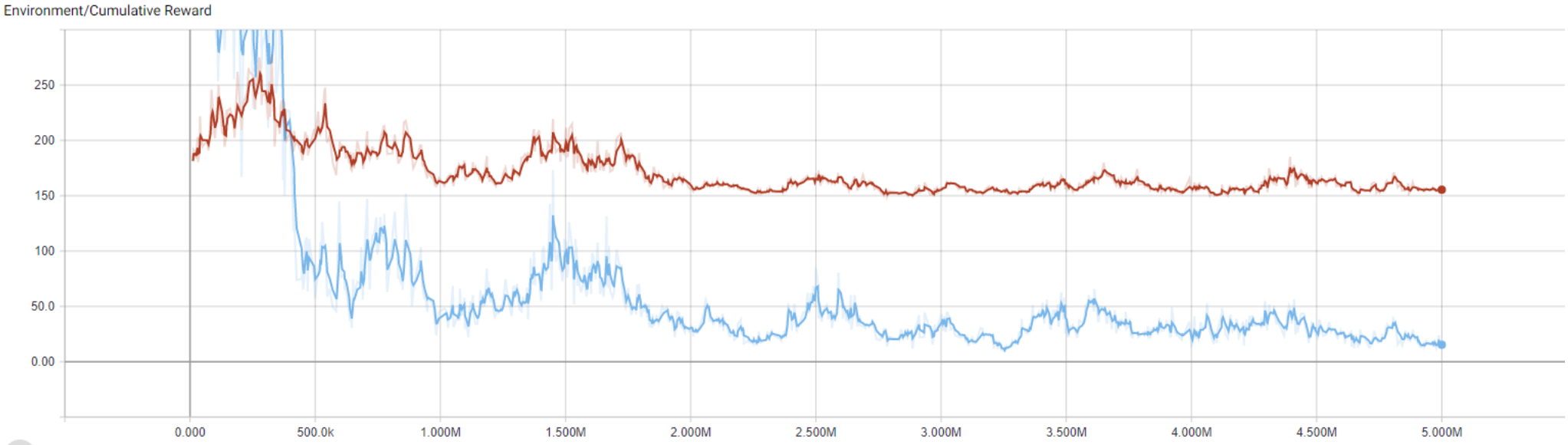
Finally, the most interesting result and a big success is the **RPG** boss agent. Since it has been trained with a pseudorandom and manually programmed agent it has learned properly, I might even say too well.

Since the training is the result of rewards given and we rewarded the RPG agent based on how low the HP of the Enemy is, it has learned to constantly attack until the player is low on hp and has to use its health potion (this is a programmed behavior and will always occur) and then again until its low, then the Boss will cast protective magic until it can't do so no more because of its closeness to dying, and then end the fight. In some cases, it waits too long for this to happen and ends up losing because of its greed.

11.2. TENSOR BOARD EVOLUTION.

Chasers

We can see that the runner (in blue) starts out pretty good but influenced by the improvement of the other one starts dropping in performance drastically and from then on it suffers a correlation that links up pretty closely, this corroborates what we previously stated that the chaser unlearned due to the runner agent giving up.

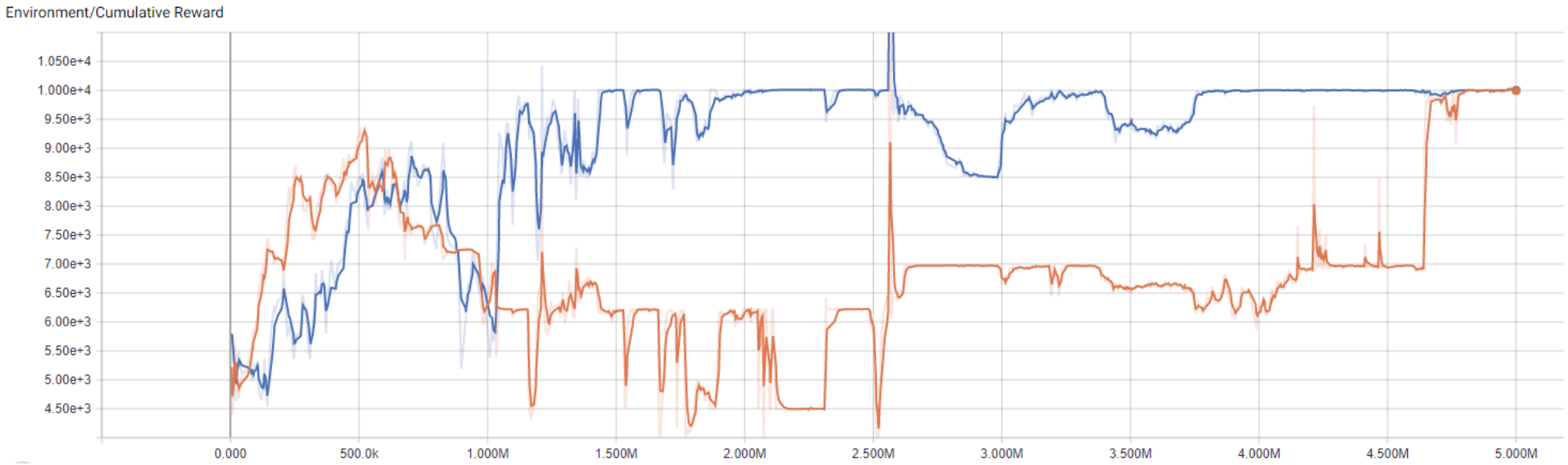


Graph 7: Rewards for each Agent in Chasers Scene

We can also see how the duration of the episode stabilizes to 200 seconds this is probably because some of the enviroments actually performed as we wanted them to, never finishing because the chaser never caught the runner. This should not have happened and could be avoided if we had a reset based on time.

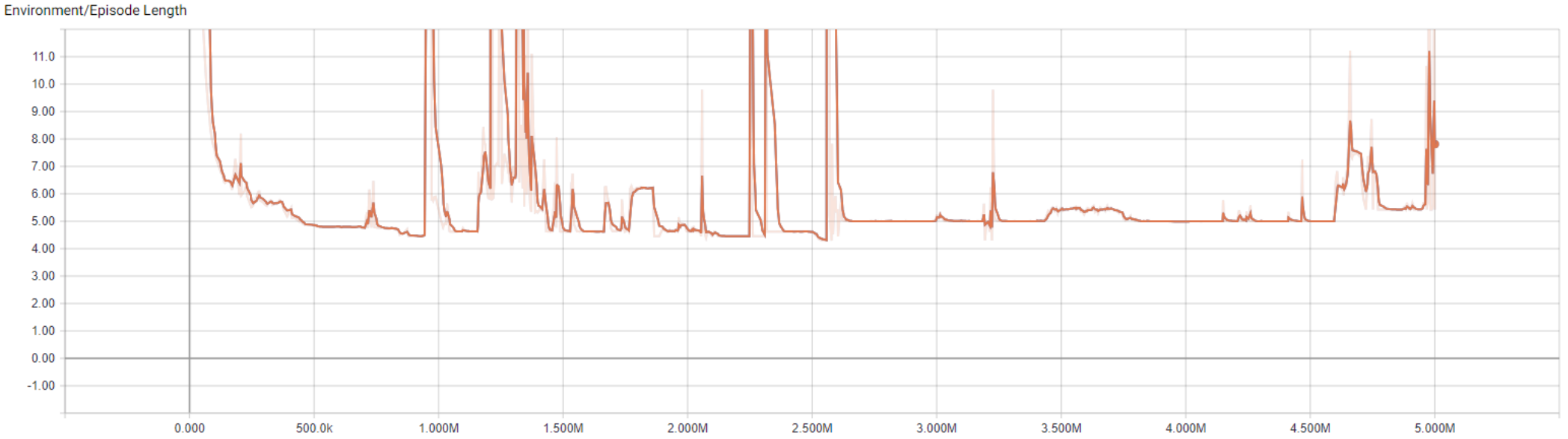
Graph 8: Episode duration in Chasers enviroment.

Sabers



Graph 9: Agent rewards in Sabers Scene

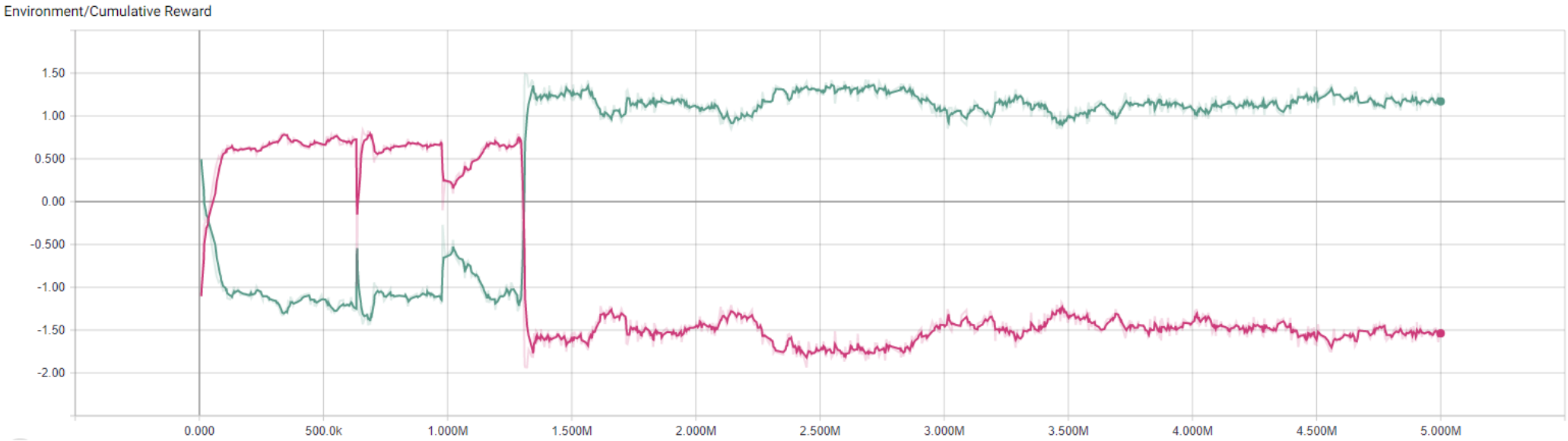
We can see how the Left (red) brain has had a worse experience even though both agents share the exact same learning script, this shows how randomness affects the results of the training greatly.



Graph 10:Episode duration in Sabers scene.

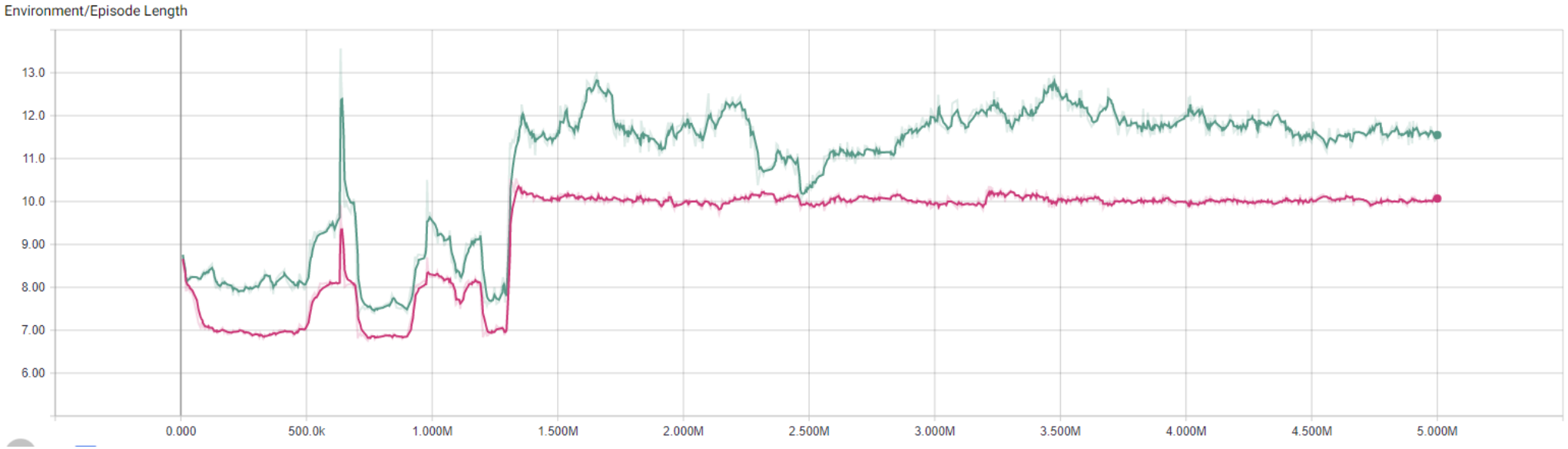
Here we see some shifts in time taken to end a scene probably because of the agents getting stuck in the corners and being unable to end an episode.

RPG



Graph 11: Agent reward in RPG Scene.

The RPG Scene starts as normal we see how the boss learns properly for the first 500k where it has a little hiccup in its reward but then goes back to normal, again a little hiccup at 1mill. But in 1.25 mill. we see a shift in reward signaling us that a problem causes the boss to go crazy and it then just stabilizes in that very bad -1.5 reward. This is probably due to what we explained before, how the boss tries to maximize the reward its given causing it to finally lose the game.



Graph 12: Episode duration for RPG Scene.

We can see in the environment duration that something causes the Agents to unlink from one another and we can also see the shifting point where the duration of the scene goes from 7 seconds to 10, accompanied by that problem we talked about before

12. SECOND TRAINING RESULTS

12.1. GENERAL MODIFICATIONS

We are now going to train the Agents by giving them **rewards only for completing the tasks**, hopefully even if we give them no "hints" on what to do they can find out by themselves.

We are also reducing the number of steps from 5 million to 3 million, since we can see that amount is more than enough for agents to stabilize.

We've also used multiple training environments simultaneously to make the training sturdier because of this we also needed to increase the buffer size.

*"**Buffer Size** - If you are having trouble getting an agent to train, even with multiple concurrent Unity instances, you could increase buffer_size in the trainer config file. A common practice is to multiply buffer_size by num-envs."* [22]

We changed how the brains were selected for the game from a fixed one to a list of candidates where we randomly select one, to acoomodate this we also added the type of brain that was randomly selected for the game to the Json data structures in the Json server.


```
"game": "rpg",
"yourPoints": 0,
"enemyPoints": 3,
"result": "lose",
"playedOn": "02/06/2020 21:15:08",
"selectedBrain": "BossDirectedBrain",
"id": 16
```


Figure 75: Json data structure v2

12.2. AGENT BEHAVIOUR

Comparing it with the first training we can say that this training went a little better in general but we cannot say that it was successful.

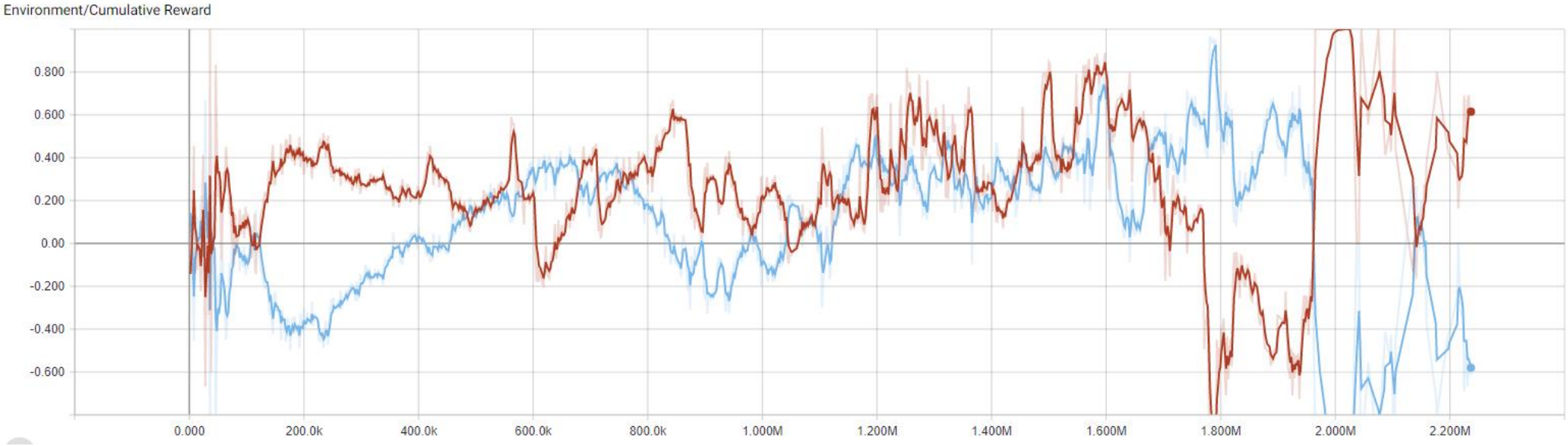
For the **Chasers**, even though they seem to have learner a little better to escape, they still get stuck when things occur in a different way to what they expected. In general this is also a mediocre result.

For the **Sabers** we again tested both networks coming out of training and they performed similarly to the previous ones, using kind of a wait and react approach more than an aggressive one.

Finally, the **RPG** boss agent was a huge success, it learned to basically spam its most damaging attack until it runs out of mana, then proceeds to attack the player to death, its basically unbeatable unless you have ludicrous amounts of luck (which was the intended result for this enemy).

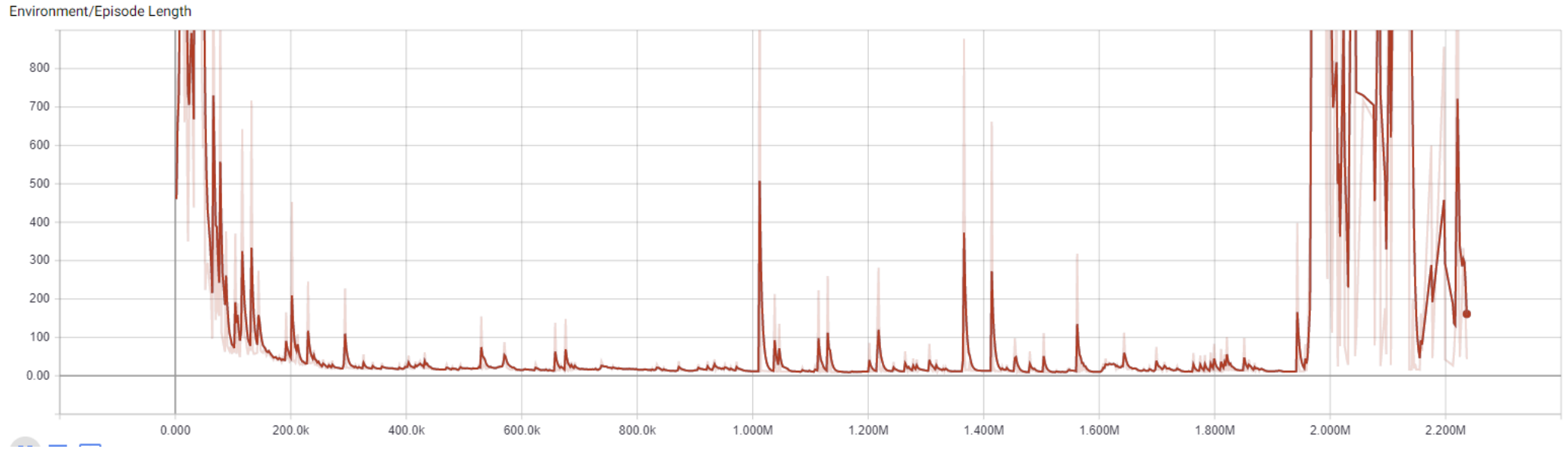
12.4. TENSORBOARD EVOLUTION:

Saber:



Graph 13:Agent Reward for Sabers second training

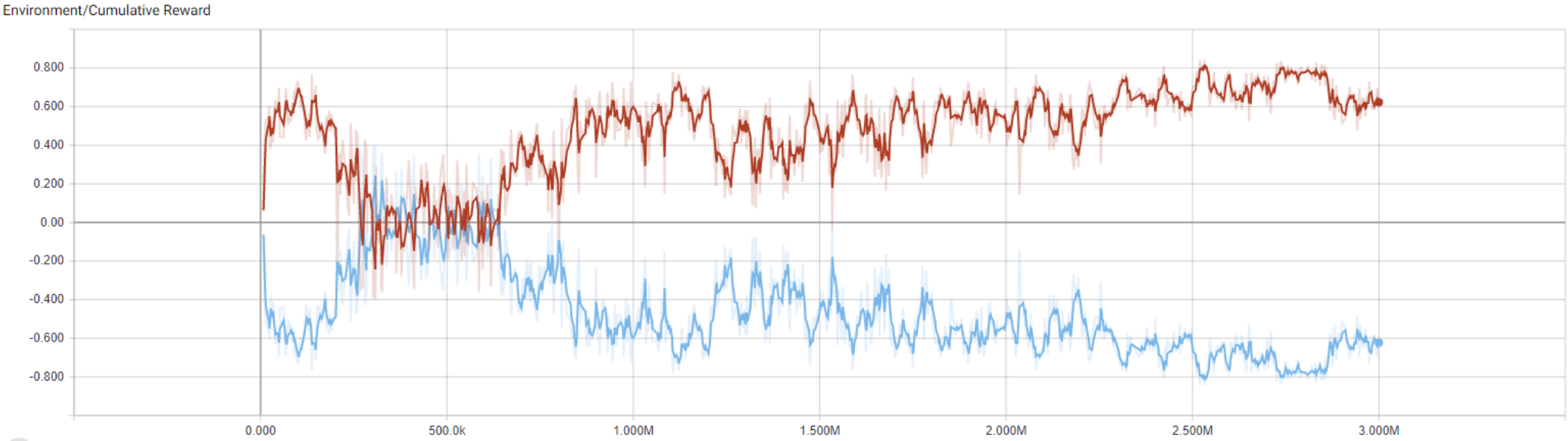
In the Sabers case we can see there are some high and lows for both agents, since they have the same brain this improvement is due to mostly randomness.



Graph 14:Agent Episode length for Sabers second training

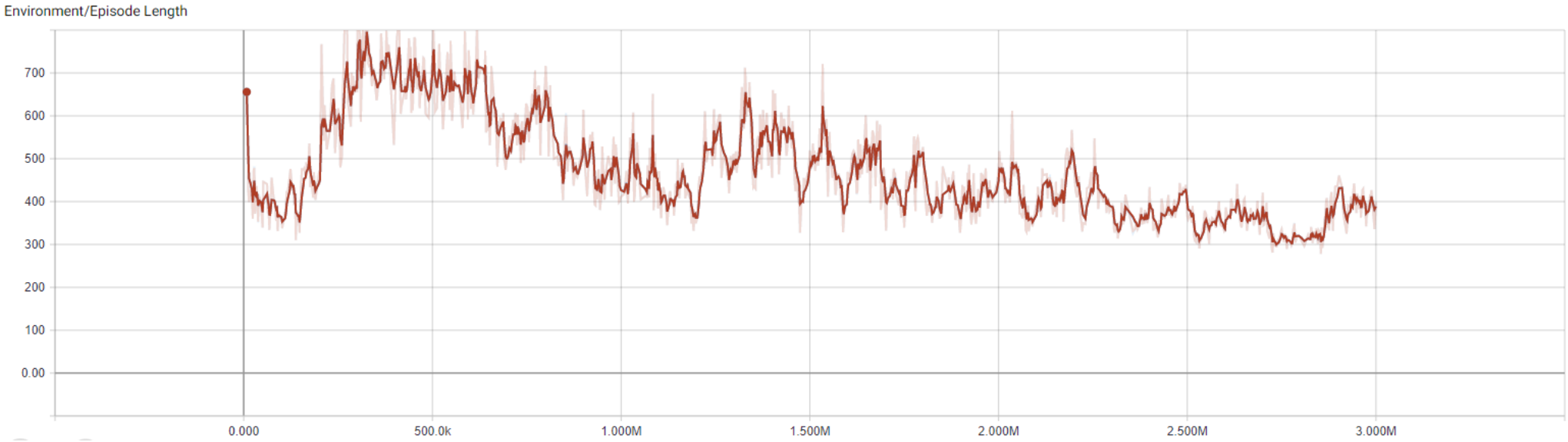
We can see that for the sabers everything went smoothly until around 2 million steps where agents started getting stuck increasing the time it took for each episode to complete. This particular training was supervised and restarted several times to avoid these locks but to no avail. In the end I decided to end the training early as the only thing that was going to happen for the agents is that they would unlearn what they learned.

Chaser:



Graph 15:Agent Reward for Chasers second training

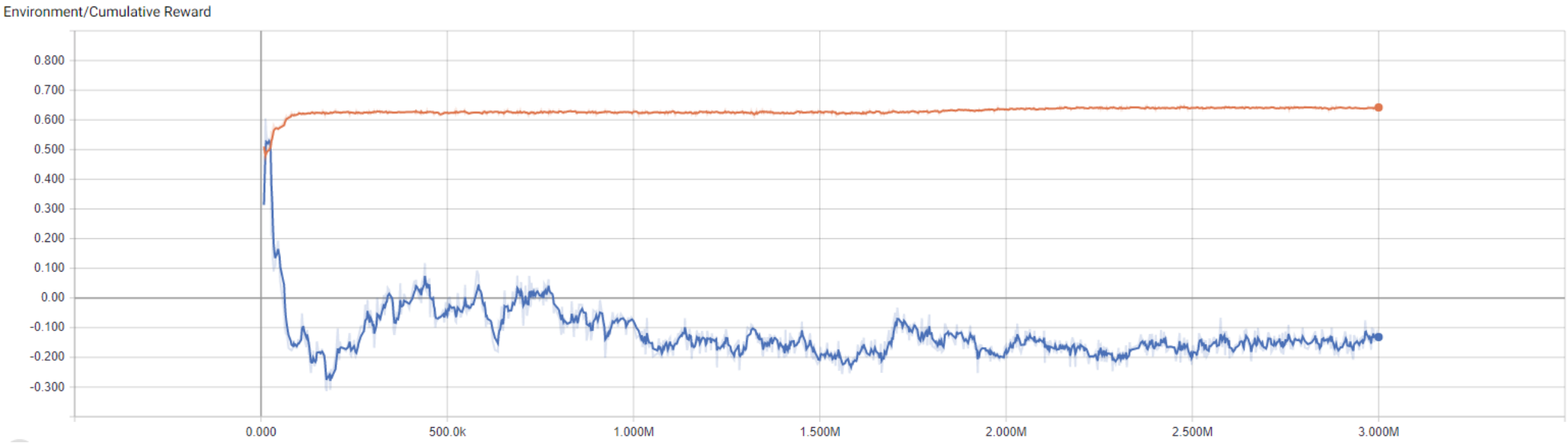
We can see how in the Chasers case a similar thing occurred with the previous training; they were pretty close until the chaser (red) eventually learned how to block every option for the runner(blue) whom ended up not responding very well.



Graph 16: Episode duration for Chasers second training

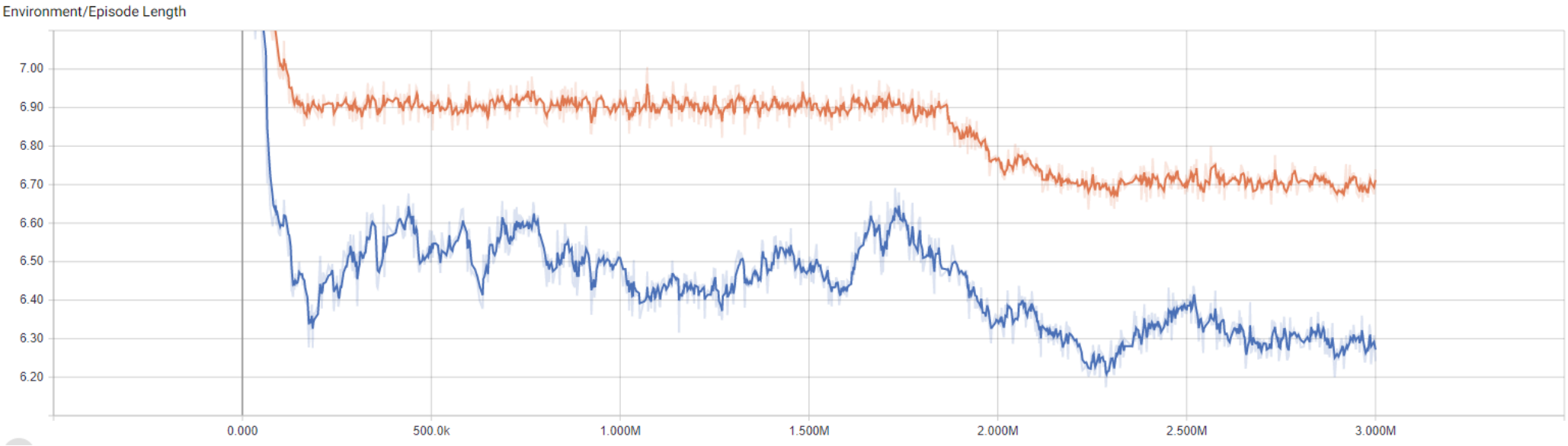
In this case we can see how the episode durations goes on for long periods of time, but end up decreasing as the chasers learn better and better.

RPG:



Graph 17:Agent Reward for RPG second training

This result is essentially what we want to see in an agent, it learns fast and remains at the top of the game for the whole duration of the training. This shows how the Boss agent learned how to perform the most damaging action and then deemed that was the best course of action and essentially did it over and over again.



Graph 18: Episode duration for RPG second training

This show the drastic reduction in time from the start of the training to the point where the boss agent has enough knowledge to just win every round in a couple of seconds, the it proceeds to stabilize there.

13. RESULTS:

With 166 total games completed (/games 166x) we can say with certainty we have enough data to analyse the results of our game training. We made some analysis and obtained the following results:

General information gathered:

game	total games	total rounds	rounds won	rounds lost	games won	games lost	round win %	round lose %	game win %	game lose %
RPG	44	137	128	9	42	2	93.43	6.57	95.45	4.55
Chasers	59	200	72	128	20	39	36.00	64.00	33.90	66.10
Sabers	63	383	85	298	6	57	22.19	77.81	9.52	90.48

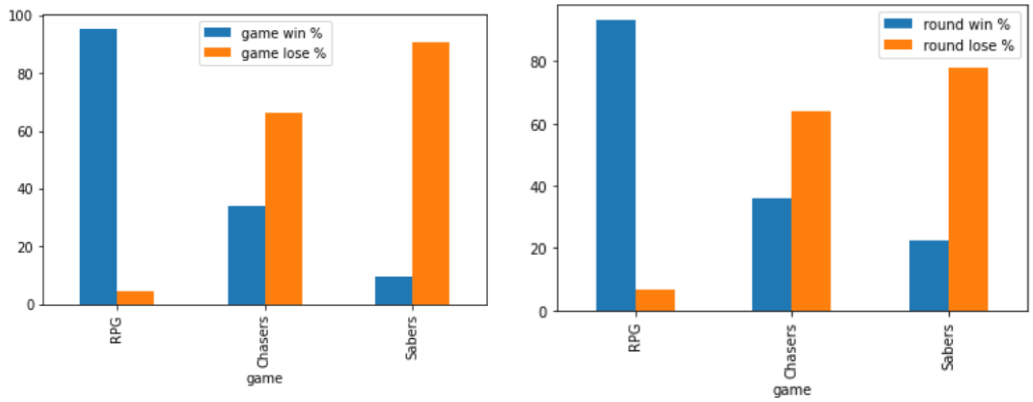
Table 3: Games Data obtained from playtesting.

Here we can see how many games and how many rounds total were played for the different games we had. First, we can see how the RPG game had the least amount of games as it was the hardest of the 3 as well as the one that took the longest time.

It was necessary to ask people to play it more since it had the least amount of plays and even then, it still has less than the other two but its not so bad anymore.

Looking at the percentages we can see how the saber agents are the ones with the least win %, this is not due to the agents being the worst of the lot, but due to the Chasers game being more confusing for the players. We did a suboptimal job of letting the players know which game they were playing once they selected it.

It is pretty obvious that the RPG was the big success of this project, with an impressive >90% win rate. It is not only due to the agents training though, it is also due to the fact that the game was "rigged" as the boss knew the players action beforehand.



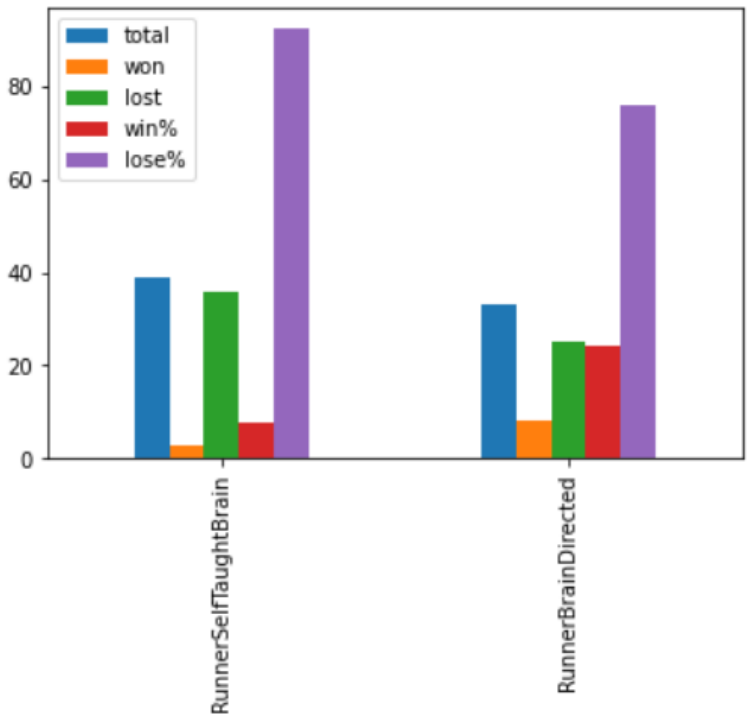
Graph 19: Win percentages for the different games.

	total	won	lost	isDirected	win%	lose%
BossDirectedBrain	59	50	9	True	84.75	15.25
BossBrainSelfTaught	78	78	0	False	100.00	0.00
ChaserBrainDirected	60	28	32	True	46.67	53.33
ChaserSelfTaughtBrain	68	33	35	False	48.53	51.47
RunnerSelfTaughtBrain	39	3	36	False	7.69	92.31
RunnerBrainDirected	33	8	25	True	24.24	75.76
SaberSelfTaught_B	110	20	90	False	18.18	81.82
SaberDirected_B	111	35	76	True	31.53	68.47
SaberDirected_A	76	11	65	True	14.47	85.53
SaberSelfTaught_A	86	19	67	False	22.09	77.91

Table 4: Brain type and game vs wins and losses

Here we can see the information broken into brain types and games, we will accompany it with several graphs to make the points clearer.

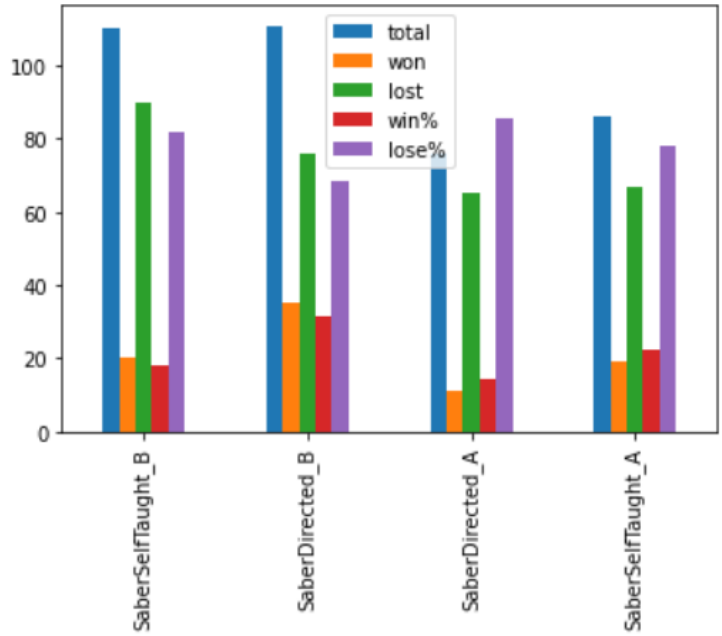
Runner:



Graph 20:Runner Brains comparison

The runners case is pretty surprising as it is the only one that performed significantly better when directed. With a win rate of 24.24% versus its directed counterpart that has just a 7.69%

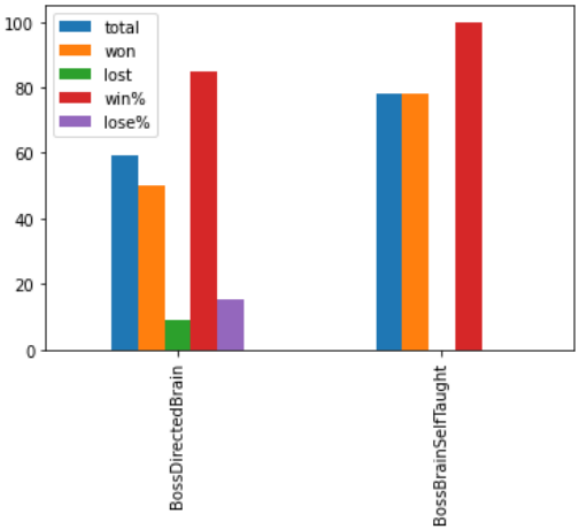
Chaser:



Graph 21: Chaser Brains comparison

But to be fair the sabers environment has also taught us that randomness also plays a factor in training and that in this case it turned out for self-taught brains to be better in one case brains and false for the other, so we can't say for sure that self-taught training is better than directed training.

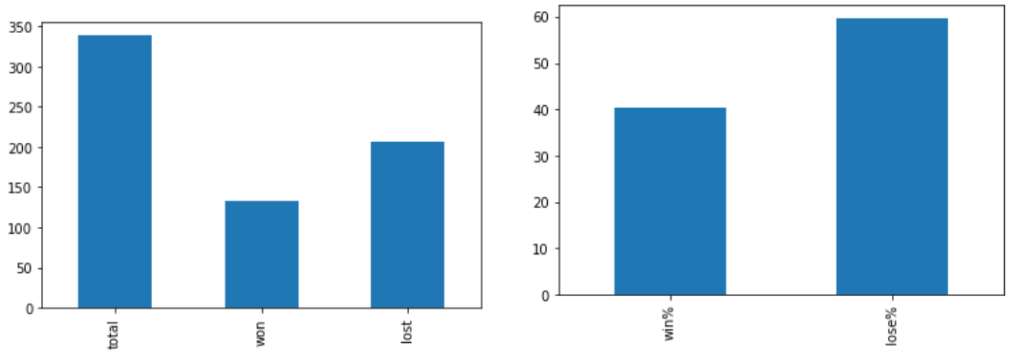
RPG:



Graph 22: Boss Brains comparison

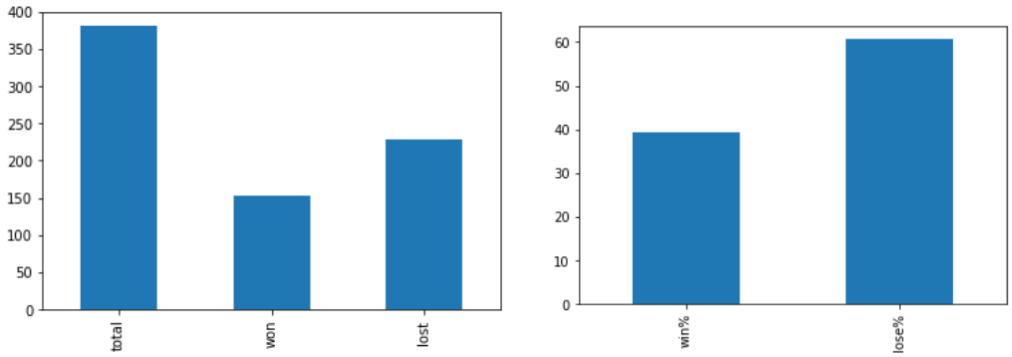
In the RPGs case it is pretty evident that self taught is better than directed, and this is mainly due to the reason we explained earlier where the directed instance chose not to attack to gain more reward, this is why networks are “smarter” that its developers since they abuse everything in their power to maximize those numbers.

Now we will see the numbers for directed and self-taught brains and compare them:



Graph 23:Directed Brains information

For Directed brains we can see how we have around 40% win ratio which is not bad considering our main source of players was people who study software engineering or people who play games often this is right about where we want them to be.



Graph 24: Self Taught Brains information.

For Self-Taught games we have around 45% win ratio, a little better than its Directed counterpart but again around the mark where we want them to be.

13.1. CONCLUSIONS

We can say confidently that the project was a success albeit not a complete one, even thought we got results that would show that neural networks would be deemed a valid substitute for human-written AI for intelligent agents, we can also say that its not the most effective and lacks uniqueness, since it can be trained to do a job and do it to the best of its ability.

We can also say that to use something like this effectively we would need very powerful computer to train a multitude of agents and keep only the best ones. Also Proximal Policy Optimization (PPO) is too random of an algorithm without any supporting components like agent memory, recurrent, curriculums etc…

We also didn't get to test SAC, this was beyond the scope of the project but it would have been interesting to compare both algorithms with enough time and resources.

Furthermore, Neural Networks have proven to be great a decision making when changes occur in a more controlled environment just like the one of the RPG, the micro inputs that the networks perform in the other training lead to a pattern eventually and when deviating from this pattern, agents lose their bearings.

Also, we have shown that controlling the networks reward too much and trying to determine the behavior we believe it should take leads to bad results as shown by the runner agent.

Training versus other agents has proven to be suboptimal as well, since both of them start out performing random actions the agents are surprised to find unrecognizable patterns in human behaviour getting stuck.

There is also the fact that we chose to train the agents more than necessary, 5 millions steps might be too big of a stepgoal for this types of agents that tend to overcompensate and end up confused.

To summarize:

- We should not work with early-development instances of technology unless we are a part of the development team.

- We have discovered how networks are good for decision making, but do not preform that good in real time control situations.

- We have shown how hardware and time are very important and limiting factors int a project of this category.

- Overly guided brains do not tend to learn well since they try to exploit the conditions they find.

13.2. RELATED SUBJECTS

Here is a list of subjects we consider to have influence in the scope of this project.

- **Statistics**, allowing the use and correct manipulation of resulting data.

- **Programming Fundaments,** since it's the base for all subsequent knowledge.

- **Algorithms & Data Analysis and Design,** this subjects provided the first look at recursive programming, algorithm efficiency, rewards etc. The knowledge it provided is of utmost importance to this project.

- **Artificial Intelligence,** this subject is the basis for this whole project, the introduction to neural networks and machine learning opens up a world of possibilities where you can apply this knowledge, eventually taking over even the most complex of subjects.

- **Software Projects Planning & Management,** this subject helped us build the planning and methodology we used in this project and helped us determine how good we were preforming based on our predictions.

13.3. FUTURE WORK

To continue the likes of this project would surely mean scrap it and star anew with a newer version of the unity editor, the new ML-Agents package and several other tools I've discovered on the way.

It would most likely lead to testing several training methods and comparing results to check the efficiency of each of them. Most notably, SAC, curriculum learning, recursive networks, among others.

It would also lead to more complex goals for the Agents to complete since the ones provided in this project proved to be simple for the agents to learn.

I would also like to apply this knowledge to any of my future projects and reduce my own need for AI development.

REFERENCES:

[1] Unity Technologies. “Unity-Technologies/Ml-Agents.” *GitHub*, 17 May 2020, github.com/Unity-Technologies/ml-agents.

[2] Dra. Marin Alonso – “GUÍA PARA LA ELABORACIÓN DEL TRABAJO FINAL DE GRADO” *bib.us.es*, 05 June 2020, https://bib.us.es/derechoytrabajo/sites/bib3.us.es.derechoytrabajo/files/guia_para_elaborar_un_trabajo_final_de_grado_1.pdf

[3] Unknown- “Cómo escribir un Trabajo Fin de Grado” *personal.upv.es*, 21 May 2020, http://personales.upv.es/fjabad/pfc/comoEscribir.pdf

[4] Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M., Lange, D. (2018). Unity: A General Platform for Intelligent Agents. arXiv preprint arXiv:1809.02627. https://arxiv.org/pdf/1809.02627.pdf

[5] Unity Technologies. “Unity | Documentation.” *docs.unity3d.com*, 26 May 2020, https://docs.unity3d.com/ScriptReference/index.html

[6] Multiple Authors, “Machine Learning in Video Games.”, *wikipedia.com*, 11 Apr. 2020, en.wikipedia.org/wiki/Machine_learning_in_video_games.

[8] kelleystarkelleystar 1, et al. “Agile for the Solo Developer.” *Software Engineering Stack Exchange*, 1 Apr. 2020, softwareengineering.stackexchange.com/questions/220/agile-for-the-solo-developer.

[9] “Sueldo: Software Engineer En Seville.” *Glassdoor*, www.glassdoor.es/Sueldos/sevilla-software-engineer-sueldo-SRCH_IL.0,7_IM1014_KO8,25.htm.

[10] “Calculadora De Impuestos, Andalucía, España.” *Neuvoo*, 2020, neuvoo.es/calculadora-de-impuesto/?iam=&uet_calculate=calculate&salary=24000&from=year®ion=Andaluc%C3%ADa.

[11] “r/Unity3D - Using Mecanim as a General Purpose State Machine Is Fantastic!” *Reddit*, www.reddit.com/r/Unity3D/comments/7bvy81/using_mecanim_as_a_general_purpose_state_machine/.

[12] Unity. “ML-Agents Publication.” *Twitter*, Twitter, 12 May 2020, twitter.com/unity3d/status/1260356826358583297?s=20.

[13] Bermudo, Miguel. “Worflow Diagram.” *Creately*, 2020, app.creately.com/diagram/k9mTgnQ29CH/view.

[14] SVG, Alex. “NN-SVG.” *NN SVG*, 2020, alexlenail.me/NN-SVG/index.html.

[15] Typicode. “Typicode/Json-Server.” *GitHub*, 26 Mar. 2020, github.com/typicode/json-server.

[16] Authors, Multiple. “Artificial Intelligence in Video Games.” *Wikipedia*, Wikimedia Foundation, 7 June 2020, en.wikipedia.org/wiki/Artificial_intelligence_in_video_games.

[17] Good, Owen S. (5 August 2017). "Skyrim mod makes NPC interactions less scripted, more Sims-like". Polygon. Retrieved 16 April 2018.

[18] Lara-Cabrera, R., Nogueira-Collazo, M., Cotta, C., & Fernández-Leiva, A. J. (2015). Game artificial intelligence: challenges for the scientific community.

[19] Wang, Ken. “DeepMind Achieved StarCraft II GrandMaster Level, but at What Cost?” *Medium*, The Startup, 4 Jan. 2020, medium.com/swlh/deepmind-achieved-starcraft-ii-grandmaster-level-but-at-what-cost-32891dd990e4.

[20] Viviane, Smith. “How to Improve Your Network Performance by Using Curriculum Learning.” *Medium*, Towards Data Science, 8 Jan. 2019, towardsdatascience.com/how-to-improve-your-network-performance-by-using-curriculum-learning-3471705efab4.

[21] Schulman, John. “Proximal Policy Optimization.” *OpenAI*, OpenAI, 9 Mar. 2019, openai.com/blog/openai-baselines-ppo/.

[22] Unity-Technologies, Unity. “Unity-Technologies/Ml-Agents.” *GitHub*, 2020, github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-ML-Agents.md.