ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Applied Deep Reinforcement Learning

DeafultVR: the AI Expansion

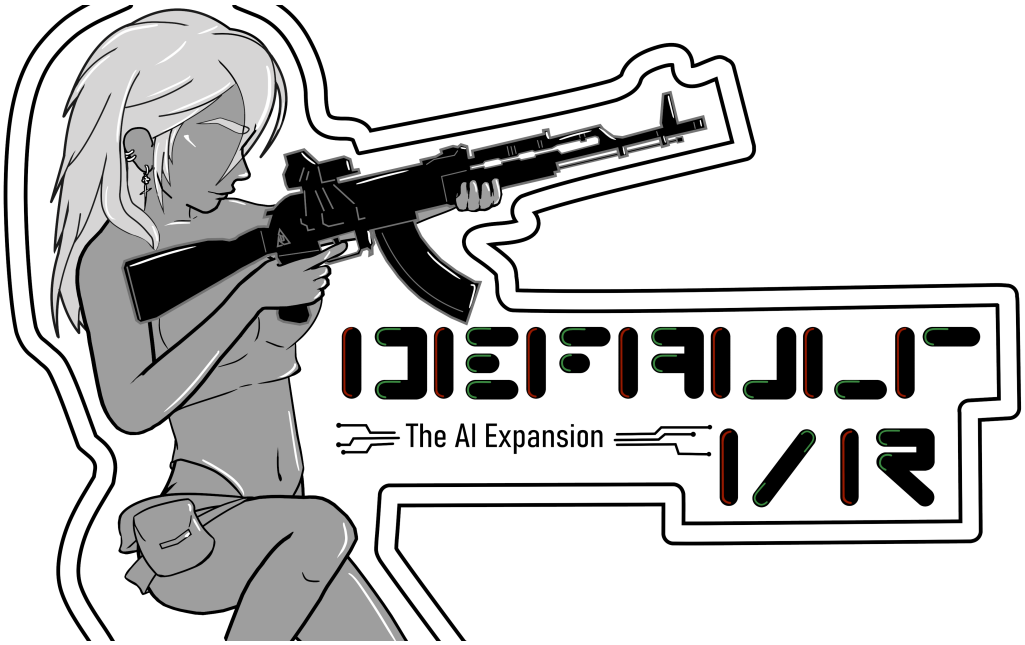
An application of artificial intelligence in competitive gaming and virtual reality

CANDIDATE:
Luca Levita

SUPERVISOR:
Prof. Andrea Asperti

Academic Year 2022-2023

Session 1st


DEFIANCE
The AI Expansion
1/12

Acknowledgements

Firstly, I would like to express my gratitude to my supervisor Professor Andrea Asperti, that steered me in the right direction. Thanks to him, I was routed toward the performance of an excellent overall work, despite our two different mindsets, one more theoretical-based and one more practical and experimental prone.

I am deeply indebted to my parents, that enabled me to study and reach this important finish line, always showing me their support. Finally now I will be able to remove my financial burden from their shoulders and delete the stress of my exams that they always felt, more than me for some poorly understandable reasons. Thank you.

Words cannot express my gratitude to Sofia, my fellow traveler, the one who made possible the awesome graphical presentation of this project, the amazing 3D models and the immersive map. She is and was always near me, offering her hand whenever I need(ed), helping me to pass through every problem and showing me how to be a better person. I love you, thank you for being with me and for your patience.

A special thanks to my grandparents, that continued to show me their support even with this long distance relationship, without any complaints about the few times we meet each other. I'm sorry that my party boy grandpa is not here; he would for sure appreciate the goal reached and the related celebrations. I know you are happy for me, even if you are not here.

My last words are dedicated to all the friends, relatives and all the people who stayed and supported me in this long journey. This goal was possible thanks to them too.

Honorable mention: my PC. It has seen and knows the worst and the best of me from the start to the end of this journey. It made possible the physical realization of all my projects (in the real sense of the word). As my mother says, it is, for sure, my third lung.

Abstract

This works presents the development of *DefaultVR: The AI Expansion*, an expansion of the first degree thesis *DefaultVR*, a virtual reality tactical shooter online game. In particular, this expansion aims to include an artificial agent, called Eve, capable of learning to play in the virtual reality world through reinforcement learning techniques. The agent learns to navigate the game environment, make decisions based on pseudo-visual information, and optimize its actions to maximize rewards. The development utilizes a deep reinforcement learning framework with the Proximal Policy Optimization algorithm included with Units's ML-Agents.

Extensive experiments were conducted to evaluate the agent's performance, comparing it against itself and human players. The results demonstrate the agent's ability to adapt and improve over time, achieving competitive gameplay skills comparable to both new and experienced human VR players. The training process involved iterative optimization and analysis of various hyperparameters, observations' and actions' spaces, and training configurations.

The successful development of the artificial agent has significant implications for the field of gaming AI, showcasing its potential for creating engaging and challenging gameplay experiences. The research contributes to the broader understanding of reinforcement learning techniques and their application in training intelligent agents for real-world tasks.

Contents

Introduction

Artificial intelligence is quickly coming in contact with everyone and with everything, making life easier and safer, since dangerous jobs and difficult tasks are gradually being accounted for by them.

The gaming field is not excluded from this rising new world. In fact, AIs could be used to improve or substitute a lot of things in this scenario, such as non-playable characters (NPC), bot enemies, bot competitive players and teams and so on and so forth. In the past gaming epoch, smart competitive bots were just a fancy idea that no one in the field would expect to become real or, at least, not this fast. For single-player games, instead, all the NPC and enemies were (and are) controlled by a deterministic and ad hoc algorithm developed for that particular character or game. This was the safer, easier and the cheapest way that companies had to give life to their heroes or villains. But nowadays AI's and the relative developing methods became more stable, affordable and the knowledge about them accessible to everyone: creation of intelligent non-human players is not a research topic only anymore.

This work is aimed to be located in the stated context. Precisely, *DefaultVR: The AI Expansion* is an expansion for my original project, *DefaultVR* [1], developed as first cycle degree thesis. It was born as an online tactical shooter competitive game in a virtual reality environment, aimed to be a realistic team military simulator. An update before this work consisted in adding a realistic physics approach, in a way the VR world has rarely seen in the current games offer. The VR software's catalogue shows solutions based on 1:1 mapping of the controllers and/or hmd (head-mounted display, or headset) with their relative virtual objects in the virtual world, ignoring collision with solid objects and reducing eventually the possible actions the player could do (for instance, if a grabbed weapon is located into one solid object, such as a wall, the user may be not able to shoot, but the

Figure 1: Physics simulation

unrealistic "passing through" action keeps the simulation far from the reality). DefaultVR was thought and designed to remove these fake approaches to the simulation cutting away the 1:1 mapping that the standard VR production offers (Figure 1), making the behavior of the player's virtual body the most similar possible to what a real body in the real world would act. This implies a better immersive experience in exchange of a little growing difficulty cost that the user must pay for adapt his mind to think that the virtual world is less virtual than the previous XR experiences he dived into. Another important feature DefaultVR offers is the full body simulation, inserting a full body avatar in the virtual world that is supposed to move realistically without the need of extra trackers but the three main commercial VR systems objects (hmd and controllers), through a technique called Inverse Kinematics, a mathematical inference method of the bone's angles given a certain number of known end points. Current VR games uses a simple method to overcome the problem of lack of extra trackers: showing just the player's virtual hands, without arms, without legs and without body.

In the expansion I will present in this thesis, there is the idea of overcoming the one big problem the original work had: it requires necessarily an internet connection in order to make the player deep dive into the simulation for a realistic feeling and an optimal immersion level, through the

creation of an intelligent agent. This project tries to expand the previous version through the idea of playing without an internet connection and let the player feel like he is playing against another person, someone that learned how to play and trained its skills, and not against a simple classic bot. The final goal is that there should be no differences between real multiplayer and enemy AI. Everyone that owns this game should enjoy it regardless of the capacity to connect to the network. (Virtual reality requires a lot of space, and often these areas are far away from an access point, such as a garage for example.)

The goal of this project is not closed to the development of a game with a neural network-based agent, but it aims to give the user a contact point between reality and artificial beings, through a realistic physical environment and an engaging enjoyable military-sim scenario.

The transition between classic bot approach and AI approach in commercial games is not immediate and, probably, will be neither taken as a possibility in the near future from gaming companies, since AIs are not fully trusted yet given their stochasticity and difficulty in having an explanation of why they took a particular decision. This is the reason why the DefaultVR's AI expansion is here in this transition period.

Moving away from the commercial gaming field, lot of studies have been done regarding the application of AI in this field, not only taking them as active entities (e.g., characters), but also as passive supervisors or helpers, such as way to full body tracks without extra trackers but the controllers and the headset in a virtual reality scenario. In general, gaming is a complex concept for an AI system, so the development of these smart agents is a good challenge and a good benchmark for researchers.

One of the early challenging examples of smart bots is the ViZDoom project, started in 2015 and still ongoing. The agent is supposed to learn how to play the 1993 game Doom, taking as input just the visual information (the screen buffer), exactly how a human would do.

Regarding the newer horizons of the research field, a new interesting topic is the development of an esports games' player, which is a good chal-

lenge and an arduous task for artificial intelligence agents. The research community has sought games with increasing complexity that capture different elements of intelligence required to solve scientific and real-world problems, such as real-time strategy (RTS) virtual competitions.

In 2016, OpenAI started an ambition project: develop a neural network-based bot able to play a long-term planning, real time strategy game called Dota2, an esport that can be played 1v1 or 5v5, where characters have each one a special role and every teammate must cooperate in order to win the $\sim$ 45 minutes match. This is an extremely challenging task, since the artificial being must deal with about 30 FPS for a long time, for a total of about 80k ticks each game to be analyzed each one in real time before taking a decision in a large action space. In 2018, the artificial guy successfully learned how to play the game and was able to beat some of this game's pro players more times. A particularly good result, even if it does not need to cooperate with anyone else. A perfect evolution of this model came in 2019, where OpenAI developed an agent with the last piece missing for a team game player: the ability to play as a team. OpenAI Five [2] was the first model able to play a long horizon strategy team game, compete and win against the world champions of Dota2. This result was obtained through an approach based on reinforcement learning.

The same year, another strategy game agent was introduced from DeepMind to the public: AlphaStar [3]. This AI learned to play 1v1 matches in the game StarCraft II. The tricky parts are the same as the previous game: long term planning, bunch of different strategies, real time decision, large action space and imperfect observations (unlike games like chess, the player does not have the full information about what is happening and what the opponent is doing). Also this model was able to beat pro players, such as Team Liquid's Grzegorz Komincz, known as MaNa.

This topic is far from being considered completed, and this project is an example of a revolution in XR gaming as it is known today.

A game driven by an AI would be a huge news in this deep-rooted deterministic-based world. And what if it is even settled in virtual reality?

People could stay in contact with these new artificial beings and interact with them in a comparable way they interact with other humans. *DefaultVR* aims to be the missing link between artificial intelligence, virtual reality, realistic physics simulation and competitive gaming world.

The dissertation is going to be structured as it follows:

Firstly, there will be the general idea of the project, followed by some gentle introductions of the main tools employed, such as Unity, Reinforcement Learning and Imitation Learning. Then there will be a study on Unity's machine learning framework ML-Agents. From this point on, there will be a deep focus on the development of the project with, at the end, a brief analysis of the optimizations performed and the game's statistics in some different VR ready PC architectures.

Related Works

Artificial intelligence plays a key role in giving new types of experiences and feelings to players. Virtual reality offers the same behavior but in a separate way. The idea of joining these two worlds and adding physics simulation to them brings the player to a new level of immersion. The overlap of those topics, both partial and total, was widely explored by the research community and industry.

Live tournament solutions achieve physical immersion in VR games recreating a simplified version of the virtual environment in the real-world arena (`https://www.youtube.com/watch?v=QJXpHp_iQF4&ab_channel=UploadVR`), as happens in the old Oculus Connect [4] (now called Meta Connect). This solution is not feasible for a generic VR system owner, since it requires a large space and a custom area for every game he would like to simulate. Another possible restricted solution is proposed in [5], where the authors present a special controller able to emulate in real life the virtual weapon recoil, and this is a possible future expansion idea of this project. In general XR games overcome to this level of immersion ignoring it, since it would require an extra development section. Framework such as HurricaneVR [6] for Unity or VR Physical Hand [7] for Unreal Engine allows the physical and realistic interaction between the real player and the virtual world, breaking the direct and rigid connection among the real controller position with the tracked position in the virtual environment. Same thing can be achieved by creating this non-rigid link manually. *DefaultVR* uses a custom heavily modified version of HurricaneVR [6] system that allows the player to have a physical virtual body and a physical multiplayer management, used to simulate the virtual VR system for the AI players. For the full body animation, an important problem is the lack of extra trackers in the commercial VR systems. Devices with this objective are sold separately and usually their price is not affordable for all the VR users. Neural3Points [8] proposes a model

that aims to achieve realistic full-body movements using the three included trackers (HMD: head-mounted display; hand controllers) with the commercial system. Conversely *Neural Inverse Kinematics* [9] propose a model that works on finding the best Inverse Kinematic output for each limb. Inverse Kinematics is a mathematical technique that, given as input the final endpoint of a limb and the arrival point, it outputs the angles of the constrained joint(s) of the limb itself in order to put the final end in a position the closest possible to the wanted point. This technique was created for robotic arms and later applied to 3D animation. The problem with those projects is the complexity that would be added to the final software: my project links VR, AI and physical interaction, creating an enormous workload for the hardware, especially for the GPU, the core for both the VR rendering and the AI inference. Adding another network (or even more) to be queried would drastically limit the set of PCs ready to run the game. So the solution I preferred to adopt was the classic Inverse Kinematics. For the development of the agent, I reviewed several aspects, such as trainers, exploration, imitation learning etc. The framework used is ML-Agents [10], an open-source library made by Unity Technologies for the Unity game engine in continuous evolution, granting an optimized way for developing intelligent agents in the Unity environment. It supports three trainers: Proximal Policy Optimization (PPO) [11], Soft Actor-Critic (SAC) [12] and MultiAgent Posthumous Credit Assignment (MA-POCA or just POCA) [13]. Among them i chose PPO for single agent training and POCA for collaborative multi agent training. I excluded SAC since PPO grants more stable single agent training and POCA was purposely developed for collaborative multi-agent behaviours. These trainers will be briefly analyzed in 2.2.

When the exploration phase part of the agent was faced, lot of literature was analyzed since this task is a well-known problem in reinforcement learning. The only one point was try to convert the information gained from papers in something ML-Agents allows. "Advanced Mechanisms of Perception in the Digital Hide and Seek Game Based on Deep Learning" [14], an Hide and Seek project develop with ML-Agents, shows some key points in the development of a target searcher agent, that brought some tests for

this project, such as the observations' types and the general network structure. I analyzed and adapt this last, making it larger, since my map is way larger and complex than the one used in this paper, while some of the other hyper-parameters have been just tried and slightly adapted to my specific case. Furthermore, it uses the Curiosity [15] method to push the agent "to be curious" and visit new states, rewarding it if it discover new physical areas or new behaviours. This kind of rewarding is called "intrinsic reward" and it is a part of a larger notion, namely *intrinsic motivation* (see [16], [17]). I modified the Curiosity module with the Random Network Distillation [18] (RND), an advanced exploration method built over Curiosity (more information are provided in chapter 3). Multiple are the uses of this innovative module, both for research purposes and real life utility, such as some of the robots presented as case studies in [19] and even for the development of OpenAI Five [2]. This module swap is because Curiosity suffers of a problem known as *Noisy TV problem* [20], an issue that keeps the agent focused on a noisy unpredictable area (like in front of a noisy tv) being continuously rewarded for these unpredictable glitches that makes the module observe new unexplored states that curiosity have to reward. In *DeafultVR*'s map there are a lot of advertisement panels that flash random images, while also random objects spawn in unpredictable areas: in this case, curiosity would bring the agent not to learn anything but observing these phenomena. In the paper "Exploration in Deep Reinforcement Learning: A Survey" [21], the authors analyze and compare several deep reinforcement learning exploration techniques. Among all of the studied methods, they analyzed the imitation-based one, claiming that they saw an impressive performance using it as a starting point for complex exploratory games, such as *Montezuma's Revenge* (see also [22] [23]). However, that was not my case: I tried this idea for the exploration training phase but I did not find an acceptable result, even wasting hours in recording demonstrations. Also in [21], authors state that the actual best exploration method is the novel concept that sees diversity [24] (a branch of the notion of *intrinsic motivation*, that rewards the agent if they discover a new behaviour instead of a new state) in policy-based approaches. Even if this concept is still being developed, a careful design of a diversity

criterion beats the standard reinforcement learning with a non-trivial margin [25], probably making it the near future state of the art method. The approach I chose to follow is the goal-based method, more precisely the *exploratory goal* type (well explained in [21]), where the agent learn to explore the environment while travelling toward a goal [26].

In order to overcome the problem of learning a complex and large map with limited hardware resources, I performed a "manual" *curriculum learning* [27] [28], a technique that allows the agent to learn a tricky task starting from a simplified version of it, making its job harder each time it reaches a certain score. ML-Agents supports and offer an automated way to achieve curriculum learning, but this needs a stable metric to work on, and this task has not a stable reward in my case, given by multiple factors that will be reviewed later. In [29] and [30] authors tries to mitigate the problem of rare sparse rewards in complex exploration environments through automatic and adaptive curriculum learning, especially [29], that faced my same problem, where the primary metric is not stable. However, they used an adversarial technique that requires more model to be trained and my limited resources did not allow this path. The cheapest way to achieve an adaptive curriculum learning in my case was the creation of a deterministic algorithm that raises the difficulty of the exploration when a human supervisor decides to do so.

To achieve the shooting task, a first idea was to create a neural autoregressor, following the networks proposed in these financial prediction's papers [31], [32] [33] and projects contained in [34]. In this way, the hands' random positions that the recoil would have induct, could be predicted as a time series of a six features, namely (x, y, z) of both hands. A further concept was planned for this task, changing a bit the idea of the financial regressors: a residual network. In this way the model could move its predictions among a perturbation of the original positions, i.e. the human aiming pose, keeping the resulting visual agent's posture credible. However this network structure resulted in non-acceptable predictions. Using as starting point the proposed models in [34], the concept adopted was to remove the "autoregressor" idea and make the agent aim, shoot and kill before the recoil

becomes unmanageable. In case this last scenario would appear, the agent was pushed to learn that stopping its fire barrage would result in a recoil state reset, with the relative precision restored.

Lastly, in order to prepare an "enemy-aware" agent, I used the self-play technique [35], a method created for the development of self-supervised competitive agents. It lets the agent play against a snapshot of itself, promoting the achievement of smart techniques to beating the enemy. ML-Agents supports the original algorithm [36] with the possibility of switching "enemy's brain" in run-time with an older or newer version of it, avoiding, in this way, the overfitting. I widely used soft-play with this anti-overfitting technique for the second phase of the exploration training, where opponent agents are supposed to search each other; for the weapon fire management, where agent need to kill the other one before this will do it first; and in the end for the union of these two primary tasks. The problem of self-play is that has been designed for symmetric games (i.e. where both agents have the same goal). Creating agents with different roles at a certain time would require the training of two different networks and the swap of the brains during the changing role time. There is no problem for the deathmatch mode of this project, but when this will be extended with the "search and destroy" mode as the common tactical shooters, attacker and defender brains need to be trained. OpenAI worked on a project that expands the self-play to asymmetric goals too: in "Asymmetric self-play for automatic goal discovery in robotic manipulation" [37] they showed how two ai-based robots learned to play an asymmetric game where one agent asks the other to solve a challenging task (known or unknown) to the other. The current limitation of this approach is that it requires a resettable environment, since the solver needs to start from the same initial state of the problem proposer. In my case this is not a real drawback, since the role swap would take place in a completely scene reset scenario.

1. Chapter 1

1.1 The Project

This project is an expansion to the *DefaultVR* game, a VR application developed two years ago in my first degree thesis. [1]

The original project aimed to develop a multiplayer shooter VR game with the "team deathmatch" and "search and destroy" modes. In both cases, a specialized handmade matchmaking algorithm creates teams automatically taking in account the skill level of each component in order to make a balanced game instance.

An interesting feature was added to this game after the first graduation: the physics simulation. This means that grabbable objects (such as weapons) and hands are not able to pass through virtual physics objects (such as walls or other grabbables). So the controllers real position and virtual relative are not in a 1 by 1 mapping, but the second just tries to reach the target position without the pretension to coincide.

In this expansion, I try to overcome the problem of the play-space without an internet connection: virtual reality requires lot of space in order to freely move the hands and often these spaces are not in range of access points (just imagine a garage for example). With the growth of artificial intelligence in games, I thought that a single-player mode with the same hype of a multiplayer match could be made real. This project is about the development of an artificial player that learns how to play as a human would do, understanding how to move hands for aim and how to explore and find targets in a complex map.

The designed mode is the team deathmatch, or better, the deathmatch, since for now it is supposed to be a 1v1 game, where a human player will challenge the artificial being, called Eve. The rules are simple: explore the

map, find the enemy and kill him before it does it. At least one rifle will be provided to each player every time a spawn occurs. Each weapon deals a different amount of damage depending on the body area the projectile hits and the type of weapon itself. The one who kills more time the other before the time expires, wins the match.

In addition, an extra mini-mode will be available where the player and artificial being will challenge each other to a mini-game in the range. Each participant will have an unlimited number of rounds in their magazine and the one who kills more enemies within a certain amount of time wins the match. Other variants could be also included in the final game.

Eve's brain has been trained using deep reinforcement learning, making her training path the most similar possible to the human learning process, but restricting a bit her range of actions in order to overcome the problem of limited resources available for the training.

1.2 A gentle introduction to Reinforcement Learning

Reinforcement learning (RL) is a branch of the machine learning's field. It deals with sequential decision-making problems. Training an agent with this technique means interacting with an environment, taking action in it and possibly modifying its state. After playing some of the possible actions, it receives feedback under the form of rewards and penalties, improving its decision-making process considering this score. The goal of the agent is to maximize its cumulative reward over time selecting the best possible action in its current situation. This is done using a function that maps states to actions, called *policy*. This function can be established in various ways, depending on the problem.

Reinforcement learning is based on the *Markov decision process*'s (MDP) concept, which is a mathematical framework that models sequential decision-making problems. An MDP consist in a set of states, actions, reward and transition functions that describes the probability of moving from one state

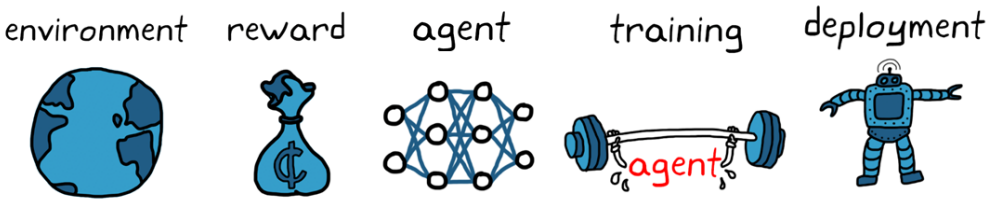


Figure 1.1: Reinforcement Learning Workflow [38]

to another when an action is taken. Formally, consider a set of states S, a set of possible actions A, a reward function $R(s,a)$ with $s \in S, a \in A$, a transition probability function $P(s',r|s,a)$ and a discount factor γ. At each time step t, the agent takes observations from the environment in the current state s_t, selects an action a_t from the set of allowed actions and it receives a reward $r_t = R(s_t,a_t)$, making it transit from the current state to a new state s_{t+1} with a probability of $P(s_{t+1},r_{t+1}|s_t,a_t)$.

Reinforcement learning can be split in two big categories:

- **Model-based**. Model-based RL algorithms aim to learn a model of the environment, which includes the transition probabilities and rewards associated with each state-action pair. Once the model is trained and ready to use, the agent will use it to plan the future actions and determine the optimal one to take in each state. An example of a model-based algorithm is the *Monte Carlo tree search*. [39]
- **Model-free**. Model-free RL algorithms, on the other hand, directly learn a policy of a value function without explicitly creating a model of the environment. They also splits in two families:
  - **Policy-based**, methods that learn a parameterized policy that maps states to actions directly (for instance *PPO* [11]);
  - **Value-based**, methods that learn an optimal value function's estimate, which models the expected cumulative reward that can be obtained by following a particular policy (such as *Q-Learning* [40]).

Model-based methods tend to be more sample-efficient wrt the model-free alternative, because they can use the learned model to infer the possible tra-

jectories and estimate the expected cumulative reward. In general, model-free algorithms are simpler and more flexible, but they usually require more samples before convergence. Model-based methods can require less samples, but they are intrinsically more complex to implement and require good model of the environment in order to learn something.

A further classification in RL can be made regarding the training process:

- **Online RL**;
- **Offline RL**.

In *online RL* the agent gather data directly, collecting data from experience interacting with the environment and using them, immediately or adding them to a small buffer, to learn the policy. On the other hand, in *offline RL*, the agent uses data and demonstrations previously collected (for instance by humans), learning from them without interact with the environment at training time. Both methods presents pros and cons. Online RL requires the agent to be trained on the real world or in a simulator, which requires a complex pre-training phase, i.e. the building of the environment, but in this way it is free to explore and try all the possible actions, taking a direct feedback from them. Offline RL does not require a training environment, but it can fall in the the so called *counterfactual queries problem*, where the agent decides to do something for which there is no data available (for instance, a robot can decide to turn right in a certain intersection and our data does not contemplate such choice). More information about online and offline RL methods can be found here: [41] [42].

1.2.1 Deep Reinforcement Learning

A winning choice for complex task is for sure a method that combines reinforcement learning with deep neural networks: *Deep Reinforcement Learning* (DRL). It has been successfully applied to a wide range of applications, such as robotics and gaming. One of the most famous projects that used this technique to train its agent is OpenAI Five, previously mentioned [2]. The main advantage of DRL consist in the ability of a neural network (NN) in learning a good representation of the state and action spaces that suits perfectly for

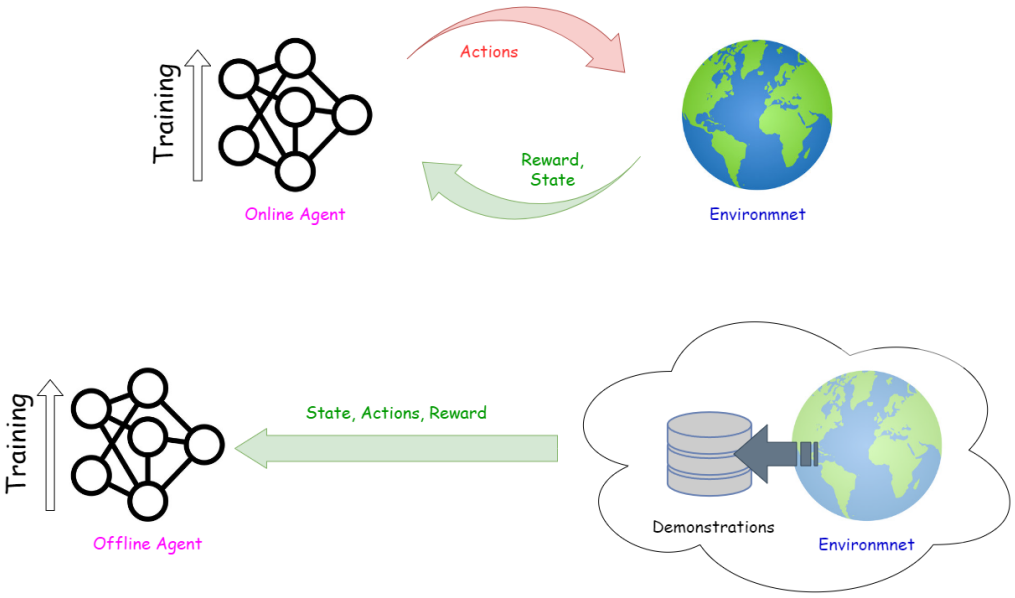


Figure 1.2: Online reinforcement learning vs Offline reinforcement learning

the task it is working on. For example, in game like Go, the state space consist in the position of the pieces on the board, while the action space in the possible moves. A DRL can learn an optimal representation of these spaces in a way that capture the relevant feature of the game and making the agent take effective decisions.

The key components of a DRL algorithm are:

- the policy network (pn): it takes the current state s as input and outputs a probability distribution over the possible actions $\bar{A} \in A$ (some actions may be not allowed in a certain state);
- the value network (vn): it requires the current state s and returns an estimate of expected future reward;
- the replay buffer: a memory of the past experience the agent observed that can be use to learn also from the previous interactions with the environment.

During training, the agent queries the policy network and performs the action this last returns, while the value network outputs its prediction. The policy is updated wrt the rewards received and what the vn returned. In

fact, the error is typically defined as the difference between the predicted cumulative future reward value and the discounted actual one, where the discount factor γ hyper-parameter determines how much the agent should weight and learn from the actual reward versus the future ones.

1.3 A gentle introduction to Imitation Learning

Imitation learning (IL) is also a machine learning's branch where an agent is supposed to learn its policy or behaviour imitating someone, that could be a human or another agent as well. In other words, rather than learning from scratch in a "trial and error" way, IL tries to make the agent learn from demonstrations of the task. There are several methods for achieving IL, here it follows a brief introduction to the two tested in this project.

- **Behavioral Cloning**. It is a simple approach that aims to train a *pn* to directly mimic the given demonstrations. It takes the current state s_t and returns the action the expert would perform in that state. The required data is a supervised set of state-actions pair. The hugest drawback of this method is the needing of a massive amount of demonstrations, since in order to let the *pn* learn what an expert would do, the dataset should exhaustively cover all the possible states the agent could meet. It also could not generalize well, bringing the network in an overfitting status.
- **Generative Adversarial Imitation Learning (GAIL)** [43]. It is a powerful technique that applies *Generative Adversarial Networks* (GANs) [44] to IL. In fact, the main idea behind GAIL is to use a *discriminator network* to distinguish the demonstrations to the behaviour generated by the *pn*. The reward function is given by the miss-classification of the discriminator: when this last starts classifying the generated trajectories as they were the expert ones, it means the policy is more or less ready, therefore the *pn* learned its task. The overfitting problem is not avoided by this method neither.

Imitation learning is considered to be a great starting point for complex task

learning, creating a non-optimal policy that behave well in certain situations and expanding it through a second training based on classical reinforcement learning.

1.4 A gentle introduction to Unity

Unity is a powerful game engine that allows the development of high quality applications for each possible genre, from classical arcade games to virtual reality immersive experiences. In this section it will be a brief presentation and analysis of Unity, in order to better understand the development of all the project in the next chapters.

1.4.1 Components

Application development through Unity is based on the concept of *Component*. To each object present in the scene are automatically linked a certain number of components that allow its correct visualization and correct functionality. Furthermore, optional components can be added or customized to extend or modify the entities behaviour. In fact, even a developer's implemented C# script become a component, which can be freely assigned to each object (the designed ones) in the scene. Certain components may require some services from others of them. In order to automatize this check, it is possible to declare in the script code the list of the dependencies, making the engine automatically add in the scene what it is needed, ensuring the script will have all the required working services. Obviously, the editor will just add these extra components without setting them in the proper way: this is up to the programmer, either via code or graphically using the editor itself. Components are also able to call some specific named functions in a custom script, as reply to an event, such as the *OnCollisionEnter(Collision collision)* that is automatically called when the following conditions are met:

- the object that is running this script has a *Rigidbody* component (the one that control the basic physics simulation) attached to it;
- the same object must have a collider component (the one that manages

the hitboxes);

- the object is entering in collision with another one which contains a collider object;
- both objects lies on a layer that can physically interact with the other one. (Layers are attributes of a GameObject that allow them to act in complex ways, such as ignoring collisions or avoiding a camera renders them).

So, even if the programmer does not subscribe its scripts to any service, some events trigger these function calls anyway.

The standard Unity registry offers a wide range of components for each possible complex behaviour where this is shared among all the common requests of game-plays, such as the basic physics simulations, the basic animation management, hitbox design, UI auto-organizer layout (for instance grid or column collection of UI objects) and so on. Furthermore, it is possible to download extra packages with extra behaviour typically not common among basic games, such as the machine learning framework and the OpenXR library, or for try in advance some experimental features the Unity Technology is working on. In addition is even possible to buy other developers' components, 3D models, SFX and all the possible useful entities a game could need under the name of *game assets*.

```
using UnityEngine;
public class ClassName : MonoBehaviour
{
    void Awake() {}
    void Start() {}
    void Update() {}
    void FixedUpdate() {}
}
```

All the custom components must inherit from the class *MonoBehaviour* or from a class that descend from it. This primordial class creates and makes possible the execution of a custom component. A C# script that inherit from

this key class has the above basic structure, where the *Awake* function is a function that is automatically called when the script object is initialised, regardless of whether or not the script is enabled; the *Start* function is called in the first frame the object containing it is enabled and, as the Awake, it will be called exactly once in the life time of the script; *Update* is called once per graphic frame while the *FixedUpdate* can run once, zero, or several times per frame, depending on how many physics frames per second are set in the time settings, and how fast/slow the framerate is. It manages the physics simulation update of the object. These four are just a small set of all the customizable functions that the engine calls automatically, more information can be found in 1.4.2 and in [45].

1.4.2 The MonoBehaviour Workflow

All the *MonoBehaviour* scripts' lifetime follows a precise flow of function calling and internal updating. The official chart in figure 1.4 (and 1.3) shows the structure of this lifecycle. It follows a brief explanation of the various sub-phases of this flow.

- **Initialization** and **Editor**. In this first phase there is the initialization of everything in the script and, while in development, in the editor. It concludes with the execution of the Start function.
- **Physics**. In this second part, the physics cycle (that loops not necessarily in sync with the rest of the MonoBehaviour cycle) works, updating the internal status of the simulation, the animations and calls the collisions callbacks if the requirements previously described are met. The loop's time depends by a user defined parameter *fixedTimestep* (ft) and by the quantity of physics interactions must be computed, making it occurs from zero to $1/ft$ times each second.
- **Input Events**. An old small phase dedicated to the call of the mouse input events from the old Unity's input system.
- **Game Logic**. It is the main phase of the cycle, where the the Update function and a large subset of coroutines (a thread like process) command returns are executed.

- **Scene Rendering**. In this section, all the graphics rendering and culling are performed for each camera present in the scene, calling the relative callbacks methods.
- **Gizmo rendering**. Phase where the *gizmos* (graphic overlays visible in development, for debug or simplicity purposes) are computed and rendered.
- **GUI rendering**. Rendering the UI graphics and calling the relative callbacks.
- **End of frame**. Returns the command to the coroutines that were waiting for the end of the frame.
- **Pausing**. This phase manages the forced pause from the application, such as when the "home" button in Android phones is pressed.
- **Decommissioning**. The last phase of the cycle, where the destruction or the disabling of the object occurs, or even when the whole application is closed. This part is usually used for unsubscribe from events.

1.4.3 OpenXR as virtual reality framework

OpenXR (*Open Extended Reality*) [46] is an open standard for extended reality applications which aims to provide an unified API for developers working on VR and AR software.

The main goal of this framework is to overcome the problem of fragmentation and lack of compatibility in the current XR developing environment. With the rapid growth of devices offer, developers keep facing the chal-

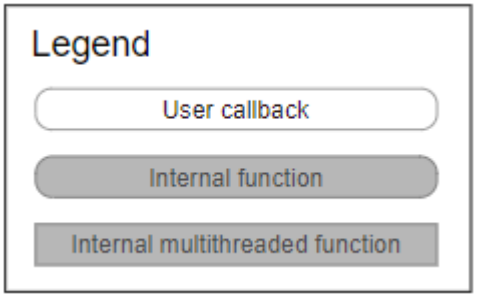


Figure 1.3: Script lifecycle flowchart legend

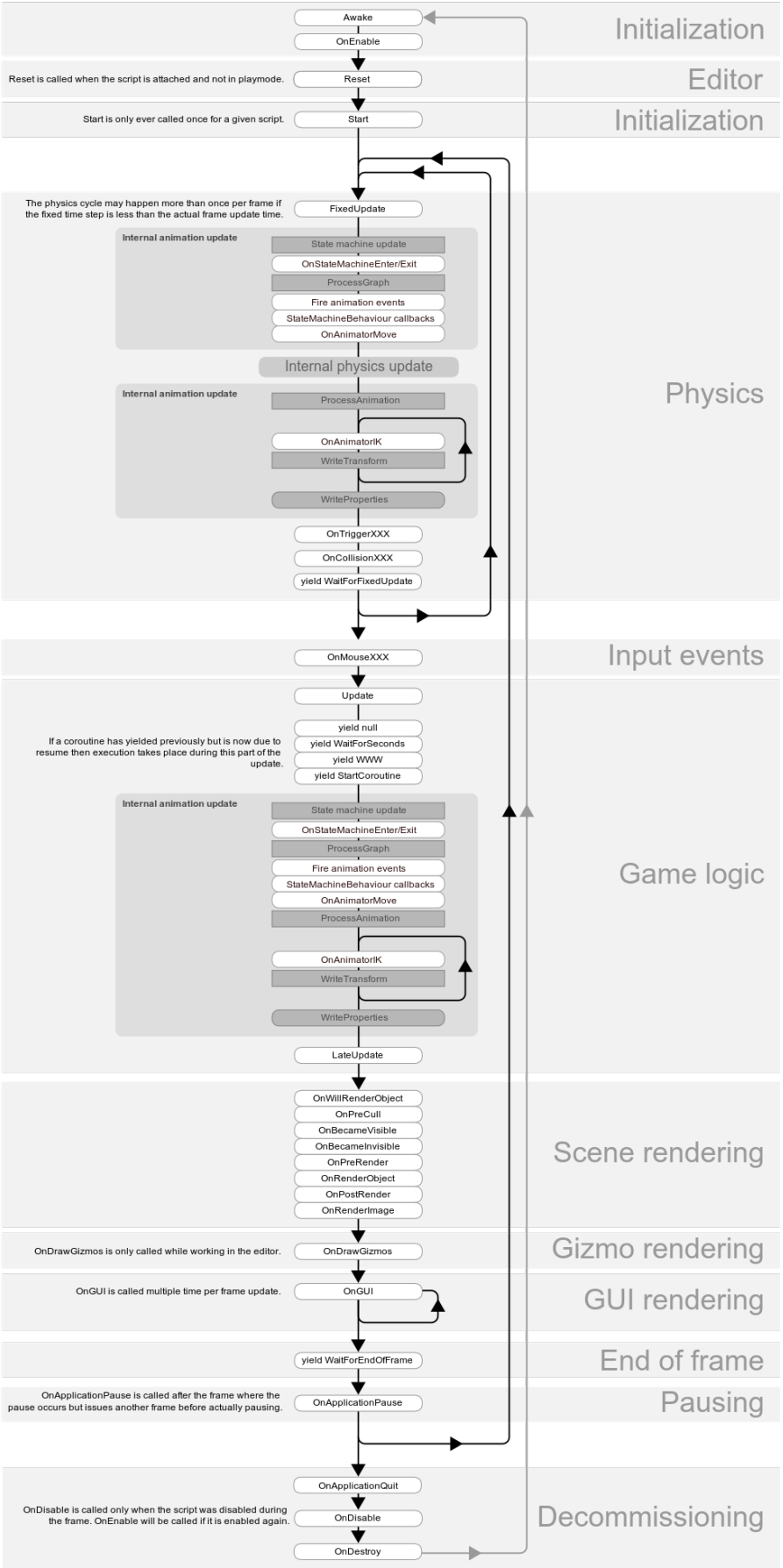


Figure 1.4: Script lifecycle flowchart

lenge of creating applications that can run on all the platforms available on the market (Figure 1.5). The solution that OpenXR propose is a new layer of abstraction that allows the automatic conversion and convergence of the hardware inputs in a unified API, and vice versa each command from the software to the device in the correct format that the target platform is waiting for (Figure 1.6).

Thanks to this standard API, OpenXR simplifies the development process and the maintenance phase, reducing the need for developers to write and maintain multiple code sources for different XR platforms.

Using this standard, several benefits are granted to the developer, such as:

- support for all the commercial XR systems;
- automatic support for all the known devices, such as external trackers;
- the possibility for unknown or startup hardware developers to build their own device and write a firmware that works with the OpenXR API, making third-party systems compatible with the big XR ecosystem;
- interoperability across all the supported devices, ensuring (under the condition of hardware compatibility) the use of different brand devices at the same time (for instance an HMD from brand A and controllers from brand B);
- full control of special devices that are not common (or even new at all) to the XR world, such as haptic gloves, finger trackers or smell generators.

Unity proposes its own implementation of this standard that can be easily downloaded and imported in the project from its libraries and frameworks manager (the Unity Package Manager). However it still supports the classic development with XR companies' SDK, including in its main settings the most popular brands.

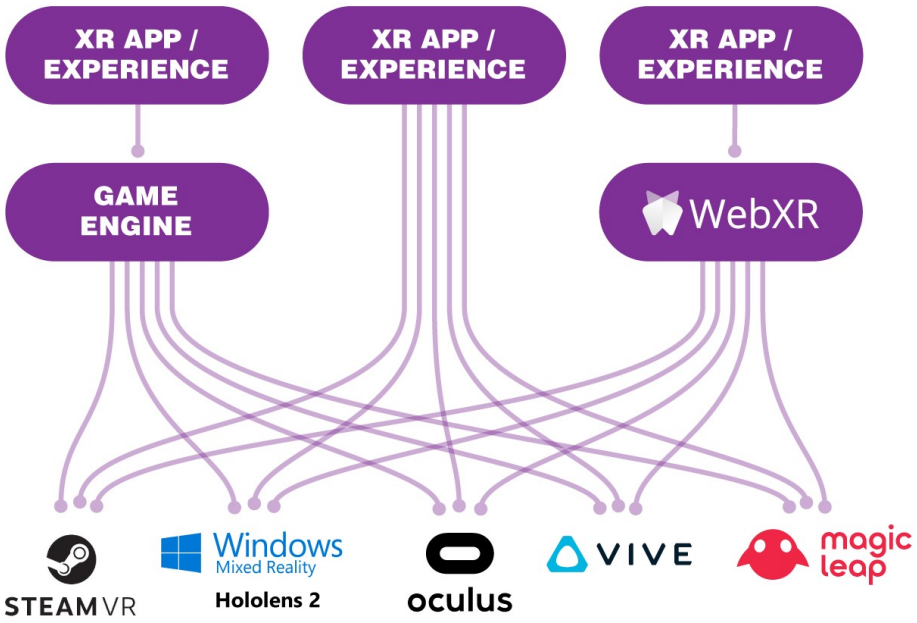


Figure 1.5: Development without OpenXR

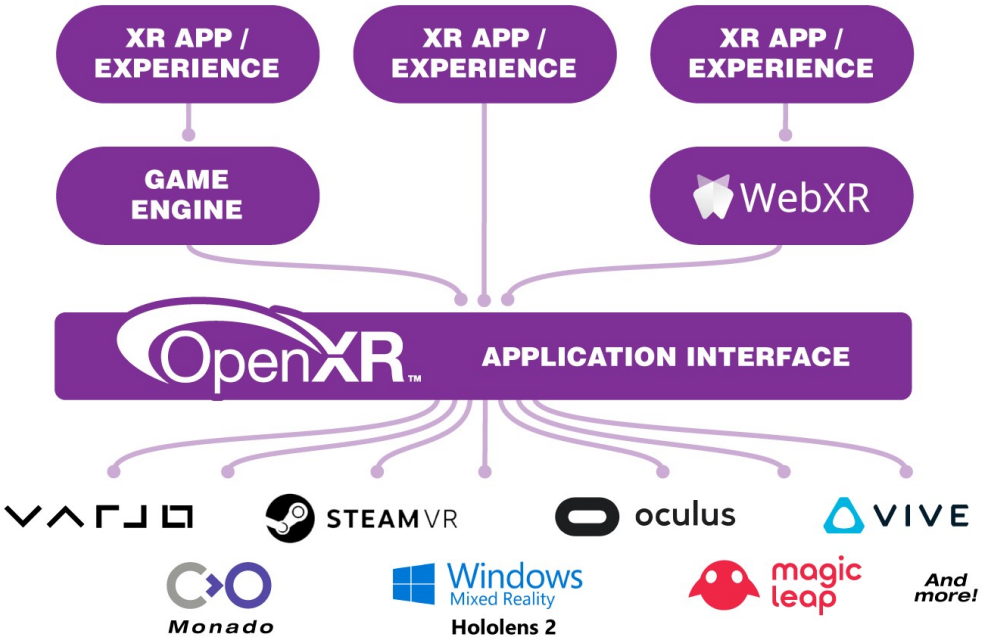


Figure 1.6: Development with OpenXR

2. Chapter 2

2.1 ML-Agents, the Unity's machine learning framework

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source initiative that allows the use of games and simulations as training environments for developing intelligent agents. This project offers PyTorch-based implementations of state-of-the-art algorithms, enabling game developers and enthusiasts to train their own smart agents effortlessly across 2D and 3D games. Moreover, researchers can leverage the user-friendly Python API to train agents through reinforcement learning, imitation learning, neuroevolution, or other applicable methodologies. These trained agents have diverse applications, including NPC behavior control in various scenarios such as multi-agent and adversarial settings, automated testing of game builds, and pre-release evaluation of different game design choices. By serving as a centralized platform, the ML-Agents Toolkit fosters a mutually beneficial relationship between game developers and AI researchers, allowing for the evaluation of AI advancements on Unity's immersive environments and subsequent accessibility to the broader research and game development communities.

The main features this package offers are:

- Support for training single agents, multi agent cooperative and multi agent competitive using deep reinforcement learning algorithms;
- Support learning through imitation learning, specifically behavioural cloning and GAIL;
- It allows to easily integrate itself in an already completed game;
- Support for custom training algorithms;
- Automatic curriculum learning possibility;

- Train using multiple concurrent Unity environment instances;
- Furnish a Unity built-in inference engine (Unity Inference Engine) to provide native cross-platform support for models trained with ML-Agents official trainers;
- Offers some built-in sensors, such as the *Camera Sensor* which provides a visual input, and allows the creation of custom ones, in addition to the possibility to give as input just some basic types, such as float, booleans but even vectors, quaternions and collections;
- You can wrap the Unity learning environments as a Gym [47] (for single agent training) or a PettingZoo [48] (for multi agent training) environment.

2.1.1 Observations and Actions

The final goal of a ML-Agent user is to create an intelligent NPC, called *Agent*, training a neural network, called *Brain*. This agent needs a simulation area where he can make its error-trial learning path, namely the *Environment*. Three entities must be defined at every moment of the game:

- **Reward Signal**: a single numerical value that is utilized to assess the performance of the agent. It is important to note that the reward signal is not necessarily provided continuously but rather when the character executes actions that are deemed positive or negative. For instance, if the character dies, a significantly negative reward can be assigned, while a moderately positive reward can be given whenever the character progresses toward its target. Similarly, a modest negative reward can be assigned when the character deviates from the correct path. The reward signal plays a crucial role in communicating the task objectives to the agent, ensuring that it is structured in a way that maximizing reward leads to the desired optimal behavior.
- **Observations**: the agent's perception of the environment. They can be either numeric or visual. Numeric observations quantify the attributes of the environment as perceived by the agent. For example, in the case of our character, it would involve the attributes of the vis-

ible area. In many complex environments, an agent typically requires multiple continuous numeric observations to effectively understand its surroundings. On the other hand, visual observations consist of images generated by the agent's attached cameras, providing a visual representation of what the agent sees at a given moment. It's important to note that there is often a tendency to confuse an agent's observation with the environment or game state. The environment state encapsulates information about the entire scene, including all game characters, whereas the agent's observation only comprises the information that the agent is aware of, typically forming a subset of the overall environment state.

- **Actions**: the character has a range of actions it can take within the environment. Similar to observations, actions can be categorized as either continuous or discrete, depending on the complexity of the environment and the agent. Continuous actions are characterized by a continuous range of possibilities, allowing for a smooth and fine-grained control over the agent's behavior. On the other hand, discrete actions represent a finite set of distinct choices that the agent can make. These actions are often used in environments where the decision-making space is more limited or can be easily quantized.

Once these three entities are defined, the next step is to train the agent's behavior. This is made possible by simulating the environment through multiple trials. Over time, the character learns the optimal actions to take for each observed state by maximizing its future rewards. Through this iterative process, the agent gradually improves its decision-making abilities and develops strategies that lead to the most favorable outcomes. By learning from past experiences and striving to maximize cumulative rewards, the agent fine-tunes its behavior and becomes more adept at navigating the environment.

The training phase is based on an interaction between the following five high-level components:

- **Unity learning environment**, which contains the Unity scene and all

the game characters. Here the agent can observe, take actions and learn;

- **Python Low-Level API**, an external component that lives outside Unity, that manipulates the learning environment through the *Communicator*;
- **Communicator**, a pipe-like concept that connects the previously two components;
- **Trainer**, the training core of the process. It is a user chosen machine learning algorithm which lies on the *Python Low-Level API* and that does not communicate directly with the learning environment;
- Optionally a **Gym or PettingZoo Wrapper** (not showed in figure).

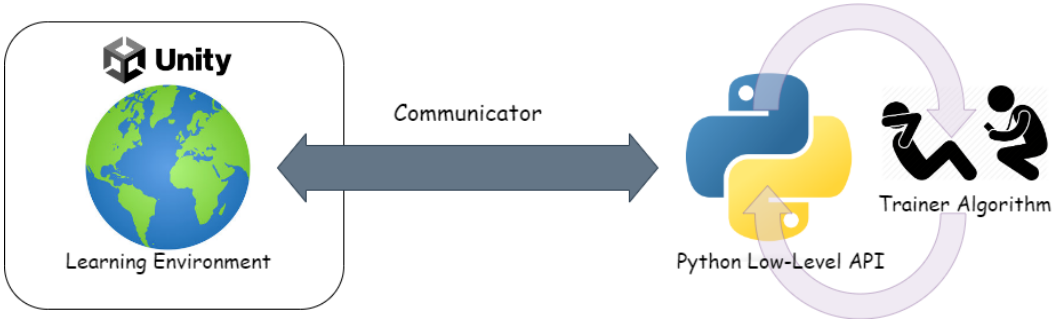


Figure 2.1: ML-Agents High-Level diagram

ML-Agents provides multiple ways for an Agent to make observations:

- Overriding the `Agent.CollectObservations()` method and passing the observations to the provided `VectorSensor`;
- Adding the `[Observable]` attribute to fields and properties on the Agent;
- Using a component that implements the `ISensor` interface.

These possibilities will be analyzed in the next section.

2.1.2 Reinforcement Learning in Unity

Designing an Agent in Unity requires these steps:

- create a learning environment;
- define the observations;

- define the possible actions;
- implement a function that maps the brain's outputs to physical actions.

After the physical character is ready, the agent's neural network can be defined in the training configuration file (2.3) with all the hyper-parameters: the training process now begins.

In order to create an intelligent agent, for first it is required the creation of the environment where it will learn. After the design of this first block, here it comes the creation of the agent itself. As for every non-static objects in the scene, our character should contain a behavioral component, a C# script that inherit from the `Agent` class (obviously it derives from MonoBehaviour), included with ML-Agent package. Adding our custom component will trigger the auto-dependencies inserting, attaching a `Behaviour Parameters` component to the object. Through this entity, some preliminary configuration of our brain will be possible:

- **Behavior Name**, the name of the brain we want to implement;
- **Vector Observation**, here it is possible to define the number of basic inputs for our network, such as `boolean, floats, Vector3, Vector2, Quaternions` and collections. The first sub-field is the number of element we want to manually feed the brain, overriding the method `Agent.CollectObservations()`. Note that a Vector3 is composed by three floats, so the voice "Space Size" must be incremented by 3, so the Quaternion by 4. The second sub-field, *Stacked Vectors* is the number of previous observations we want to give to the network in an array-like format. For instance, if I want to keep trace of the last 3 float observations, obs_0, obs_1, obs_2 respectively collected during the time steps t_0, t_1, t_2, the inputs will be in this form:

$$t_0[obs_0, 0.0, 0.0]$$
$$t_1[obs_1, obs_0, 0.0]$$
$$t_2[obs_2, obs_1, obs_0]$$

This is a simple way to give an Agent limited sort of "memory" without the complexity of adding a recurrent neural network.

- **Actions**. In this field is possible to define the output layer of the brain, declaring how many continuous actions and/or discrete actions we want. The first one always return a value between -1 and 1, it is up the developer to extend this range through a mapping if needed, while the discrete actions requires an additional parameter each, namely the number of possible actions for branch. For example, one branch could be used to move the agent toward is forward direction, so the possible actions would be 2 ($[0, 1]$, 0 for no movement, 1 for forward), while a second branch could make it rotate left and right and the relative actions would be 3 ($[0, 1, 2]$, 0 for no rotation, 1 for left and 2 for right).
- **Model**. This field is used to put the agent in the inference mode, after the training, when our model is ready. It is also possible to chose the inference device.
- **Behaviour Type**. Allows three choices:
  - if set to *Default* and a model is available, the agent will run inference; if there is no model and the Python back-end is available, it will train, else it will go for the heuristic (if defined);
  - if set to *Heuristic Only*, it will run the heuristic algorithm;
  - if set to *Inference Only*, it will run inference if a model is provided.
- **Team Id** is used for competitive agents. Two agent with different team id will act as enemies;
- **Use Child Sensors**, if set, it will use the sensors in the children objects;
- **Observable Attribute Handling** defines which observable attributes must be used as input for the network. If set to ignore, no observable attribute will be used; if Exclude Inherited is selected, it will ignore the parents observable attributes, taking just the ones declared in the class; Examine All uses all the observable possible.

Before the analysis of the Agent script, other observation methods should

be examined. As previously said, another way to make the agent observe the environment is the use of a `SensorComponent` or a custom sensor which implements the `ISensor` interface. ML-Agents offers some useful ready-to-use sensors, including:

- **RayPerceptionSensorComponent**, that uses information gain from a set of ray casts as observations;
- **CameraSensorComponent** and **RenderTextureSensorComponent**, that create visual observations from respectively a Camera rendering and a RenderTexture;
- **GridSensorComponent**, that uses a set of box queries in a grid shape as observations.

Internally, both `Agent.CollectObservations` and `[Observable]` attribute use an `ISensors` to write observations, although this is mostly abstracted from the user.

To implement a custom intelligent character, our new script should necessarily implement the method `OnActionReceived`, from the original `Agent` class. In this function it is possible to make the character acting in the scene. This method is called automatically by the Python API or by the Unity Inference Engine whenever an action is ready to be performed, passing an `ActionBuffers` object as parameter. This element contains the brain decision, which can be accessed through the two public collection properties `DiscreteActions` and `ContinuousActions`, respectively for discrete and continuous actions. It is up to the developer to give a meaning to these numerical values, writing in the same function a certain activity the agent should perform given the decision. The super class Agent contains these others overridable methods, useful in some situation:

- `OnEpisodeBegin`, a function that is called before the start of a new episode, useful to reset the scene;
- `CollectObservations`, seen before, as a possible input method;
- `Heuristic`, where it is possible to define a heuristic based behavior. This is useful for testing the code in the `OnActionReceived` since it

plays the role of the brain, making possible the creation of a "fake network output". For instance, if I said that with a 1 in the first discrete branch my agent should move forward, through the Heuristic method I can say that if "w" is pressed, the output should be 1, so I can see if my agent will go forward; if not, there are some errors in the main method;

- `WriteDiscreteActionMask`, an advanced case method that restricts the action space to a limited subset. It is used in case an action is forbidden in a certain state (for example, if the possible actions are ["Shoot", "Reload", "DoNothing"] and the weapon's magazine is empty, I can prohibit the first action, forcing my network to take a different decision).

```
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;
public class TestAgent : Agent
{
    //Optional
    [Observale] float _f0;
    //Optional methods
    public override void OnEpisodeBegin() {}
    public override void CollectObservations
                        (VectorSensor sensor) {}
    public override void WriteDiscreteActionMask
                        (IDiscreteActionMask actionMask) {}
    public override void Heuristic
                        (in ActionBuffers actionsOut) {}
    //Required
    public override void OnActionReceived
                        (ActionBuffers actions) {}
}
```

This is an example of a basic agent script.

In order to add rewards or penalties to the agent, ML-Agents offers two functions: `SetReward` and `AddReward`. The first one sets the reward score to the given float parameter, regardless if some scores occurs before it, while the second one adds the given reward to the total of the iteration. Passing a positive number means giving a reward, while a negative number will penalize the agent.

When the agent completes or fails its task, it possible to call the method `EndEpisode`, which will terminate the current episode, starting a new one and calling the `OnEpisodeBegin` function.
Sometimes, an agent could be unable to continue its adventure not for its fault. For cases like this, there is the `EpisodeInterrupted` method, which acts identical to the `EndEpisode`, but affects the training in a different way.
The Agent super class offers an attribute which tells the agent the maximum number of steps it can perform. When the time expires, the `EpisodeInterrupted` is invoked. If this property is set to 0, there will not be a maximum amount of steps.

When an agent takes a decision? It is possible to manually ask the agent to query the brain using the `Agent.RequestDecision` method, which will trigger the observation-decision-action-reward cycle. In a slow environment or in certain tasks could not be necessary to collect observations at every steps, speeding up the simulation: it is possible to take an action without query the network calling the `Agent.RequestAction`, that will use the last return. If you need the Agent to request decisions on its own at regular intervals, ML-Agents offers a component that automatically queries the brain at regular steps interval: the `DecisionRequester` component. In general this is a recommended methodology for physics-based simulations.

2.1.3 Imitation Learning in Unity

In order to enable IL, a set of demonstrations is required. So, the first thing to do is recording some of these. It is important that the agent is fully implemented and its observation and action spaces defined. Using the `Heuristic`

function, it is required to provide a code that makes the user act as it was the agent (through keyboard inputs for example). At this point, add the `Demonstration Recorder` component to the agent gameobject, specifying the demo name, the destination folder, the number of steps to be recorded (leave 0 if you don't want a limit) and your intention to record in the next simulation. Completed this step, just begin the simulation and record as many successful episodes it is possible.

Now that the demonstrations are ready, just specify the intention to use IL in the configuration file (see the configuration file section in 2.3) and locate where the demonstrations are stored;

2.2 Supported Trainers

ML-Agents comes with three state-of-the-art DRL algorithms: Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC) and MultiAgent POsthumous Credit Assignment (MA-POCA or just POCA). It follows a brief explanation of each method.

2.2.1 Proximal Policy Optimization

PPO is a policy-based model-free DRL algorithm. The idea behind PPO [11] is to improve the training stability of the policy without moving away too much from the previous status during each epoch. The key is avoid large policy changes in a short update period for two main reasons: it is empirically proved that smaller policy updates tends to converge to an optimal solution, while big steps during the update can fall in a very bad policy that could take a lot of time or no chance to recover to the previous better one. For this reasons PPO is known to be a "conservative" trainer. Before moving on with PPO, let's introduce the general *Policy Objective Function* in reinforce:

$$L^{PG}(\theta) = E_t[log(\pi_\theta(a_t|s_t)) * A_t]$$

where $log(\pi_\theta(a_t|s_t))$ is the log probability of taken the action a_t at the state s_t, while A_t is the advantage coefficient (if $A > 0$, the selected action is better than the other possible actions). The idea is performing a gradient descent step of the negative of this function (i.e. the gradient ascent), so our agent will be pushed to take more rewarding actions and avoiding harmful ones.

PPO re-design this function in a way that prevents destructively large updates, constraining the policy not to change too much: the *PPO's Clipped surrogate objective function*.

$$L^{CLIP}(\theta) = \bar{E}_t[min(r_t(\theta)\bar{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\bar{A}_t)]$$

$r_t(\theta)$ is called the Ratio function which represent the ratio between the probability of taking actions $a_t|s_t$ in the current policy and the same for the previous policy status, and it is defined as

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

If $r_t(\theta) > 1$, $a_t|s_t$ is more likely in the new policy version;
If $0 \leq r_t(\theta) \leq 1$, $a_t|s_t$ is more likely in the old policy version.
This ratio is a simple way to deduce which policy version is better.
The second section of this new function, called the *Clipped objective* ensure $r_t(\theta) \in [1-\epsilon, 1+\epsilon]$, where ϵ is an hyper-parameter that should be tuned (in [11] is proposed a value of 0.2).
In the end, the formula takes the minimum value between the clipped and unclipped section, making the final objective a lower bound of the unclipped objective.

2.2.2 Soft Actor-Critic

SAC [12] is a value-based model-free actor-critic algorithm that uses double Q-learning. It works as it follows:

it initializes a policy network (the actor) $\pi(a|s)$ and a value function network (the critic) $Q(s,a)$. The policy network outputs a probability distribution over actions given a state, and the value function network estimates the state-action value function. During data collection, SAC interacts with the environment to collect trajectories (s_t, a_t, r_t, s_{t+1}). The policy network samples actions from the distribution defined by the exploration policy. The value function network is updated by minimizing the mean squared Bellman error:

$$L_Q = E[(Q(s_t,a_t) - y_t)^2] \qquad y_t = r_t + \gamma(1-d_t)Q'(s_{t+1},a_{t+1})$$

where Q' is a target network with delayed updates, γ is the discount factor, and d_t is a binary indicator for episode termination. The policy is updated by performing gradient ascent on the expected reward and entropy objective function:

$$J(\pi) = E[r_t + \gamma(1-d_t)V(s_{t+1}) - \alpha log(\pi(a_t|s_t))]$$

where V is the value function network, α is a temperature parameter controlling the level of exploration, and $log(\pi(a_t|s_t))$ represents the entropy of the policy. The final update formula is:

$$\nabla_\theta J(\pi) = E[\nabla_\theta log(\pi(a|s))(Q(s,a) - \alpha log(\pi(a|s)))]$$

A replay buffer is used to store past experiences, and mini-batches of experiences are sampled during updates to improve sample efficiency and reduce correlations.

SAC iteratively performs data collection, value function updates, soft Q-function updates, and policy updates to improve the policy and value

estimates over time, balancing exploration and exploitation.

2.2.3 MultiAgent POsthumous Credit Assignment

MA-POCA [13] is an innovative approach to train multiple agents developed by Unity Technologies, employing a centralized critic as a neural network acting as a "coach" for the entire group. With MA-POCA, rewards can be assigned to the team as a whole, enabling agents to learn how to collectively contribute toward achieving that reward. Additionally, individual agents can receive rewards separately, and the team collaborates to support each individual in accomplishing their specific goals.

In the course of an episode, agents can be dynamically added or removed from the group, which may occur when agents spawn or are eliminated in a game scenario. Even if agents are removed from the game mid-episode (such as due to death or removal from the game), they still continue to learn whether their actions contributed to the team's ultimate victory. This allows agents to make decisions that benefit the group, even if it involves self-sacrifice or potential removal from the game. The posthumous credit assignment aspect of MA-POCA ensures that agents understand the impact of their actions on the team's success, fostering cooperative and strategic behavior among the agents. In fact, the "posthumous credit assignment" title is given by what the authors calls the *posthumous Credit Assignment problem*. In cooperative settings with shared rewards, an agent's primary objective is to maximize the expected future reward of the entire group. However, there are situations where an agent's set of actions could contribute to the group in a first moment, requiring then the termination of the agent itself for the the final task's objective, i.e. a self-sacrificial event. From the perspective of a reinforcement learning agent, being removed from the environment means it will no longer receive the reward that the group might achieve later. Moreover, the agent lacks the ability to observe the state of the environment when the group eventually receives the reward. This presents a critical credit assignment problem for the agent. The challenge lies in the fact that the agent needs to learn how to maximize re-

wards it cannot directly experience. It must understand that its actions can have long-term consequences for the group, even if it is not there to witness or benefit from the reward. Effectively addressing this credit assignment problem requires the agent to develop the capability to make decisions that take into account the collective interest of the group, balancing individual sacrifices for the overall benefit of the team. By mastering this skill, the agent can contribute to achieve future rewards, even if it means foregoing immediate personal gains.

MA-POCA objective function $J(\psi)$ and the advantage Adv for agent j are defined in this way:

$$J(\psi) = (Q_\psi(RSA(g_j(o_t^j), f_i(o_t^i, a_t^i)_{1 \leq i \leq k_t, i \neq j})) - y^{(\lambda)})^2$$
$$Adv_j = y^{(\lambda)} - Q_\psi(RSA(g_j(o_t^j), f_i(o_t^i, a_t^i)_{1 \leq i \leq k_t, i \neq j}))$$

where

- Q_ψ is the general baseline parameterized by ψ for agent j;
- RSA is a residual self-attention block;
- $g_j : O_i \to E$ is an encoding network for observations $o_i \in O_i$, where E is the embedding space. $O_i = O_j \implies g_i = g_j$;
- k_t is the number of active agents at time step t, such that $1 \leq k_t \leq N$, where N is the maximum number of agents that can be active at any time;
- i is the index of the other active agents;
- o_t^i, a_t^i are respectively the observations and the actions of the agent i at time t;
- o_t^j are the observations of the agent j at time t;
- $f_i : O_i \times A_i \to E$ is an encoding network for observation-action pairs;

- $y^{(\lambda)}$ is the value function update defined as:

$$y^{(\lambda)} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)}$$
$$G_t(n) = \sum_{l=1}^{n}\gamma^{l-1}r_{t+l} + \gamma^n V_\phi(RSA(g(i^i_{t+n})_{1\leq i\leq k_{t+n}}))$$

  where k_{t+n} is the number of active agents at time $t+n$, V is the centralized state value function and r is the shared reward function.

2.3 Network(s) and Training Configuration

ML-Agents requires a configuration file where the user can design his own networks (policy, reward, rnd and so on) and define the hyper-parameters for the training. Some voices are common among all the supported trainers, while others are trainer-specific. In this section I will perform an explanation of the most important fields of this file.

2.3.1 Common Trainer Configurations

- `trainer_type`: the name of trainer we want use (ppo, sac, poca);
- `time_horizon`: the number of experience steps the agent should perform before adding them to the buffer. A long time horizon brings to a less biased but higher variance estimate, while a short time horizon the opposite;
- `max_steps`: the max number of steps (observations + actions) that can be performed during the training;
- `summary_freq`: the number of experiences before generating the statistics for the Tensorboard's plots;
- `checkpoint_interval`: the number of experiences collected between each checkpoint;
- `keep_checkpoints`: the max number n of checkpoints saved. When the new checkpoint CP_t is created, the old CP_{t-n} is deleted;

- `init_path`: if you want to initialize this model with a previously trained one, the target checkpoint path must be defined here;
- `threaded`: by default, when the network is being updated by the Python API, the environment stops. If this value is set to true, Unity will continue to perform steps;
- `learning_rate`: the initial learning rate for the optimizer, ADAM in ML-Agent case;
- `learning_rate_schedule`: the scheduler for the learning rate. It can be either linear wrt `max_steps` or constant for the whole training;
- `buffer_size`: the number of experiences that must be collected before a network update;
- `batch_size`: the number of experience used in each iteration of the optimizer step.

2.3.1.1 Networks Configurations

- `num_layers`: the number of hidden fully connected layers the network will have;
- `hidden_units`: the number of hidden units per hidden layer;
- `normalize`: if set to true, a running average normalization of the vector observations is performed on vector data;
- `vis_encode_type`: the encoding type of visual observation, usually a known CNN structure;
- `conditioning_type`: if set to true and the agents contains some kind of goal-marked observations, an HyperNetwork is used to generate some weights of the policy. This brings to faster training but an higher memory and resources are requested, resulting often in an OOM exception.

Recurrent Neural Networks In order to use an LSTM-based RNN, the `memory` field must be added under `network_settings`, specifying the following parameters:

- `memory_size`: size of the memory the agent must keep;
- `sequence_length`: the number of LSTM blocks.

2.3.2 PPO-specific and MA-POCA-specific Configurations

PPO and MA-POCA shares the same configurations.

- `beta`: strength of the entropy regularization, which ensure the agents properly explore the action space;
- `beta_schedule`: the scheduler of the previous hyper-parameter. As for the learning rate one, it can be either constant or linear;
- `epsilon`: the ϵ of the PPO objective function;
- `epsilon_schedule`: same as the previous schedulers;
- `lambd`: regularization value for the *Generalized Advantage Estimate*;
- `num_epoch`: the number of epochs the optimizer should work when performing the network updates.

2.3.3 SAC-specific Configurations

- `buffer_init_steps`: number of experiences to collect into the buffer before updating the policy network;
- `init_entcoef`: the initial entropy coefficient;
- `save_replay_buffer`: if set to true, the replay buffer will be saved and loaded between training interruptions;
- `tau`: corresponds to the magnitude of the target Q update during the network update;
- `steps_per_update`: average ratio of actions taken to updates made of the agent's policy;
- `reward_signal_num_update`: number of steps per mini batch sampled and used for updating the reward signals.

2.3.4 Reward Signals

All the possible reward signals, both intrinsic and extrinsic, must specify at least the following two parameters:

- `strength`: coefficient for the reward given by the environment, depending by the signal;
- `gamma`: the discount factor for future rewards.

2.3.4.1 Extrinsic Rewards

Extrinsic reward requires just the two field described above. Usually the `strength` value is set to 1, in order to make the weight of the environment-based rewards greater than the intrinsic rewards.

2.3.4.2 Intrinsic Rewards

Curiosity

- `network_settings`: the network specs for the curiosity model;
- `learning_rate`: lr used to update the curiosity network.

RND

- `network_settings`: the network specs for the RND module;
- `learning_rate`: learning rate for the RND network.

GAIL

- `network_settings`: the network specs for the GAIL discriminator;
- `learning_rate`: learning rate for the discriminator;
- `demo_path`: the path where the demo file/folder is located;
- `use_actions`: if set to true, the discriminator will use the demo's actions (other than observation) to take its prediction;
- `use_vail`: if set to true, it adds a variational bottleneck in the discriminator, forcing it to generalize better, making the training more stable at price of higher time.

2.3.5 Other Options

2.3.5.1 Behavioral Cloning

- `strength`: it corresponds more or less to how strongly BC should influence the policy;
- `demo_path`: the path where the demo file/folder is located;
- `steps`: the number of steps BC should play. Usually BC is used in order to make the agent "see" the rewards, as a starting point for the

policy, letting then all the work to RL;

- `batch_size`: number of demostration experiences used for one iteration of optimizer update;
- `num_epoch`: number of passes through the experience buffer during the optimizer step;
- `samples_per_update`: maximum number of samples to use during each imitation update.

2.3.5.2 Self-Play

In order to create a competitive training environment, the *Self-Play* module can be added.

- `save_steps`: number of steps before a new snapshot;
- `team_change`: number of steps before switch control to the opposite team of the current agent;
- `swap_steps`: number of steps (from another counter, not the training one) before swapping the opponent policy;
- `play_against_latest_model_ratio`: the probability of playing against the latest policy's snapshot;
- `window`: size of the sliding window of past snapshots from which the agent's opponents are sampled.

2.3.5.3 Training Options

- `--torch-device`: [**cpu**, **cuda**, **mkldnn**, **opengl**, **opencl**, **ideep**, **hip**, **msnpu**, **xla**], the device where you want the training be performed on;
- `--time-scale`: the speed of the environment. Note this parameter speeds-up the Unity's graphic loop but not the physics one. This means that the agent will go faster, observing less close states, bringing then a slightly different behaviour during inference. For physics-based simulation it is recommended to use 1.

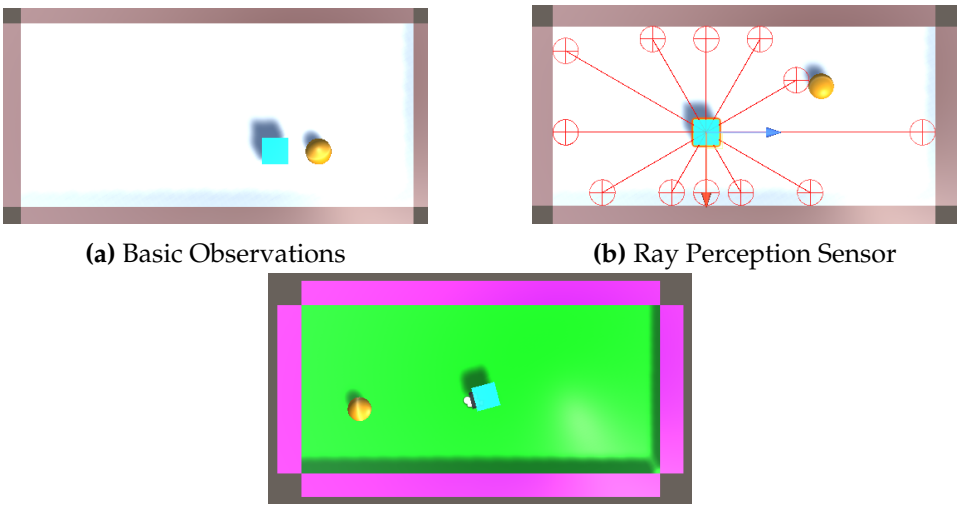
3. Chapter 3

3.1 Network Tests

Before dive into the proper project, I performed an extensive study and multiple tests on the ML-Agent framework. In this first part I will summarize them and what were the take-home concepts from each of those trials.

The first tests consisted in training simple cube agents capable of find and touch a yellow sphere target without colliding against the walls of their fields. The following tests share the same reward function: +1 for touching the target, -1 for touching the walls, -0.0001 each step, in order to force the agent not to stay still. The first two outcome cases bring with them the end of the episode.

- **With Basic Vector Observations** (Figure 3.1a):
  - action space defined as two discrete branches with three possible values each: [forward, stop, backward], [strafe-left, stop, strafe-right];
  - observation space defined as the agent position (x,y,z) and the target position;
  - result: agent learned to move against the target (without the awareness of the existence of the walls) in about 25 minutes of training and 1m steps.
- **With Ray Perception Sensor** (Figure 3.1b):
  - action space defined as two discrete branches with three possible values each: [forward, stop, backward], [strafe-left, stop, strafe-right];
  - observation space defined using 12 sphere casts all around the agent. They could detect both the target and the walls;
  - result: agent learned to move against the target in about 13 min-



(a) Basic Observations **(b)** Ray Perception Sensor

(c) Camera Sensor

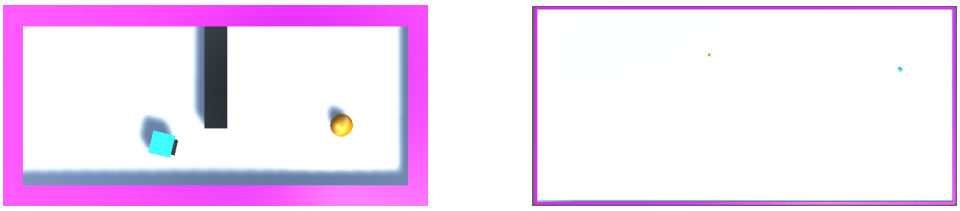
Figure 3.1: First tests

utes of training and 800k steps.

- **With Camera Sensor** (Figure 3.1c):
  - action space defined as two discrete branches with three possible values each: [forward, stop, backward], [rotate-left, stop, rotate-right];
  - observation space defined using am RGB camera placed in an hypothetical eyes-position of the cube agent. Walls and target have a different color to make them distinguishable;
  - result: agent learned to rotate and move against the target in about 5.5 minutes of training and 200k steps.

Additional tests based on variant of the above described ones were performed, such as adding an obstacle in the middle of the field, limiting the agent's perception of the environment (Figure 3.2a), and making the field as huge as possible in order not to show the target to the agent immediately (spawning them far from each other) forcing the agent to explore (Figure 3.2b).

The last study case was the *Lovers* behaviour (Figure 3.3), where the agent was supposed to explore a random generated maze in order to find another freezed agent. It took me a bit to make the agent work, testing curriculum learning (to adding complexity to the maze) and RND module (see



(a) Camera with obstacle **(b)** Camera with a huge field

Figure 3.2: Examples of test variants

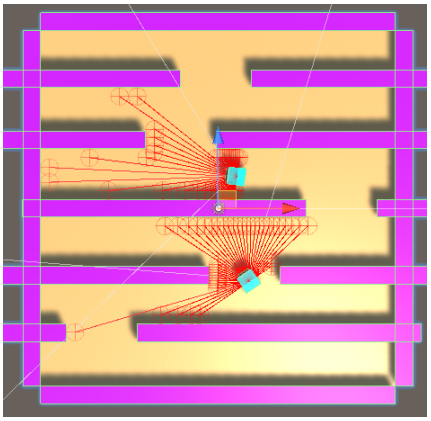


Figure 3.3: Lovers' maze

Random Network Distillation below). At the end, the agent mastered its task finding always its lover. Initializing another model from this, I tried the cooperative setup, where the other agent was supposed to execute the task too, make the average episode length shorter.

Random Network Distillation *RND* [18] is a technique used in RL to promote exploration in environments with sparse rewards. It provides intrinsic rewards to the agent that guide its learning process.

At the start, two NNs have to be initialized: the target network Y and the exploration network E. The first mentioned is fixed and usually pre-trained on prediction tasks, while the second one is randomly initialized and has no knowledge on the environment. Y is used to extract features from the agent's observed states. Let $Y_g(s_t)$ represent the extracted features from the state s_t using Y. The exploration network takes the observed state as input and tries to predict the corresponding Y's generated features. Let $g(s_t)$ denote the output of this network. At this point, the intrinsic reward $r_{intrinsic}$ computation can be performed: it is calculated as the prediction

error between $g(s_t)$ and $Y_g(s_t)$, in this way:

$$r_{intrinsic} = ||g(s_t) - Y_g(s_t)||^2$$

where $||.||$ is the Euclidean distance between vectors.
During training, E is updated, trying to predict as well as possible the Y's outputs. It is possible to consider the E network as an encoder observations-features and Y as a "dataset generator". Encoders performs in an optimal way with data that they have already seen or with something similar, while their predictions are terribly inaccurate with new data. So if a state was never seen before, the exploration network will predict something wrong, resulting in an elevated error value. This produces an high intrinsic reward, indicating the agent that it is exploring new states. More the exploration model is wrong with its result, more the agent is "surprised".

The final reward the agent will receive is

$$r_{total} = \phi_{extrinsic} * (\gamma_{extrinsic} * r_{extrinsic}) + \phi_{intrinsic} * (\gamma_{intrinsic} * r_{intrinsic})$$

where ϕ is the corresponding reward's strength and γ the corresponding discount factor.

3.2 Eve

Eve (Exploratory Virtual Executor) (Figure 3.4a) is the name of the artificial player, given after the two main tasks she is employed on and the optimizer used to train her, ADAM (ADAptive Moment estimation). The objective of this agent is to learn to explore the map, finding the enemy, aim to vital parts and kill the player before he does it. Due to the physical realism of the simulation, she has to learn that not all the actions are possible, such as make the weapon (that it is an external object from the agent perceptions) passing through a wall. Another difficulty is the physical weapon recoil,

(a) Eve 3D model **(b)** Eve's hitbox

Figure 3.4: Eve

that makes the projectile not to go precisely where she is aiming at and this issue should be understand and mastered.

The original idea was to create a wide complex network structured like this (Figure 3.5):

- Observations:
  - a visual observation from a camera placed near her eyes;
  - a one hot encoding observation of what she is aiming at;
  - her position;
  - last known enemy position;
  - the current position of both hands.
- Actions:
  - 3 discrete branches for movement [forward, stop, backward], [strife-left, stop, strife-right] and [rotate-left, stop, rotate-right], emulating in this way the controllers' joystick inputs;
  - one boolean output for fire or not [not-fire, fire];
  - 6 continuous actions for $(\Delta x, \Delta y, \Delta z)$ of both the hands bounded by the physical arms' limits.

Due to a limited set of resources available for training, this plan was not feasible, forcing me to design her brain in a different and simpler way, reducing

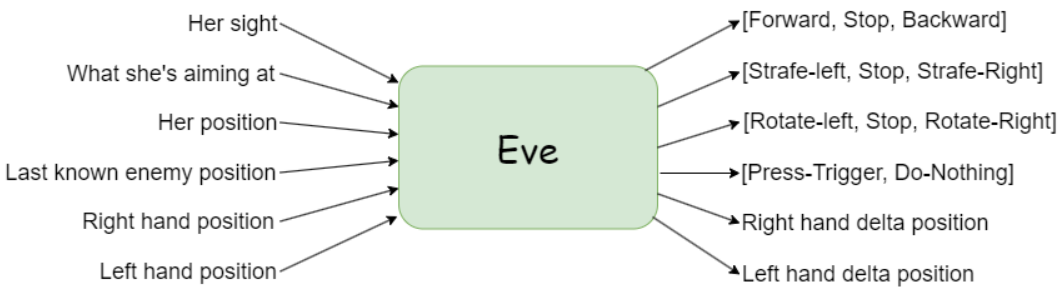


Figure 3.5: Eve's original brain idea

and changing inputs and outputs.

3.2.1 The 3D Model

As preliminary point, all that concern the graphic models of the character and the dummy enemy was created, except the final visual form that was added as the final step of all the project, in order not to make the training environment more resources-hungry than it already was. It is important to set those things, since she must learn which are the vitals points and what is the enemy shape. So, both the 3D models were rigged (the phase where the skeleton is created and each vertex of the mesh is constrained to one or more bones, to create the animations) and all the hitboxes were placed to each bone (Figure 3.4b), with a different tag relative to the type of body part (for weapon damage purposes).

3.2.2 The Brain

With the new trainer MA-POCA released recently, a new possibility opens for training this complex character: split the brain in two different parts, in a way these two are strictly related to the sub-task they need to learn: the *Shooter* and the *Explorer*. The first one is the hand manager, the one who learns how to place the virtual controllers in the space and, if it is the case, when to shoot. The second one is delegated to the exploration of the map and target finding, keeping it in sight when found.
Further changes to the original project will be described in the next sections.

Eve will automatically grab her rifle and there is no way she can release

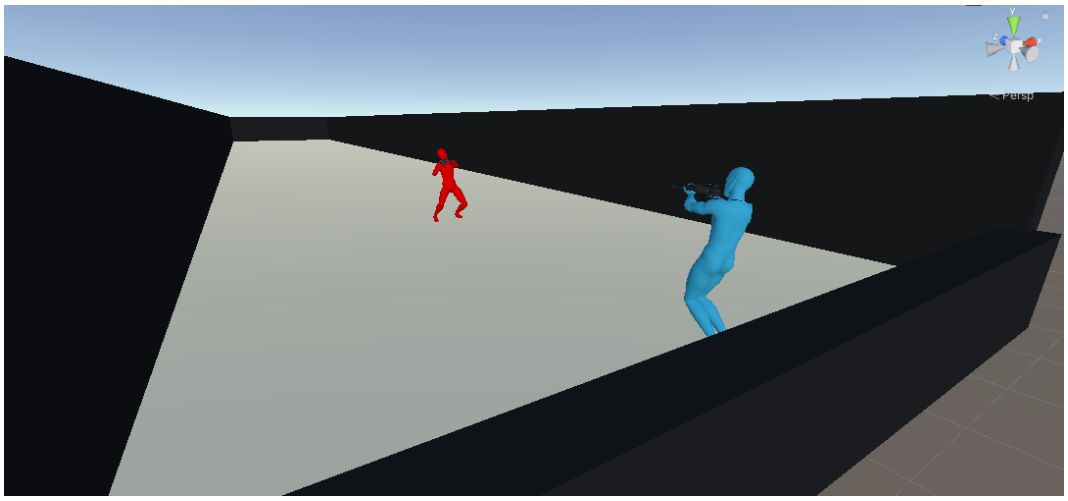


Figure 3.6: Shooter Learning Environment

the grip. This is both a necessary and a reasonable choice, since it can be seen as a deterministic behavior: no rifle no win, and letting her the decision when to grab or not would lead her to lose the weapon due to a probable outlier in the network's output (if the release is performed, the physics simulation would make the weapon fall, and she is not able to crouch. In addition, the time required to teach her how to recover a grabbable would take ages before the certainty she would not lose it somewhere in the map). This simplification was planned since the beginning, before the presented original idea.

3.2.2.1 Shooter: Observations, Network structure and Actions

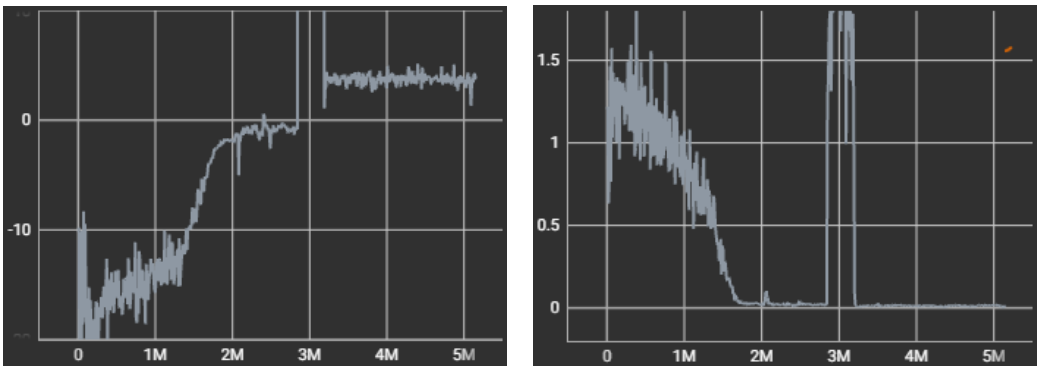
The first brain component I worked on was the *Shooter*. The learning environment was built as a shooting range (Figure 3.6), where the enemy spawns with a random position and with a random crouch height.

The initial plan was to create a residual network-like model for weapon recoil random movement prediction, following some papers for financial time-series regression with deep neural networks (more specifically these: [31], [34], [32], [33]). *Residual* because the starting position of the hand is fixed (i.e. the aiming pose) and the brain was just supposed to predict the $\Delta(x, y, z)$ of each hand, in order not to lose the correct pose.
What I designed as first try was a couple of LSTM-based regressor (one

for each hand) that take as input the current position of the hands, what the weapon was aiming at and some rays that tells the network what is surrounding the aimed point. The expected output would be the $\Delta(x, y, z)$ of the attached hand and, just in the right hand case, the trigger status. So even the *Shooter* behaviour was splitted in two and trained as cooperative agents task. The reward function $r_{\bar{shooter}}(a_t, s_t)$ was defined as it follows:

- +0.5 + 0.1 * (number of saved rounds in the magazine) for the kill;
- +0.1 for hit the enemy's head;
- +0.025 each hit to the enemy's chest;
- +0.017 each hit to the enemy's legs;
- -0.1 for each missed round;
- -0.0001 for each step the enemy is available, to force the agent to take a decision;
- -1 if the magazine is empty and the enemy alive.

After some training trials with several hyper-parameters, the best behaviour obtained was not an acceptable one, since the kill rate was lower than the 50%; So, the next step was to add a new simplification: remove the z coordinate from the prediction of both hands, taking it fixed. This is reasonable, since the backward and forward movement while aiming plays a tiny role. Without any change to the network, I restarted the training and a slightly better result came out, something acceptable with a kill rate of about 75%. The expected behaviour must be almost perfect, so I re-designed the whole network. Precisely the *Shooter* brain was reduced to a one agent task, making both hands controlled by one model (using the PPO trainer this time). I kept the idea of the residual-like NN, but I changed the regression concept: the new brain was supposed to aim and kill with the minimum number of rounds (i.e. performing head-shots), without the requirement to predict an advanced-state recoil. If the enemy dies within the first 4 or 5 projectiles, the recoil remains more or less controllable, besides the fact the agent can learn that stopping his fire barrage would bring back the weapon to a quiescence state in a small amount of time. This new network showed good



(a) Cumulative reward over steps **(b)** Policy loss over steps

Figure 3.7: Shooter Plots

performance since the first training steps, but it seemed to require a lot of time to master perfectly this skill. So other simplifications were performed:

- the right hand position is completely fixed. This is reasonable since is the left hand the one who controls mainly the weapon orientation, and what a human would do is take the right hand steady as a pivot to better control the backward recoil force;
- the observations were reduced to three elements:
  - a boolean that indicates if the enemy is available (a communication pipe between the Shooter and the Explorer)
  - what the agent is aiming at (a one hot encoding);
  - if the enemy is visible, the angle between the correct aiming direction and the current one.
- while the actions are:
  - $\Delta(x, y)$ of the left hand;
  - weapon related [Shoot, Not-Shoot].

In addition, the space where the left hand (x, y) can move on was reduced to the physical reachable one and a new possible reward was added to the $r_{\bar{shooter}}$ function: +0.001 if the left hand delta is tiny when the enemy is not available. This is just to make the agent move its hands the less possible while not needed. This new model (Figure 3.8) reached the initial desired goal: 95% of kill rate within the first 4/5 rounds and, in general, the 100%

within 10 bullets.

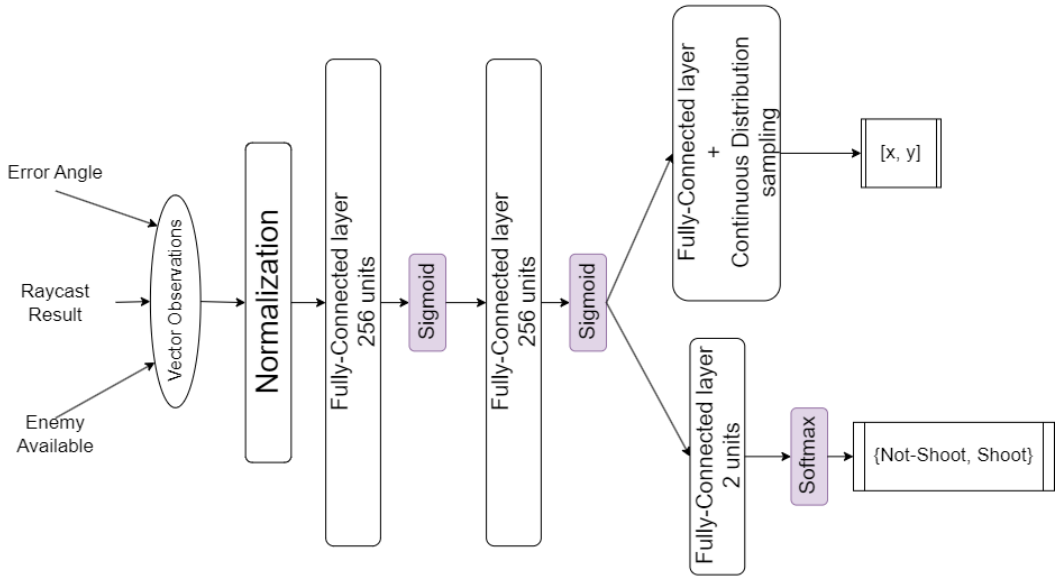


Figure 3.8: Shooter model

3.2.2.2 Explorer: Observations, Network structure and Actions

The objective of the *Explorer* is, as said previously, to explore the map searching the enemy. The learning environment consist of a simplified version of the official map, where "simplified" means lighter in terms of GPU rendering, but the difficulty is exactly the same. The design of the network was not trivial, since the observation space was not so simple to define for an optimal result while the action space received lot of simplifications. Even the inner architecture of the NN was not easy to structure and the hyperparameters tuning delayed a lot the final model training phase.

As already mention, the initial plan for the agent was to give as observation the rendering of the camera placed near her eyes. The problem is the time and the memory required to learn CNN kernels, that was not feasible with the resources available. So, instead of a visual observation, the input choice became a ray perception vector observation with "FOV" similar to the one the human eyes would provide. It is possible to consider this as an higher level information that a CNN would learn. This input method brings

with him two main problems:

- sphere casts returns what a sphere collided with, giving not precise information, since the enemy could be visible behind the corner hitted. This could be easily solved setting to 0 the sphere radius, making them classic ray casts;
- more the ray is long, more distant areas will not be visible, since all the rays start from the mid-eyes point of the agent, spreading around him. So the areas between lines will not be visible. The final solution for this issue was adding the maximum number of rays (50) and use an LSTM to remember something about the past observations.

Another possibility to overcome the second problem is to feed the NN with info about when the enemy is within the FOV of the sensor and the angle between the current agent direction and the correct enemy direction, using the function `CollectObservations`. In fact, lot of trials have been made with this idea, resulting in a sub-optimal solution that saw the agent intercepting the target but, after a bit, preferring to explore the map instead of continuing approaching him. This seems due to a wrong reward function, but it was not the case, since the same reward function was used to train the final working model.

The analyzed possibilities before the solution of the observation difficulty were the following:

- three Ray Perception Sensors, one all around the agent, useful for observing and exploring the environment without the possibility to detect the target, the second with 180° FOV to check corners without losing the actual path, and the third with a restricted FOV just to better track the enemy when found. The problem was the huge amount of inputs that would require a massive network to well-understand them;
- two variation of the previous solution, one without the environmental Ray Perception Sensors, and the other without the 180°FOV one. These solutions suffers of the same issue of the previous, always too

many inputs for a mid-sized network;

- one environmental Ray Perception Sensor, a boolean that tells the agent that the enemy is somewhere in sight, maybe in a blind spot and the standardized number of degree between the agent direction and the correct direction toward where the enemy is. The number of inputs in this case is acceptable, but the fact the sensor is not able to reveal the target position fools the agent, making it ignoring more or less the two other inputs;
- a variant of the previous one where, instead of the environmental sensor, a 180°FOV was provided and it was able to perceive the target too. So the two manual observations where there just to cover the blind areas. For some reason, this solution lead to a sub-optimal model that was not able to focus for too long on the enemy;
- just one Ray Perception Sensor with 180°FOV able to detect both walls and enemy (as the previous ones).
- iterations of previous sets with stacked vectors, to achieve a simple kind of memory of the past observations, increasing the number of inputs.

All those steps were tried plenty of time with several hyper-parameters set and different network structures in a simplified learning environment (a small area with just some corner to check). I tried Imitation Learning to create a good starting point for the *pn*, mixing both Behavioral Cloning and GAIL with PPO's reinforcement learning, spending a lot of time in recording as many demonstrations as I could each time a new change affected the observation space (since the demonstrations preserve the observations of the expert). I tried a lot of network types, spacing from recurrent neural networks to plain ones, using normalization or avoiding it. The solution seemed to be really far.

This problem required me to reduce the action space as well. The first cut applied targeted the strafe branch. From that moment, the agent was able only to go forward, backward and rotate itself. All the previous solutions

were tried again but the results were not good. A second reduction was performed, removing the backward action. Now the agent can just move forward and rotate. Some better results appear with all the previously mentioned tests. With a bit more complex network structure, the solution with one sensor with 180° FOV looks like the outstanding one: the agent was able to perfectly complete its task in the simplified field. A better performance was achieved limiting lateral perception angle to 80° instead of 90. Through a slow curriculum learning process, this model could reach the optimal desired form.

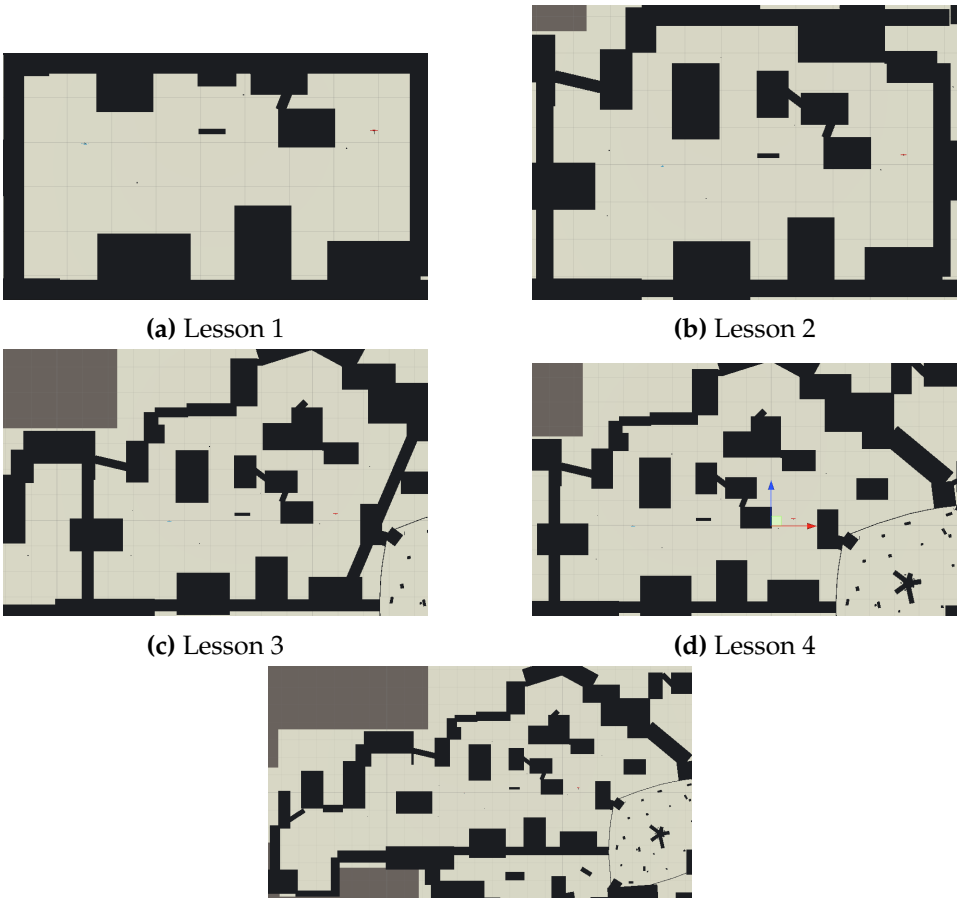
The final reward function $\bar{r_{explorer}}(a_t, s_t)$ is defined as follow:

- +1 for keep the enemy exactly in the agent forward direction for at least n_e steps;
- $+1/n_e$ for every step the agents keeps the enemy exactly in its forward direction;
- $+1/n_e$ for every step the angle between the agent forward direction and the correct direction is less than 1 degree;
- $-1/n_w$ for each step the agent stays in collision against a wall;
- -0.0002 for each step to force the agent to take a decision;
- -1 if the number of steps the agent stayed in collision with a wall is grater than n_w (only for the first two curriculum lessons);

where n_e is the number of steps the agent should look precisely toward the enemy, while n_w is the maximum number of steps the agent can touch the walls in the first two lessons, and a simple coefficient for the penalties from the thrid phase on.

A previous version of $\bar{r_{explorer}}$ rewarded the agent if the rotation it was performing decreased the angle between it and the enemy (when found), but it was removed because it was both unnecessary and harmful for the agent, since it started to make big rotation over the target in order to gain more rewards.

Five curriculum lessons were designed, where the map increases on each



(a) Lesson 1 **(b)** Lesson 2

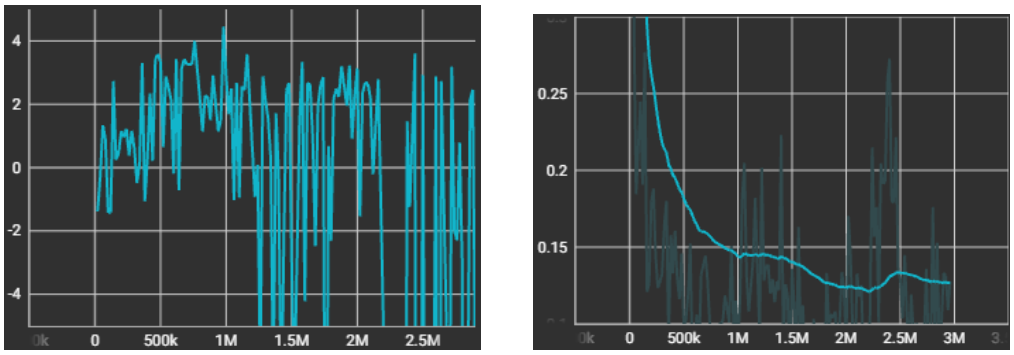
(c) Lesson 3 **(d)** Lesson 4

(e) Lesson 5

Figure 3.9: Curriculum Lessons

of them. As said previously, an automatic curriculum learning (CL) was not possible, since the reward function is not stable (Figure 3.10a) (it depends on how many walls the agent requires to touch to complete its task in an optimal way and how many exploration steps are required to find the enemy, since the map is huge) . The manual CL consisted on visually check the quality of the policy in that lesson and initialize a new model from the previous weights enabling the new phase. The difficulty increment is showed in the Figure 3.9. In addition, before swap to the next stage, the n_e is also incremented in order to teach the agent to look at the target more time as possible.

The exploration part, after a deep analysis of the literature and lot of trials in the ML-Agents study phase, was quietly easy to achieve. The key role is played by the Random Network Distillation module that pushes the



(a) Cumulative reward over steps

(b) Policy loss over steps

Figure 3.10: Explorer Plots

agent to the exploration of newer states with intrinsic rewards. Finding a good network for RND was not too easy, but this problem was faced during the extensive tests on the ML-Agent framework; so the design of this secondary model was quietly rapid for this final brain. It consists in three fully-connected hidden layers with 512 units each, without normalization.

The final model for the *Explorer* is described in Figure 3.11.

Before and after joining the two brain components, *Explorer* has been trained alone again to make it faster. A slightly adjusted reward function has been applied, penalizing the agent with the double amount of negative points for each step. A tiny enhanced version slowly started to appear. After about 500 hours of extra training, the exploration seems more stable, the agent learned to keep its "forward" output for most of the time, denying the physics controller to slow down its movement, and it understand to be faster in turning itself toward the enemy when he is in sight, without losing time and without let the torque friction decelerate too much its rotation. The final result is a great "hide and seek" agent, able to find its target almost everywhere (even in narrow corners, where a human player could decide to "camper") in a relative small amount of time and in the complex and wide environment like the game's map.

3.2.2.3 Final Training and Hyperparameters

With the *Shooter* and *Explorer* models ready and well trained with PPO, a last small group fine-tuning is required. Since the two brains must cooperate to

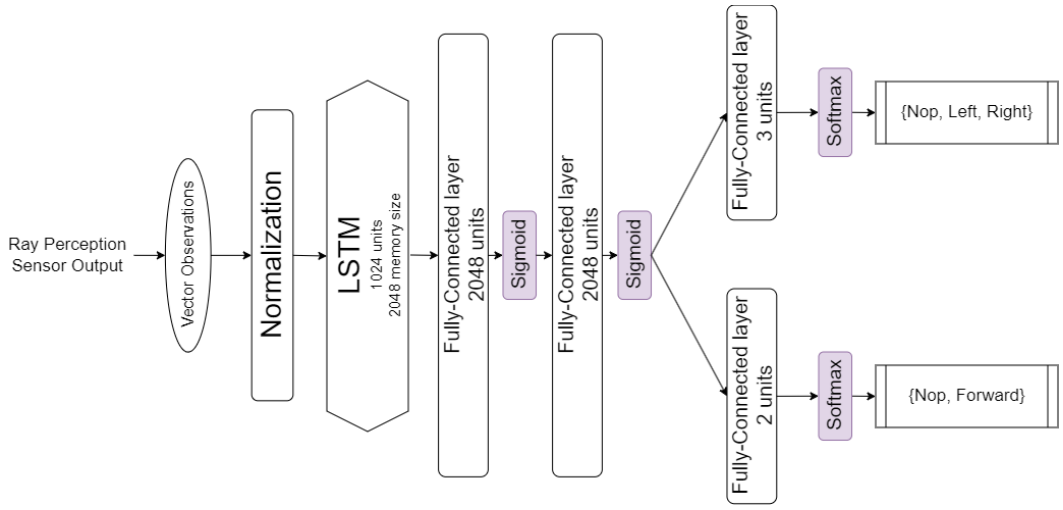


Figure 3.11: Explorer model

kill the target, it is time to use MA-POCA. It was necessary to re-initialize the optimizer, since MA-POCA works different from PPO, but the two models are already correctly weighted and the policy cannot change too much. The extrinsic reward function is the union of $r_{explorer}$ and $r_{shooter}$, while the RND intrinsic function is the same as the one used for the *Explorer* model. At this point the *Self-Play* module was used, making two agents play against each other in an AI deathmatch. In a limited number of steps, the agent became faster and her aim more accurate, showing some new strategies have been acquired (Figure 3.12).

Figure 3.12: Final Training Elo Plot
Orange *Shooter*;
Gray *Explorer*

Hyperparameters All the trainings were performed using up to 32 concurrent environment instances, collecting more observations in a relative small amount of time.

I tried several sets of hyper-parameters that pushed the training to a better or worse situation. In order to address the agent to force itself to collect the rewards it finds on its way, I set the gamma hyper-parameter to be almost 1: do not think on future possible rewards, all you can achieve is already in your sight. In this way, it learned to fast focus on the enemy, avoiding the search of a new hypothetical reward source. It follows the complete and final set of hyper-parameters used for the final training.

- Explorer:
  - `batch_size`: 2048
  - `buffer_size`: 20480
  - `policy learning_rate`: $3.0e^{-4}$
  - `beta`: $1.0e^{-2}$
  - `epsilon`: 0.2
  - `lambd`: 0.95
  - `num_epoch`: 3
  - `learning_rate_schedule`: constant
  - `extrinsic reward strength`: 1.0
  - `extrinsic reward gamma`: 0.99
  - `rnd strength`: 0.05
  - `rnd gamma`: 0.99
  - `rnd learning_rate`: $1.0e^{-4}$
- Shooter:
  - `batch_size`: 128
  - `buffer_size`: 2048
  - `policy learning_rate`: $3.0e^{-4}$
  - `beta`: $5.0e^{-2}$
  - `epsilon`: 0.2
  - `lambd`: 0.99
  - `num_epoch`: 3
  - `learning_rate_schedule`: linear

  - `extrinsic reward strength`: 1.0
  - `extrinsic reward gamma`: 0.99
- Self-Play (in common):
  - `window`: 10
  - `play_against_latest_model_ratio`: 0.5
  - `save_steps`: $2.0e^4$
  - `swap_steps`: $1.0e^4$
  - `team_change`: $1.0e^5$

3.2.3 An attempt to make the agent damage aware

One big problem with the Eve's brain's structure is the impossibility to give her the perception of someone who is shooting at her out of her field of view. The human player is able to understand that someone is shooting at him thanks to sounds and visual effects, but the *Explorer* network observes only what is shown in front of it. From the human player's point of view, watching the enemy going for her path completely ignoring the danger is unrealistic. Since the learning time required for the *Explorer* to became super-reactive was massive, re-train a new network with a new observations was not an option. Transfer-learning is not allowed by ML-Agents, and an external change to the network (like in a custom python script) would make the model incompatible with the *Unity Inference Engine* (so no cross-platform support and more GPU workloads, as described in the optimization section 5.1).

After some further studies on the framework, I found two possible solutions:

- ignoring the problem, since, from statistics performed during training and at test time, the probability that a human successfully hits the agent on great distances is really low, while with a closer distance, Eve usually rotates herself a bit and she is able to locate the enemy. The only exception occurs when the enemy lies precisely in the opposite of her noisy forward direction, but this is a risk case that I could accept;
- forcing Eve to rotate until her field of view includes the enemy posi-

tion. This event will occur only when and only if someone is shooting at her and he does not hide after the shot. The negative side of this solution is that it sounds like a cheat, but if we think at it, it is not, since she will not be forced to turn until the enemy is in front of her, but just to bring the foe in one of the extremes of her FOV. This means that she is free to decide if intentionally keep her rotation or ignoring the enemy, which will be $FOV/2$ degrees away from Eve's aiming direction.

In order to implement the second solution, the `WriteDiscreteActionMask` function was used.

It follows a slice of the *Explorer*'s script, so it has to be considered from the Eve's point of view.

- `beingHitted` is a boolean variable that is set to true when Eve's enemy is shooting at her;
- `(enemyPosition - cameraPosition).normalized` is the versor which connects Eve's position to the enemy's position;
- `HittablesLayers` is a `LayerMask` which specifies the hittables layers, such as walls and the players hitboxes;
- `TargetLayer` is a `LayerMask` that specifies which is/are the layer/s where the enemy's hitbox/es lies;
- `TargetLayer==TargetLayer | 1<<hit.collider.gameObject.layer` is a bit-by-bit check of the mask that tells if the Raycast's hitted object is the enemy or if the direction between Eve and the opponent is occluded by something. This is necessary in order to understand if the other player hid himself after the shot, so Eve will not rotate toward him since she does not have idea where the shot came from;
- if the previous condition is satisfied, then all the *Explorer*'s actions are masked with the exception of one side rotation, which depends by the `directionAngle`, a variable that indicates the signed angle between Eve's forward direction and the enemy direction. If this value is grater than zero, then it means that the opponent is located to her current

right direction, so the left rotation is masked, if it is lower than zero, then the right rotation is masked. This until the enemy enters her FOV, then is Eve that choose what to do.

```
public override void WriteDiscreteActionMask
    (IDiscreteActionMask actionMask)
{
    if (beingHitted) {
        if(!(Physics.Raycast(cameraPosition,
        (enemyPosition - cameraPosition).normalized,
        out RaycastHit hit, Mathf.Infinity, HittablesLayers)
        && TargetLayer ==
        (TargetLayer | 1 << hit.collider.gameObject.layer)))
        {
            beingHitted = false;
        } else {
            if (Mathf.Abs(directionAngle) > FOV) {
                actionMask.SetActionEnabled(0, 1, false);
                actionMask.SetActionEnabled(1, 0, false);
                actionMask.SetActionEnabled(1,
                    directionAngle > 0 ? 2 : 1, false);
            }
        }
    }
}
```

This new model structure is shown in figure 3.13.

With this new network, a novel training session started. I was interested in keeping the original *Explorer*'s optimizer state, so I used ppo once again, initializing the weights to the latest ppo version of this brain component. In order to force the agent to check corners, I disabled self-play - to make the dummy target stationary - and I increased the probability that the enemy spawns in those areas. In order to train this advanced behaviour, the

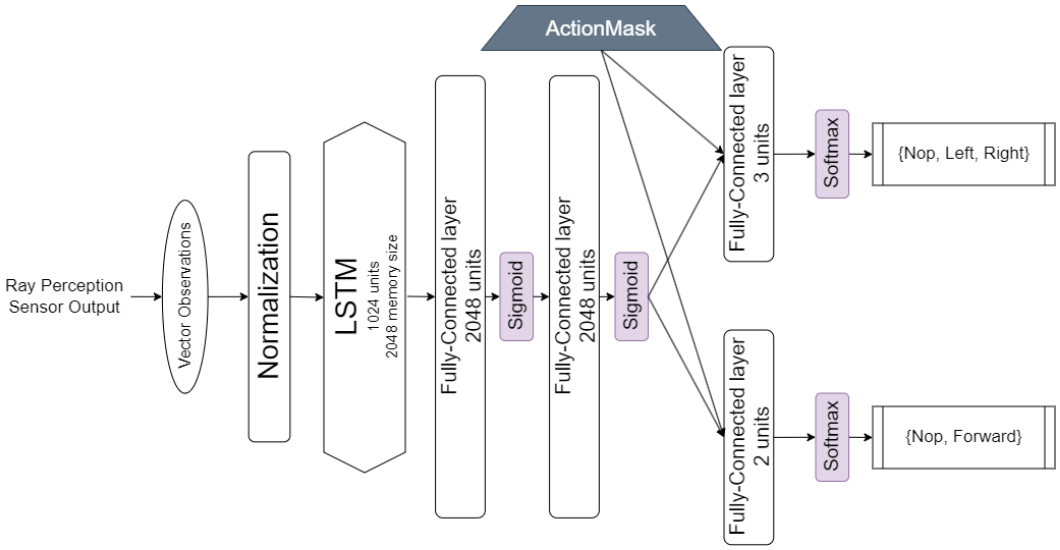


Figure 3.13: Explorer model with action mask

dummy agent should "shoot" against the AI somehow. To do so, I made this entity always re-spawn pointing toward the largest empty area that surrounds it, increasing in this way the probability to "see" the real agent's back. Furthermore, I gave it a small FOV (15 degrees for side) where to use some `Raycasts`. In case one of these rays hits the AI outside its FOV, the agent would be warned about an hypothetical shot, making it start its "forced turn routine". However, the probability that the dummy target hits the back are not so high, so it is really hard to completely train and test this kind of conduct. The learning session lasted for some days and it will be pursued even after this application's first release.

Since this second solution is still experimental and it was not exhaustively tested, the final application gives the user the opportunity to chose which Eve's version he wants to use.

4. Chapter 4

4.1 An Overview of the other game's elements

In this section I will provide a small picture of the other elements that rule this game, such as the human user inputs, weapons settings and so on.

4.1.1 Human Inputs

To play this game, users must own a VR system with not only the HMD but also the two hand controllers, no matter the system brand. For the sake of simplicity I will use as example the Meta Quest 2 and the Valve Index [49] (Figure 4.1).

The movement in the virtual environment can be achieved by physically walking in the real world, making the HMD tracker works for this purpose, and/or using the secondary joystick. A similar concept is for the rotation: it is possible either to physically turn in real world or using the primary js. These are the two types of movement input that Eve uses. Furthermore, the headset is able to track all the three Cartesian axes, so human players can crouch in real life and their action will be mapped in the virtual environment.

Controllers are tracked, so any motion the user performs in real life, it will impact the position of the virtual hands, under the condition that move is physically possible in both the simulated and the real worlds. Using the grip buttons it is possible to perform a grab action, called *Grab*, on a grabbable object, making the hand gameobject and the grabbable related. For the most common remotes, like the Meta Quest ones, the grip button is a physical toggle, while for the Valve Index's system it is a capacitive sensor, similar to a touchpad. All the grabbable and the grabbers (such as hands, sockets and bags) contains two event lists for the *Grab* action, one when the

action is performed and one when release happens. They are useful in case something must be activated when there is a grab state change. An example will be provided in 4.1.2.

Some grabbables have some intrinsic capacity that the user can exploit, for example, a flashlight can be turned on and off. This action is called *Activate*, and it also comes with two events: `OnActivate` and `OnDeactivate`, respectively for the pre-post activation phases. This action is induced by the "trigger" backside button.

Many controllers are built with two buttons each, but some VR system provides just one for remote. To better generalize and allow a multiple range of ecosystems, I implemented a behaviour only for the primary button, common in every VR SDK. More specifically, in this case, it is responsible to manage the jump in the virtual world. However, in the AI-based mode, this action is not allowed for the moment.

With the new Unity Input System, it is trivial to link a physical action to the name of the relative virtual one. Using the OpenXR framework, it is even possible to link functions' names with actions. Furthermore, if some action is related to a grabbable object, relative events will be triggered if and only if that grabbable is controlled by the user during the call time. For instance, pressing the trigger with nothing in the hands, will not turn on the flashlight lying on the floor, vice-versa, if this was in the user hands, it would work as expected.

Last but not least, it must be considered the set of "inputless" actions: this project includes some kind of actions that can be only performed making a particular movement with the hands. For example, there are no physical buttons requested to release the magazine attached to a weapon. To reload the gun you can chose either to grab the current magazine and pull it out or to hit the old one with a new one, using a certain force.

At the end, it was prepared the finger tracking. More or less, all the recent controllers includes a capacitive sensor on each of their buttons. So, the thumb position is inferred by the area it is touched on the main board of the controller (where the joysticks lies), the index finger by the trigger

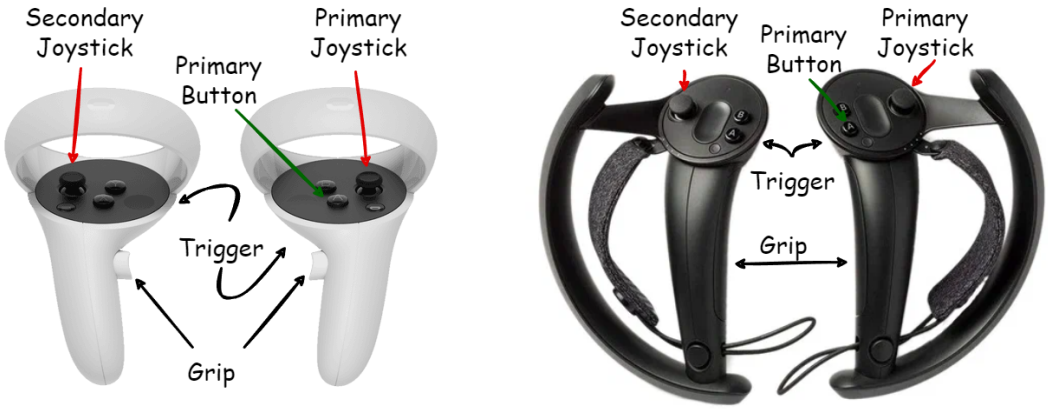


Figure 4.1: Example Controllers

button, and the other three are in general considered as one tracked entity, by the grip button. In the Valve Index's controllers, this tracking is more precise, since all the area where the grip button should be located, is a large sensor, designed for an accurate single finger tracking. In fact, if you are using this headset, the virtual avatar will be able to replicate with a good precision how you are moving all your fingers separately.

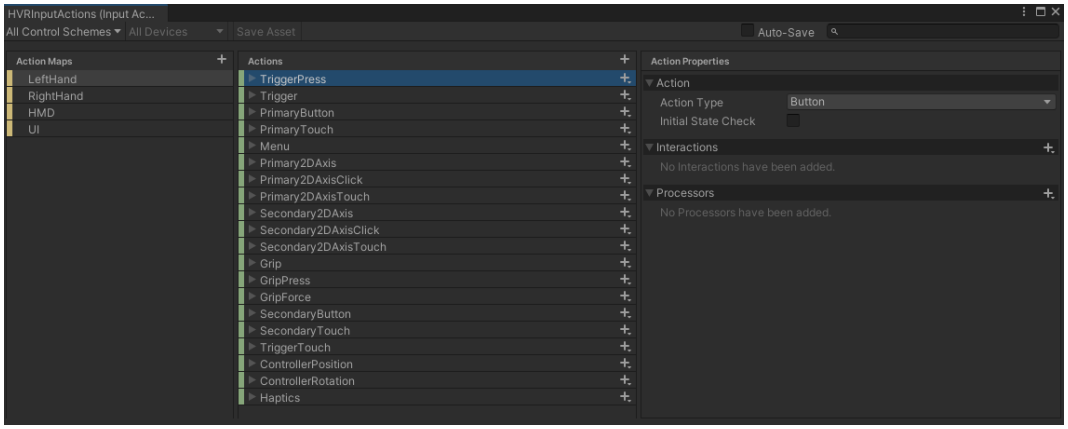


Figure 4.2: Motion-Action Map File

4.1.2 Weapons

Only two types of weapon are available with this release: an automatic rifle and a semi-automatic pistol. The creation of a new gun requires lot of time and studies: in order to give a realistic feeling to the user, the 3D model must be lifelike, the hitbox must be a good trade-off between accuracy and computational costs, the type of recoil must recall the true motion of a firing

gun and so on. Let us assume the 3D model is delivered perfectly as we want and ready to be made alive, and its shape is not too complex and can be approximated with some cubic hitboxes, that are not computational expensive, the last cited requirements remains not trivial. For each weapon it must be designed two types of recoil, one for a one hand grab and the other for the two hands case. To model the recoil I used information gained by an interesting recoil forces dataset [50] collected utilizing a wrist worn accelerometer. Taking as example the entries relative to the type of weapon I was working on (such as automatic rifles or handguns) and watching some demonstration videos about similar weapon barrages, I was able to recreate something similar in the virtual simulation.

In figure 4.3 it is shown the rifle that the AI uses, including the approximated hitbox.

Each player will have a limited set of magazines per weapon in their left pocket. This shows a clear example of a possible use of the actions' events: when a player grabs a gun, `OnGrab` method will spawn the correct type of magazine in the pocket, while `OnRelease` will despawn everything, in order to reduce the rendering and collision detection workloads. In case a player grabs two guns, one for each hand, what will be the magazine contained in the pocket? The answer is always the type of the second entity grabbed: to reload a weapon it is necessary to use both hands (one to keep the gun and the other to pick the new magazine), so there will never be a scenario where the bag contains the wrong type of ammo. Another example of the `OnGrab` and `OnRelease` functions is the following: some type of rifles uses magazines that remain outside the body of the weapon itself (as the one showed in Figure 4.3). When these external structure is attached, a new hitbox is required, and this job is performed in those methods.

4.1.3 The Map

For this project, only one map is provided, the one where Eve learned her task, More immersive and suggestive environments will be added in a near future. The one included with this first release consists in a futuristic dystopian

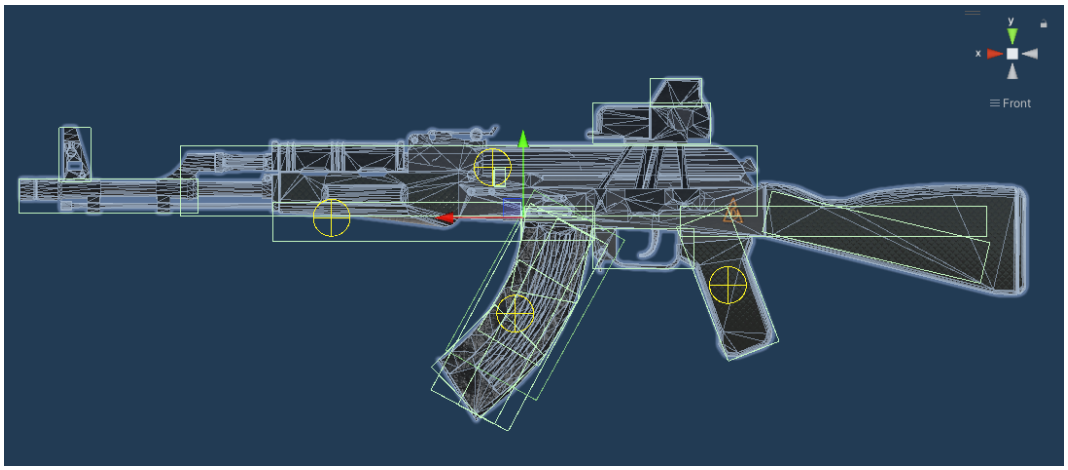


Figure 4.3: Example of weapon

cybernetic city. The player can enjoy this charming environment and all the elements that the real world cannot offer yet, such as giant mechs, self-controlled security drones and so on. The structure of the map was designed to make all the players at the same level of difficulty, regardless where they spawn, avoiding deep corners and too different height walkable levels. This last property was thought specifically for this first prototype of artificial agent, making it enjoy a fair match with its restricted vertical field of view. The map's bounds are showed in the last curriculum lesson in the previous chapter (Figure 3.9e). A set of high-resolution view examples of the final aspect are showed in figure 4.4.

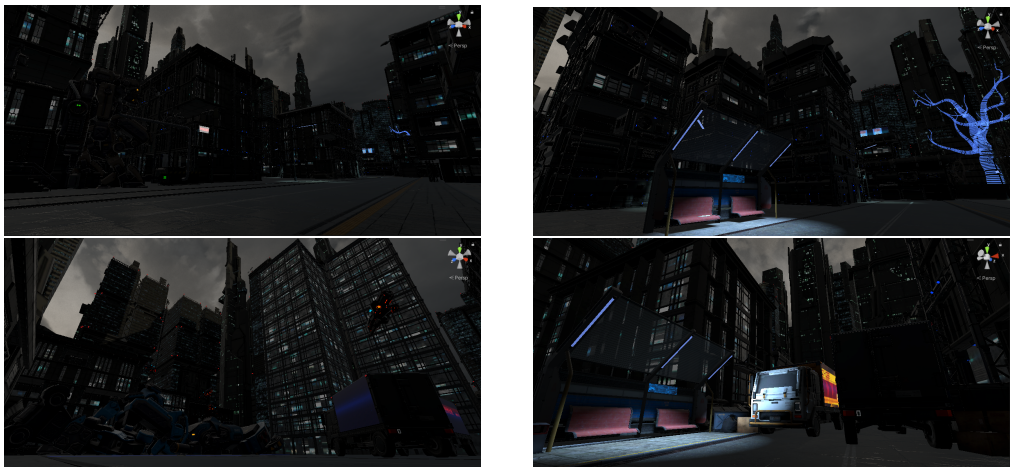


Figure 4.4: The Map

5. Chapter 5

5.1 Optimizations

It is well known that PCs for virtual reality require some high specifications. In addition, to bring the user to a new way of interaction and gaming experience, the development of VR devices has further boosted the demand for high-performance hardware. If you add a superior quality graphic and a realistic physic simulation, the hardware problems start to be more prominent, with low FPS and glitches. To make matters worse, my game includes two neural networks to query, burden more on the GPU. It is required a massive software optimization step. This work was analyzed and branched in three parts: *network query optimization*, *CPU optimization* and *GPU optimization*.

5.1.1 Network query optimizations

Shooter and *Explorer* models are already the smallest possible and a query through them is acceptably fast. However, the problem is the workload the GPU has to perform for an inference each physical step (around 60/70 times per second). A graphic card plays the hardest role in a VR setup and it should not be commissioned with other tasks. So, the only two possible optimizations applicable in this case are the following:

- use the *Unity Inference Engine* as inference process. This grants not only a cross-platform integration for neural network queries, but also a faster response from the brain, since the possible model blocks are limited to some types of layer, and the UIE is optimized to query those types;
- use CPU + Burst as inference device. The processor is way slower than the GPU in terms of inference time, but C# provides a special

algorithm, called *Burst*, that makes network's calls faster. Correctly set the inference device to the voice *Burst*, will relieve the GPU from the inference job, while an acceptable query time is kept.

5.1.2 CPU optimizations

CPU workload is strictly related with the GPU one. Methods for the two hardware optimizations are linked: apply that method to improve performance of that component, and you will improve a bit the other one too.

5.1.2.1 Static Batching

Static batching in Unity provides performance improvements for both the GPU and CPU, with a greater impact on CPU performance. By marking stationary objects as static, Unity combines them into a single large mesh, resulting in faster rendering. This optimization reduces the number of draw calls and decreases the amount of CPU processing required. Static batching offers benefits beyond rendering efficiency. By tagging objects as static, Unity recognizes that these objects will not move and excludes them from physics calculations. This further reduces the computational load on the CPU, particularly when dealing with complex physics simulations. If your application is primarily limited by CPU performance, marking more objects as static can be an effective technique. By reducing the number of individual objects that require physics calculations and optimizing rendering through static batching, you can improve overall application performance and alleviate CPU bottlenecks.

5.1.2.2 Adjusting the Rendering Method

Unity supports three rendering methods for virtual reality:

- Multi-Pass;
- Single-Pass;
- Single-Pass Instanced.

Multi-Pass (Figure 5.1). An easiest explanation can be done with an ex-

ample: the Google VR. In this context, Unity needs to render a scene twice because there are two textures, one for each eye. However, to optimize performance and avoid duplicating unnecessary work, Unity employs various techniques. To prevent duplicating work, it identifies elements such as shadows that do not require rendering twice and avoids redundant calculations. However, for most objects, it still performs a multi-pass rendering approach, rendering each entity twice but creating the scene graph only once. This technique ensures more accurate lighting and visual consistency between the left and right eye views. Although multi-pass rendering comes at a computational cost. Since the two renderings do not share GPU work across textures, this method is considered less efficient in terms of GPU utilization. Despite this, multi-pass rendering is compatible with a wide range of devices and is a common rendering path used in Unity for Google VR applications.

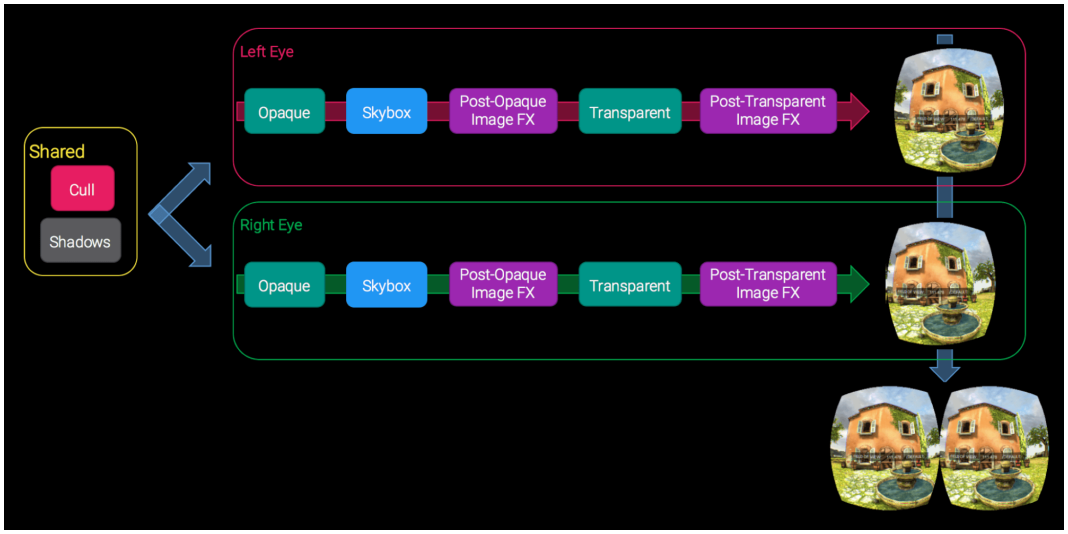


Figure 5.1: Multi-Pass rendering

Single-Pass rendering combines the two eye textures into a double-wide texture, resulting in faster CPU performance. It requires fewer processor computations by going through the scene graph only once. However, this approach necessitates additional GPU state changes, which can impact this component's efficiency.

Single-Pass Instanced (SPI) (Figure 5.2) is a powerful technique that offers simplified integration and improved performance in rendering. It ad-

dresses the CPU and GPU overhead concerns by reducing both more effectively than single-pass rendering. Similar to this last, SPI lowers CPU overhead by reducing the number of draw calls. This optimization streamlines the rendering process, resulting in improved CPU performance. However, the key advantage of single-pass instancing lies in its ability to further minimize GPU overhead compared to traditional single-pass rendering. By leveraging instancing, the GPU can efficiently process multiple draws in a more optimized manner. Not having to change the viewport between draws, which is required in traditional single-pass rendering, reduces state updates and enhances GPU efficiency. SPI is the most performant among the three described (as shown in figure 5.3), but it is not available on all devices.

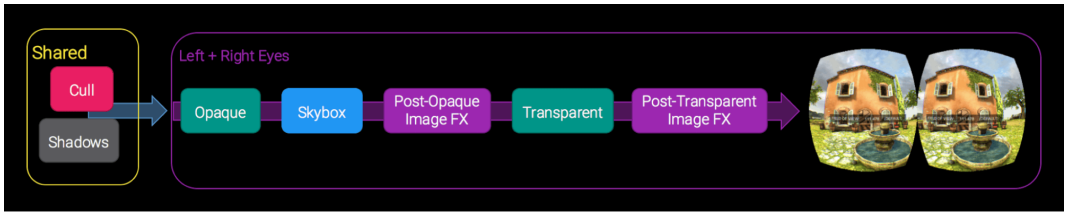


Figure 5.2: Single-Pass Instanced rendering

The choice of rendering path depends on the specific requirements of the VR application, the performance capabilities of the target devices, and the desired balance between visual accuracy and computational efficiency. In my case, I used SPI since it is fully compatible with Oculus.

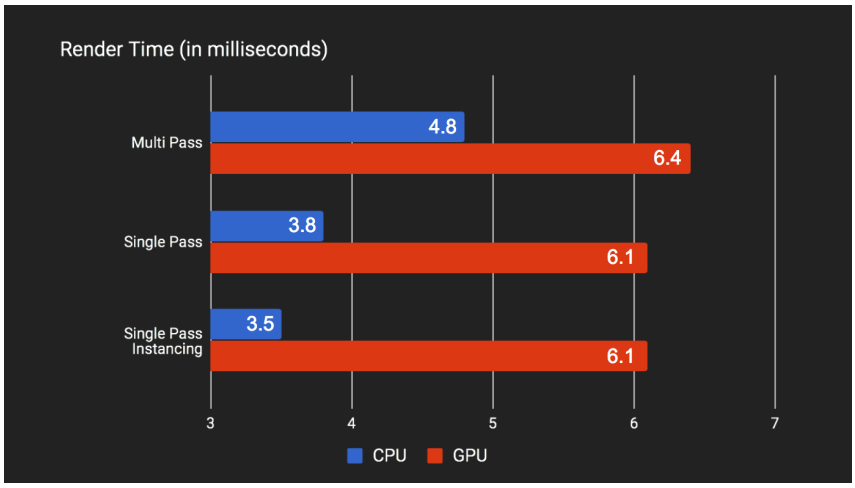


Figure 5.3: Rendering Method CPU/GPU performance comparison [51]

5.1.3 GPU optimizations

The greater amount of work is related to the most important component in VR setups: the GPU. The first steps were achieved during the optimization phases above, in particular making more object possible marked as static and redirecting the inference job to the CPU.

The next steps work around the graphics optimizations.

5.1.3.1 Quality Setting Tuning

Quality settings in Unity allow you to control the graphical quality of the rendered objects in your application. These settings come pre-configured with options such as pixel light count, light map resolution, and other parameters that directly impact the visual fidelity and performance of your scene. By adjusting the quality settings, you can choose between maximizing graphical quality or optimizing performance. Increasing the first one enhances the visual fidelity of objects by enabling features like higher resolution textures, more detailed lighting effects, and increased particle effects. This results in a visually stunning experience but can potentially impact performance by requiring more GPU resources. On the other hand, maximizing performance settings prioritize efficient resource usage and rendering speed. By lowering quality settings, such as reducing the pixel light count or lowering the light map resolution, you can achieve better performance by reducing the computational workload and memory requirements. This is particularly useful for applications targeting lower-end hardware or aiming for smoother performance on a wider range of devices. The choice of quality settings ultimately depends on the specific requirements and the hardware capabilities of the target platforms. Striking the right balance between graphical quality and performance is crucial to provide an optimal user experience and ensure smooth application performance.

5.1.3.2 Light Baking

Light baking (Figure 5.4) is a technique that helps to decrease the computational resources required for rendering a scene. It involves pre-calculating

Figure 5.4: Before and after bake

the illumination of the scene before run-time, resulting in no additional computational overhead for baked lights. In Unity, by default, each object is rendered for every light that affects it. This means that if an object is affected by five lights, Unity will render the object five times when it's in view. This process can significantly increase the tri count and the number of draw calls, leading to decreased application performance. By utilizing light baking, you can effectively reduce the number of tri and draw calls since these calculations are performed prior to run-time. To bake all the lights in the game's map with medium lightning quality settings, the time required with my hardware was 21 hours. This optimization step was the most important one from the FPS' point of view: it allowed a massive increment of the graphics performances, reaching up to 90 FPS from a previous value of 27.

Occlusion Culling

Occlusion Culling (Figure 5.5) is a feature in Unity that prevents the rendering of objects that are not within the view of the Camera. By default, Unity excludes objects that are outside the camera's viewing frustum from the rendering process. However, objects that are blocked by foreground entities are still rendered, resulting in unnecessary overdraw or pixel overlap. Overdraw occurs when pixels are drawn multiple times, layered on top of each other, which adds unnecessary computational load. To address this issue, it is possible to utilize Occlusion Culling, which effectively "culls" or removes objects that are obstructed by other objects in the scene. By analyzing the scene's geometry and visibility, Unity determines which objects are

not visible and can be safely excluded from rendering. This optimization technique helps to minimize overdraw and reduces unnecessary rendering work, improving overall rendering performance and efficiency.

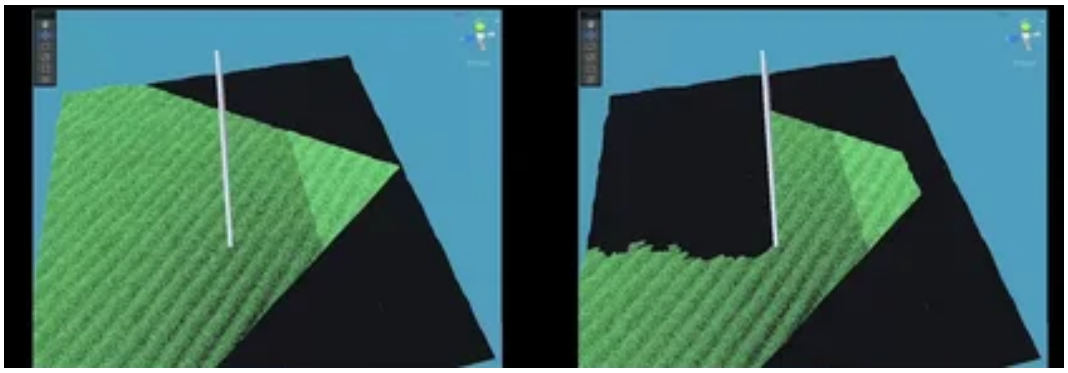


Figure 5.5: Before and after occlusion culling, from the Unity3D Subreddit

For more information, Unity Technologies wrote a step by step optimization guide on VR and AR applications [52]. Additional studies were on [53], [51].

5.2 Performance

5.2.1 Game Performance

In a virtual reality experience the most important quality metric is the FPS. A poor FPS VR application defeats its "full dive" role, since the player could not appreciate well the virtual environment, while the motion sickness could gain the upper hand. It is important to keep the minimum number of frame per seconds at least above 60, and this was the objective reached with all the GPU optimization shown above. In particular, before the long process for light baking, the game was not able to reach 30 stable FPS. This optimization step was the most important among all the others, since alone was able to bring the overall frames up to the upper limit set, namely 72. This value, tuned during the development of the original *DefaultVR*, allows the `FixedUpdate` to loop 72 times per second, regardless the speed of the `Update` (that keeps 72 executions per seconds as well on the most performant tested setup), necessary for a good physics simulation in the multiplayer modes.

Table 5.1 shows the number of FPS divided by the VR system and hard-

ware specs used for that experiment. Obviously, particular stressful scenarios (such as a long fire fight in a heavy graphics effects area) could cause a brief FPS drop (during tests, the max drop detected was not below 60 FPS), but this is not a real problem, since the physics routine keeps its normal throughput, while the smoothness of the video is restored in a very short amount of time, usually imperceptible.

It follows the specs list for each computer used for testing the application.

- Hardware 1:
  - CPU: Intel Core i9-9900k
  - GPU: NVidia RTX 2080 Super
  - RAM: 32 GB Dual Channel
- Hardware 2:
  - CPU: Intel Core i7-10870H
  - GPU: NVidia RTX 3060 for laptops
  - RAM: 32 GB Single Channel
- Hardware 3:
  - CPU: Intel Core i7-7700HQ
  - GPU: NVidia GTX 1070 for laptops
  - RAM: 16 GB Dual Channel

	Hardware 1	Hardware 2	Hardware 3
Meta Quest 2	68 to 72 FPS	Not suitable	Not suitable
Oculus Rift CV1	70 to 75 FPS	Not suitable	Not suitable
Valve Index	67 to 70 FPS	Not suitable	Not suitable

Table 5.1: FPS per VR systems and PCs

The main problem is the high specs requirement necessary to run smoothly the whole application: as it is possible to notice from the above table, only the most performant hardware available was able to run in an acceptable way the program. After a long secondary optimization analysis, it was found that the problem was not in the heavy map graphics and VFX, but in the Unity Inference Engine during the query of the *Explorer* network, really resource-hungry while performing the forward step, since the NN is not small as the Barracuda library expects. This problem points a massive issue with the inference engine and/or the Python external API. In order

to play against Eve, the minimum requirements for the PC is a RTX 2080 Super linked to a powerful CPU, at least an I7 of 9th generation, broadly comparable with the I9 9900K. Fortunately, recent VR ready PCs present these elements or even better ones, so the requirements are not inhibitors.
The slightly difference in average values between the headsets considered are given by the computation cost that those devices requires: the Oculus Rift, namely one of the first VR headset system available with tracked controllers, works with an internal resolution of 1080x1200 px per eye, the Meta Quest 2 work with 1832x1920 px (per eye too), while the Valve Index uses two 1440x1600 displays with a huge amount of data sent to the PC with its finger tracking controllers values. The weakest device requires less graphic data to be computed, so it receive a slightly higher amount of FPS, while the other two expects some computation more.
As a general and exhaustive information, the other two hardware analyzed reported a final FPS value included between 25 and 45 FPS, unplayable and with possible motion sickness problem to non-experienced VR users. However, this issue is not present in the secondary modes included with the application, such as the "Range vs AI", which makes the human player able to challenge Eve to a mini-game where the winner is the one who kills more dummy targets in the range with a fixed amount of time available. The *Shooter* network is slimmer and faster to query, enabling an excellent gameplay lag-free with more then 90 FPS on all the analyzed hardware.

5.2.2 Eve's Performance

An adequate way to perform a quality evaluation for the trained agent is to compare the number of episodes successfully completed and failed.

All the tests that do not concern a human player were performed running for 6 hours straight and with 32 parallel instances, regardless of the type of task (exception for Eve's whole brain, that used just 4 parallel instances in order not to introduce lag, since 4 networks to query for instance starts to be computational expensive). This is the reason for the difference in terms of episodes' number. Human-based tests are more difficult and

expensive (in term of time), so the total amount of episodes is significantly lower than the one automatically performed.
All these tests were run on the "Hardware 1", described in 5.2.1.

Table 5.2 collects the total number of episodes, the number of completed ones as well as the expired occurrences (the failure case, since there is not a precise event that closes the episode with a bad response) for the *Explorer* component only. The same table shows the results for both the AI vs Dummy and AI vs AI tests. In this second case, the voices "AI1 Won" and "AI2 Won" tells us how many episodes a certain agent found the other agent for first. An episode is considered expired when it reaches 50k environmental steps. The first noticeable thing is the difference between the total number of episodes between the two test cases. In a certain way, this is almost obvious, since in the first scenario the agent have to search along all the map and, in the worst case, more than one time, if it miss the one narrow corner where the dummy target lies; while in "AI vs AI", there are two agents, both active searchers that never hide, so the convergence happens in lower time.
Analyzing the results, it is possible to notice that only 4 over 2411 episodes expired, so the *Explorer* has a 99.84% of successes. In the second case, it is possible to understand the speed the agent solves its task, since in a symmetric and fair target searching task, it (they) was able to find the enemy the double of occasions (wrt the previous test) in the same amount of total time. In addition, looking to the number of episodes won by each agent, the two values are very similar between them (almost equals, talking about of large numbers): this confirms the fairness of the test.

Table 5.3 shows the results for the *Shooter* model only. It collects the number of times the agent slays the dummy agent in the range, the number of times it consumes the whole magazine without reaching its goal and the average number of bullets used for each kill, everything ordered and divided for sets of distances where the dummy target spawned. Magazines contain 25 bullets and, at the beginning of each episode, they are automatically refilled. The enemy spawns in a random moment between 0 and 10 seconds from the start of the episode and its position is random on all the

three space axes, y included since it comes with an always different crouch level, exposing the head at diverse altitudes. As previously mentioned, this brain component never fails in the range, meaning that no magazine is emptied without score a kill. In 42818 episodes, the agent struck 42818 kills with an average use of 3 bullets for each enemy. Observing the other table rows, it is possible to notice that more the target is far and more bullets are needed to execute it. This is because, on long distances, it is easier to hit the body, which presents a grater area respect the head. The *Shooter* observes the angle between the vector "aiming direction" and the vector which connect the agent y-axis to the target's one, parallel with the floor. This means that the cited input can only lead to find where the enemy is sideways, but it does not provide any information about where the head is. Obviously, during training, it was learned that the head is above body and legs, and below the upper "miss area", but even a very small movement, on long distances, results in a huge change, making the task a bit harder. However, an average of 3 rounds to strike a kill in all the possible tested distances (between 0 and about 60 meters) is a great achievement.

Table 5.4 displays the statistics for the entire Eve's brain, the cooperation of *Explorer* and *Shooter*. The structure includes the number of episodes won by Eve and the number of the ones won by her opponent (which can be a human or another AI agent), divided by the matches AI vs AI (the second one is the immediately previous model checkpoint) and AI vs Human. In the first test I removed the limited magazine constraint, allowing both artificial players to challenge each other without the need to reload. This constraint alleviation is not kept in AI vs Human and, in general, the official deathmatch mode. A strange behaviour occurs in the first test, the AI vs AI scenario. Even if both entities are querying similar networks, the one whom id is 0 performed better. The only two explanations to this could be either that in those 6 hours, the AI1 was luckier than the second one or that the newer version is much better than the previous one; or maybe both. Regarding the AI vs Human, it is clear the skill gap between the agent and the player. This test involved 5 beta testers, one of which was an experienced VR user. Each game lasted for 10 minutes (a global timer was there

to ensure this constraint and to terminate the simulation upon expiry of the period) and, as in the real Deathmatch mode, both AI and Human had unlimited magazines supplies. What these tests revealed is that sometimes, in particularly stressful situations for the hardware, the models inference was a bit slower than the usual, causing a poorer performance for that single fire trade. However, for most of the time, Eve performed much better compared to her opponents. For a total of 30 matches (6 for player, namely one hour of trials for each tester), Eve was defeated 5 times with a tiny score difference. The worst score she obtained was 10-9 (on the left there is the number of kills obtained by the human, while on the right the amount of Eve's), while the better one was 3-16. Table 5.4 shows the total number of kills and not the amount of won/lost matches.

Explorer	Episodes	Completed	Expired	AI1 Won	AI2 Won
AI vs Dummy	2411	2407	4	-	-
AI vs AI	4450	4450	0	2251	2199

Table 5.2: *Explorer* test results

Shooter	Episodes	Completed	Failed	Avg rounds employed
All dist.	42818	42818	0	2,96805
0-10 m	4778	4778	0	1,848052
10-20 m	7980	7980	0	2,138717
20-30 m	7988	7988	0	2,726966
30-40 m	7996	7996	0	3,302649
40-50 m	7914	7914	0	3,63912
50+ m	6162	6162	0	3,932819

Table 5.3: *Shooter* test results

Explorer + *Shooter (Eve)*	Episodes	AI's kills	Opponent's kills
AI vs AI	1532	1235	297
AI vs Human	475	330	145

Table 5.4: Eve test results

Conclusion

I presented *DefaultVR: the AI Expansion*, an innovative virtual reality game that includes realistic physics simulation and, in particular with this expansion, the possibility to play offline against Eve, an intelligent agent trained through reinforcement learning. Challenging a self-trained enemy creates the illusion that you are playing against another human player and not against an artificial being, concept that classic bots' algorithms are not able to satisfy completely.

Eve learned to explore a complex map with the goal of finding, as fast as possible, the enemy, and learned how to shoot with an automatic rifle, mastering aim and the recoil management without wasting bullets. The complex environment and the physics simulation required some more attentions during the development and the training, but the agent successfully fulfills these challenges, despite the limited hardware resources available for her training.

Eve reached the initial objective, at the cost of some adjustment with respect to the very original network idea, but still her results are pretty impressive. For now, she is able to play against a human without any fear even with some disadvantages, but her journey is not over: those handicaps needs to be overtaken in the near future.

Future planned works and ideas for this game consists in:

- Perform an *Explorer* network optimization through the use of a NAS (Neural Architecture Search) system, designing a complex model in terms of human comprehension but with a lower internal complexity (in terms of employed units) in order to make it slimmer and faster to query;
- Replace the *Ray Perception Sensor* with an high-res *Camera Sensor*, giving the agent an acceptable vertical field of view;

- Adding more observations to this model, such as the perception someone is shooting at it and the sound position awareness when a weapon fires;
- Adding the right hand management;
- Re-introduce the z axis in both the hands, letting the agent more physical actions available;
- The capacity to use semi-automatic guns;
- The decision of changing among a set of available weapons when required;
- Making the agent crouch to pick up something from the floor or just for hiding behind small objects;
- Team-based game modes, where agents must cooperate with other artificial guys and human players to reach the final goal, exploring new single-player and multiplayer gameplays;
- New maps and new weapons both the human player and the agent can learn and enjoy;
- A custom hardware, able to emulate the recoil of all the weapons included with this game, adding a new layer of immersion.

Playing against Eve is a challenging experience: she is good in what she learned, but, like human players, she makes some mistakes too, leaving to her opponents the impression that he/she is playing against another real player and not against an artificial being: this was indeed the goal of this project.

Bibliography

[1] L. Levita, "Gaming client-server in realtà virtuale basato su libreria openxr," Univeristy of Bologna, 2021.

[2] C. Berner, G. Brockman, B. Chan, *et al.*, "Dota 2 with large scale deep reinforcement learning," *CoRR*, vol. abs/1912.06680, 2019. arXiv: 1912.06680. [Online]. Available: http://arxiv.org/abs/1912.06680.

[3] K. Arulkumaran, A. Cully, and J. Togelius, "Alphastar: An evolutionary computation perspective," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, M. López-Ibáñez, A. Auger, and T. Stützle, Eds., ACM, 2019, pp. 314–315. DOI: 10.1145/3319619.3321894. [Online]. Available: https://doi.org/10.1145/3319619.3321894.

[4] "Meta connect," Meta. (2023), [Online]. Available: https://www.metaconnect.com/it-it.

[5] A. Rahimi, J. Zhou, and S. Haghani, "A VR gun controller with recoil adjustability," in *2020 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, January 4-6, 2020*, IEEE, 2020, pp. 1–2. DOI: 10.1109/ICCE46568.2020.9043008. [Online]. Available: https://doi.org/10.1109/ICCE46568.2020.9043008.

[6] "Hurricane vr - physics interaction toolkit," Cloudwalkin Games. (2023), [Online]. Available: https://assetstore.unity.com/packages/tools/physics/hurricane-vr-physics-interaction-toolkit-177300.

[7] "Vr physical hand," VR Physical Hand. (2020), [Online]. Available: https://www.unrealengine.com/marketplace/en-US/product/vr-physical-hand.

[8] Y. Ye, L. Liu, L. Hu, and S. Xia, "Neural3points: Learning to generate physically realistic full-body motion for virtual reality users," *Comput. Graph. Forum*, vol. 41, no. 8, pp. 183–194, 2022. DOI: 10.1111/cgf.14634. [Online]. Available: https://doi.org/10.1111/cgf.14634.

[9] R. Bensadoun, S. Gur, N. Blau, T. Shenkar, and L. Wolf, "Neural inverse kinematics," *CoRR*, vol. abs/2205.10837, 2022. DOI: 10.48550/arXiv.2205.10837. arXiv: 2205.10837. [Online]. Available: https://doi.org/10.48550/arXiv.2205.10837.

[10] A. Juliani, V. Berges, E. Vckay, *et al.*, "Unity: A general platform for intelligent agents," *CoRR*, vol. abs/1809.02627, 2018. arXiv: 1809.02627. [Online]. Available: http://arxiv.org/abs/1809.02627.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347,

2017. arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347.

[12] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, J. G. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 1856–1865. [Online]. Available: http://proceedings.mlr.press/v80/haarnoja18b.html.

[13] A. Cohen, E. Teng, V. Berges, *et al.*, "On the use and misuse of absorbing states in multi-agent reinforcement learning," *CoRR*, vol. abs/2111.05992, 2021. arXiv: 2111.05992. [Online]. Available: https://arxiv.org/abs/2111.05992.

[14] Č. Livada and D. Hodak, "Advanced mechanisms of perception in the digital hide and seek game based on deep learning," in *2022 International Conference on Smart Systems and Technologies (SST)*, 2022, pp. 135–140. DOI: 10.1109/SST55530.2022.9954814.

[15] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2017, Honolulu, HI, USA, July 21-26, 2017*, IEEE Computer Society, 2017, pp. 488–489. DOI: 10.1109/CVPRW.2017.70. [Online]. Available: https://doi.org/10.1109/CVPRW.2017.70.

[16] A. Aubret, L. Matignon, and S. Hassas, "A survey on intrinsic motivation in reinforcement learning," *CoRR*, vol. abs/1908.06976, 2019. arXiv: 1908.06976. [Online]. Available: http://arxiv.org/abs/1908.06976.

[17] J. Schmidhuber, "Formal theory of creativity, fun, and intrinsic motivation (1990-2010)," *IEEE Trans. Auton. Ment. Dev.*, vol. 2, no. 3, pp. 230–247, 2010. DOI: 10.1109/TAMD.2010.2056368. [Online]. Available: https://doi.org/10.1109/TAMD.2010.2056368.

[18] Y. Burda, H. Edwards, A. J. Storkey, and O. Klimov, "Exploration by random network distillation," *CoRR*, vol. abs/1810.12894, 2018. arXiv: 1810.12894. [Online]. Available: http://arxiv.org/abs/1810.12894.

[19] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, "How to train your robot with deep reinforcement learning: Lessons we have learned," *Int. J. Robotics Res.*, vol. 40, no. 4-5, 2021. DOI: 10.1177/0278364920987859. [Online]. Available: https://doi.org/10.1177/0278364920987859.

[20] L. Weng, *Exploration strategies in deep reinforcement learning*, 2020. [Online]. Available: https://lilianweng.github.io/posts/2020-06-07-exploration-drl/#the-noisy-tv-problem.

[21] P. Ladosz, L. Weng, M. Kim, and H. Oh, "Exploration in deep reinforcement learning: A survey," *Inf. Fusion*, vol. 85, pp. 1–22, 2022.

DOI: 10.1016/j.inffus.2022.03.003. [Online]. Available: https://doi.org/10.1016/j.inffus.2022.03.003.

[22] Y. Aytar, T. Pfaff, D. Budden, T. L. Paine, Z. Wang, and N. de Freitas, "Playing hard exploration games by watching youtube," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 2935–2945. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/35309226eb45ec366ca86a4329a2b7c3-Abstract.html.

[23] T. Salimans and R. Chen, "Learning montezuma's revenge from a single demonstration," *CoRR*, vol. abs/1812.03381, 2018. arXiv: 1812.03381. [Online]. Available: http://arxiv.org/abs/1812.03381.

[24] Z. Hong, T. Shann, S. Su, Y. Chang, and C. Lee, "Diversity-driven exploration strategy for deep reinforcement learning," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*, OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=BJsD7L1vz.

[25] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine, "Diversity is all you need: Learning skills without a reward function," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=SJx63jRqFm.

[26] S. Forestier, Y. Mollard, and P. Oudeyer, "Intrinsically motivated goal exploration processes with automatic curriculum learning," *CoRR*, vol. abs/1708.02190, 2017. arXiv: 1708.02190. [Online]. Available: http://arxiv.org/abs/1708.02190.

[27] P. Soviany, R. T. Ionescu, P. Rota, and N. Sebe, "Curriculum learning: A survey," *Int. J. Comput. Vis.*, vol. 130, no. 6, pp. 1526–1565, 2022. DOI: 10.1007/s11263-022-01611-x. [Online]. Available: https://doi.org/10.1007/s11263-022-01611-x.

[28] X. Wang, Y. Chen, and W. Zhu, "A survey on curriculum learning," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 9, pp. 4555–4576, 2022. DOI: 10.1109/TPAMI.2021.3069908. [Online]. Available: https://doi.org/10.1109/TPAMI.2021.3069908.

[29] K. Fang, Y. Zhu, S. Savarese, and L. Fei-Fei, "Adaptive procedural task generation for hard-exploration problems," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=8xLkv08d70T.

[30] C. Colas, P. Oudeyer, O. Sigaud, P. Fournier, and M. Chetouani, "CURIOUS: intrinsically motivated modular multi-goal reinforcement learning," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, Califor-*

nia, USA, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, PMLR, 2019, pp. 1331–1340. [Online]. Available: http://proceedings.mlr.press/v97/colas19a.html.

[31] O. Kelany, S. Aly, and M. A. Ismail, "Deep learning model for financial time series prediction," in *14th International Conference on Innovations in Information Technology, IIT 2020, Al Ain, United Arab Emirates, November 17-18, 2020*, IEEE, 2020, pp. 120–125. DOI: 10.1109/IIT50501.2020.9299063. [Online]. Available: https://doi.org/10.1109/IIT50501.2020.9299063.

[32] N. Pai and V. Ilango, "Lstm neural network model with feature selection for financial time series prediction," in *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 2020, pp. 672–677. DOI: 10.1109/I-SMAC49090.2020.9243376.

[33] S. Reddy Beeram and S. Kuchibhotla, "A survey on state-of-the-art financial time series prediction models," in *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*, 2021, pp. 596–604. DOI: 10.1109/ICCMC51019.2021.9418313.

[34] M. Lombardi, *Rul-based maintenance policies*, 2023. [Online]. Available: https://github.com/a3i-2022-2023.

[35] H. Face, *Self-play: A classic technique to train competitive agents in adversarial games*, 2021. [Online]. Available: https://huggingface.co/learn/deep-rl-course/unit7/self-play?fw=pt.

[36] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch, "Emergent complexity via multi-agent competition," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=Sy0GnUxCb.

[37] OpenAI, M. Plappert, R. Sampedro, *et al.*, "Asymmetric self-play for automatic goal discovery in robotic manipulation," *CoRR*, vol. abs/2101.04882, 2021. arXiv: 2101.04882. [Online]. Available: https://arxiv.org/abs/2101.04882.

[38] MATLAB, *What is Reinforcement Learning?* The MathWorks Inc., 2010. [Online]. Available: https://mathworks.com/discovery/reinforcement-learning.html.

[39] M. Swiechowski, K. Godlewski, B. Sawicki, and J. Mandziuk, "Monte carlo tree search: A review of recent modifications and applications," *Artif. Intell. Rev.*, vol. 56, no. 3, pp. 2497–2562, 2023. DOI: 10.1007/s10462-022-10228-y. [Online]. Available: https://doi.org/10.1007/s10462-022-10228-y.

[40] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-learning algorithms: A comprehensive classification and applications," *IEEE Access*, vol. 7, pp. 133 653–133 667, 2019. DOI: 10.1109/ACCESS.2019.2941229.

[41] T. Simonini, *Offline vs. online reinforcement learning*. [Online]. Available: https://huggingface.co/learn/deep-rl-course/unitbonus3/offline-online.

[42] S. Levine, A. Kumar, G. Tucker, and J. Fu, "Offline reinforcement learning: Tutorial, review, and perspectives on open problems," *CoRR*, vol. abs/2005.01643, 2020. arXiv: 2005.01643. [Online]. Available: https://arxiv.org/abs/2005.01643.

[43] J. Ho and S. Ermon, "Generative adversarial imitation learning," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, Eds., 2016, pp. 4565–4573. [Online]. Available: https://proceedings.neurips.cc/paper/2016/hash/cc7e2b878868cbae992d1fb743995d8f-Abstract.html.

[44] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial networks," *CoRR*, vol. abs/1406.2661, 2014. arXiv: 1406.2661. [Online]. Available: http://arxiv.org/abs/1406.2661.

[45] U. Technologies, *Unity user manual*, 2023. [Online]. Available: https://docs.unity3d.com/Manual/index.html.

[46] T. K. O. W. Group, *The openxr™ specification*, version Version 1.0.27, 2023. [Online]. Available: https://registry.khronos.org/OpenXR/specs/1.0/html/xrspec.html.

[47] G. Brockman, V. Cheung, L. Pettersson, *et al.*, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016. arXiv: 1606.01540. [Online]. Available: http://arxiv.org/abs/1606.01540.

[48] J. Terry, B. Black, N. Grammel, *et al.*, "Pettingzoo: Gym for multi-agent reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 15 032–15 043, 2021.

[49] V. Corporation, *Valve index headset*, 2019. [Online]. Available: https://www.valvesoftware.com/en/index/headset.

[50] D. Welsh, M. A. A. H. Khan, and N. Roy, *Gunshot recoil dataset*, https://doi.org/10.21227/2k8c-tm65, Oct. 2021. DOI: 10.21227/2k8c-tm65. [Online]. Available: https://doi.org/10.21227/2k8c-tm65.

[51] U. T. Rob Srinivasiah, *How to maximize ar and vr performance with advanced stereo rendering*, 2017. [Online]. Available: https://blog.unity.com/technology/how-to-maximize-ar-and-vr-performance-with-advanced-stereo-rendering.

[52] U. Technologies, *Optimizing your vr/ar experiences*, 2020. [Online]. Available: https://learn.unity.com/tutorial/optimizing-your-vr-ar-experiences.

[53] U. Technologies, *Optimizing graphics in unity*, 2018. [Online]. Available: https://learn.unity.com/tutorial/optimizing-graphics-in-unity.