الشعبية الديمقراطية الجزائرية الجمهورية
République Algérienne démocratique etPopulaire

وزارة التعليم العالي والبحث العلمي
Ministère de l'enseignement supérieur et de la recherche Scientifique

جامعة سعد دحلب البليدة
Université SAAD DAHLAB de BLIDA

كلية التكنولوجيا
Faculté de Technologie

قسم الإلكترونيك
Département d'Électronique

Mémoire de Master

Filière d'Électronique

Spécialité Electronique de Système Embarqué

présenté par

DELLALI Mohamed Fethi

&

BOUZEGZEG Mohamed El Mahdi

Autonomous Parking Simulation using Unity Game Engine and Reinforcement Learning

Proposé par :Mme Boughrira Hamida

Année Universitaire 2021-2022

Acknowledgments

We would first want to express our thankfulness to Allah (Exalted is He above all) for enabling us in completing this project.

We also would like to express our gratitude and deep respect to
Mrs. BOUGHERIRA Hamida for her unrelenting support, encouragements and patience through the many years and especially this year, for all the guidance you have given us and the knowledge you have helped us acquire, we will never be able to repay you. Thank you.

Our extreme gratitude also goes to our head of specialty Mrs. NACEUR Djamila for her kindness, help and encouragement.

Additionally, we would like to express our gratitude to the jury members for their willingness to review our work and add to it with their suggestions in light of their interest in it.

Finally, we want to thank all of our professors from the bottom of our hearts for the knowledge they gave us and the infinite support and guidance through our years in university.

Dedication

To my Mother, and my Father

To my two Sisters, and my nephew **IYAD**

Mohamed El Mahdi

Dedication

I am dedicating this Thesis to four beloved people who have meant and continue to mean so much to me.

To my father, mother, aunt and brother.

Fethi

الملخص:

تم اتباع استراتيجية مدروسة جيدا من أجل تحقيق الهدف المعلن للمشروع، وهو استخدام التعلم المعزز ومحرك لعبة Unity لإنشاء محاكاة الوقوف الآلي للسيارات. بدأت هذه الاستراتيجية بمناقشة فروع مختلفة من الذكاء الاصطناعي وأساليبها، تليها مناقشة مفصلة للتعلم المعزز، ومحرك لعبة Unity، وML-Agents. وأخيرا، أنشأنا محاكاة باستخدام محرك لعبة Unity التي شملت موقف للسيارات ”كبيئة“ وسيارة ”كعميل“، ثم استخدمنا التعلم المعزز وML-Agents لتدريب السيارة على الوقوف بشكل آلي.

Résume :

Une stratégie bien pensée a été suivie afin d'atteindre l'objectif déclaré du projet, qui est d'utiliser l'apprentissage par renforcement et le moteur de jeu Unity pour créer une simulation de stationnement autonome. Cette stratégie a commencé par une discussion sur les différents sous-ensembles de l'intelligence artificielle (IA) et leurs méthodes, suivie d'une discussion détaillée sur l'apprentissage par renforcement, le moteur de jeu Unity et ML-Agents. Enfin, nous avons créé une simulation à l'aide du moteur de jeu Unity, avec un parking comme “environnement” et une voiture comme “agent”, puis nous avons utilisé l'apprentissage par renforcement et ML-Agents pour entraîner la voiture à se garer de manière autonome.

Abstract:

A well-thought-out strategy was followed in order to achieve the project's stated objective, which is to use Reinforcement learning and Unity game engine to create an autonomous parking simulation. This strategy started with a discussion of various artificial intelligence (AI) subsets and their methods, followed by a detailed discussion of reinforcement learning, Unity game engine, and ML-Agents. Finally, we created a simulation using Unity game engine that included a parking lot as “the environment” and a car as “the agent”, and then we used reinforcement learning and ML-Agents to train the car to park autonomously.

Table of Contents

List of Figures

Abbreviations List

AI	Artificial Intelligence
ANN	Artificial Neural Network
DL	Deep Learning
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
ML	Machine Learning
ML-Agent	Machine Learning Agent
MDP	Markov Decision Process
PPO	Proximal Policy Optimization Algorithm
RL	Reinforcement Learning
TPU	Tensor Processing Unit
VS code	Visual Studio code
2D	2 Dimensional
3D	3 Dimensional

General Introduction

For decades, car ownership levels have been gradually rising, with some individuals even purchasing numerous automobiles, pushing the vehicle/person ratio to figures that are currently higher than ever. Even though this increase in car ownership is considered as a good indicator of a healthy social status, it causes some unexpected problems on society. One of these problems is parking and especially in high density areas. Parking problems in cities and metropolitan regions are getting more noticeable, and have been one of the most often discussed issues in the recent years. Parking availability is always reduced as population density rises and more cars are used. As a result, drivers must deal with long queues and smaller spaces while accessing parking lots causing them to waste a lot of energy and time on a simple task.

To help solve these issues autonomous driving and parking are now the subject of a lot of research in both the automobile industry and academic and though a lot of advancements have been made in these fields most autonomous parking models that are available in the market still require the drivers to be alert and take control whenever necessary. But with each advancement made, driver requirements become more and more minimal.

The goal of this project is to prove that reinforcement learning (RL) is a good way to solve this issue due to its ability to train a model to mimic a complex activity by simulating that situation and rewarding and punishing the model for its actions until it exhibits the desired behavior.

In order to achieve this, we use Unity video game engine to create a simulation of a parking environment and utilize Unity's ML-Agents toolkit to train a car in this environment to learn how to park using the Proximal Policy Optimization (PPO) algorithm.

In the first chapter of this thesis, we presented the various types of artificial intelligence (AI) and discussed in detail Reinforcement learning (RL).

Additionally, we explained how Unity video game engine functions and how ML-Agents is combined with it to develop intelligent models.

In the second chapter, we went into depth about how we created a reinforcement learning environment, an agent, and with the help of Google Colab trained the agent to learn how to park in a full parking lot.

Finally, in the third chapter, the results of the training process were discussed and we tested the trained model.

Chapter I: Generalities on Artificial intelligence (AI) and Unity.

I.1 Introduction

In this first chapter, we briefly discuss the various types of artificial intelligence (AI) and discuss reinforcement learning (RL) detail.

Later, since we will use it to simulate the parking environment, we examine the functioning of the Unity video game engine and how to apply ML-Agents to it.

I.2 Artificial Intelligence (AI)

Artificial Intelligence (AI) is defined as a specific field in computer science which focuses on the ability of machines to imitate human intelligence and thinking by analyzing their environment and taking action -with some degree of autonomy- to achieve specific goals. [1]

John McCarthy originated the phrase Artificial Intelligence (AI) in 1956, when he hosted the first academic conference on the subject. However, the quest to determine if machines can actually think began far earlier. Alan Turing proposed the renowned "Turing Test" in 1950 in a study on the idea of machines being able to simulate human beings and the ability to do intelligent things like play Chess in his famous paper "Computing Machinery and Intelligence".

AI popularity has been rapidly increasing especially after the covid pandemic where different businesses had to shut-down due to their dependency on human workers and ideas on how to replace essential human workers with AI powered robots have been further explored.

Artificial intelligence (AI) popularity was also heavily impacted by the development of better computer hardware and the explosion in availability of data in many shapes and from all over the globe. These data are taken from different search engines, social media, e-commerce platforms, and other digital sources.
There are two subsets of AI that are very dependent on data: Machine Learning (ML) and Deep Learning (DL). (**Figure I.1**)

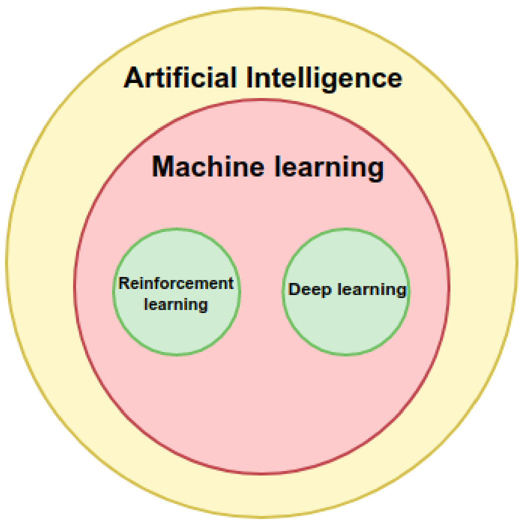


Figure I.1: Artificial Intelligence (AI) subsets

I.3 Machine Learning (ML)

Machine Learning (ML) is a subset of artificial intelligence (AI) focusing on a specific goal: setting computers up to be able to perform tasks without the need for explicit programming.

Computers are fed structured data and 'learn' to become better at evaluating and acting on that data over time.

The learning process starts with observations or data, such as examples, direct experience, or instruction, so as to look for patterns in data and make enhanced decisions in the future based on the examples that we give. The main aim is to allow the systems to learn by itself without human intervention or support and adjust the actions accordingly. [4]

Depending on the type of problem we are trying to solve and according to the type of input data and the output we desire, Machine learning (ML) is mainly divided into 3 types (**Figure I.2)**:

- Supervised Learning.
- Unsupervised Learning.
- Reinforcement Learning (RL).

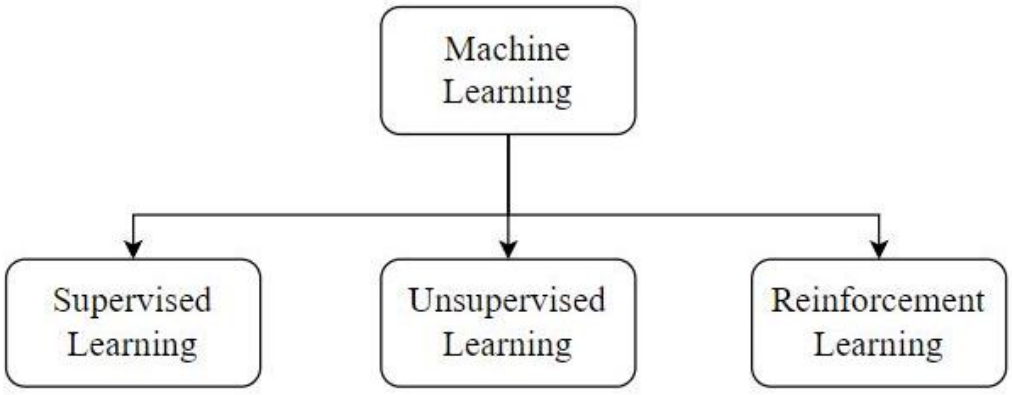


Figure I.2: Types of machine learning [18]

Although machine learning (ML) only started to flourish in the 1990s, it has quickly become the most popular and most successful subfield of AI, a trend driven by the availability of faster hardware and larger datasets. [3]

ML algorithms

Choosing the right algorithm or combination of algorithms can be a challenge for anyone working in the field of machine learning (ML) it is essential to understand the three main categories of machine learning (ML), shown in **Figure I.3** especially reinforcement learning (RL) which we will explore more later in this chapter.

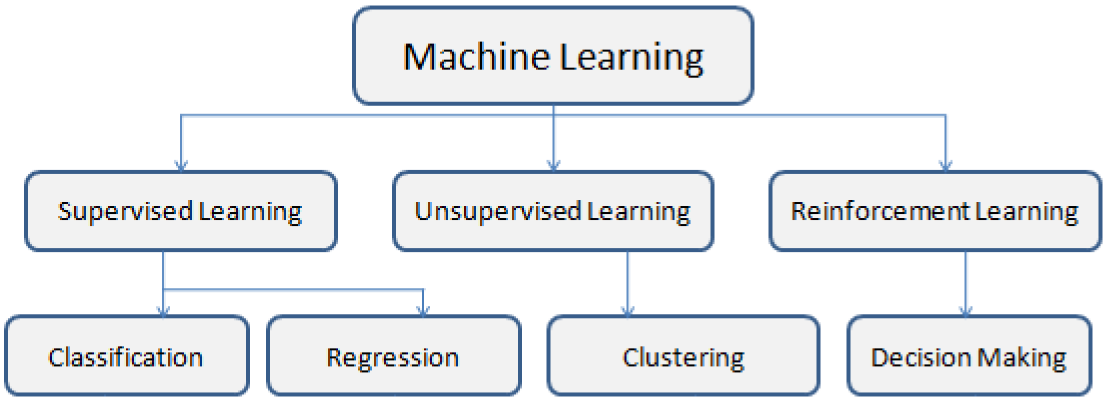


Figure I.3: Machine learning (ML) algorithms

Supervised Learning: Supervised learning requires labelled input and output data during the training phase. This training data is usually organized by a human in the preparation phase, before being used to train and test the model.it can be used to classify new and unknown datasets and predict outcomes once it has learned the relationship between the input and output data. Because at least some of this approach involves human control, it's called supervised learning. The largest portion of data is unlabeled and unprocessed. In most cases, human participation is required to appropriately classify data that is ready for supervised learning.

Some of the tasks Supervised learning is used for:

- **Regression**: predicting outcomes from continuously changing data
- **Classification**: identifying input data as part of a learned group.

Unsupervised Learning: Unsupervised learning is the training of models on raw and unlabeled training data. It is frequently used to find patterns and trends in raw datasets, as well as to group similar data into a collection of groups. It's also a common method for gaining a better understanding of datasets.

When compared to supervised learning, unsupervised learning takes a more hands-off approach. Although a human will configure model hyperparameters like the number of cluster points, the model will process large amounts of data efficiently and without human intervention.

Some of the tasks Unsupervised learning is used for:

- **Clustering**: This task involves grouping data into clusters that are similar within each cluster but different from one another.
- **Dimensionality Reduction:** The goal of this task is to reduce the data's complexity/size by lowering their dimensions while maintaining the most important ones.

Reinforcement Learning (RL): In reinforcement learning we teach certain agents to solve specific problems by interacting with their surrounding environment. From this interaction the agent collects observations, these observations are then used to determine a specific action to be taken.

Reinforcement learning (RL) is discussed in detail in section I.6.

I.4.Artificial Neural Network (ANN)

Because of its incredible ability to perceive the outside world and learn, the human brain is the most complex organ in the human body. The neuron is the basic building block of the brain.

A Neuron can be divided into three main parts: dendrites, cell body (also known as “soma”), and axon. [4]

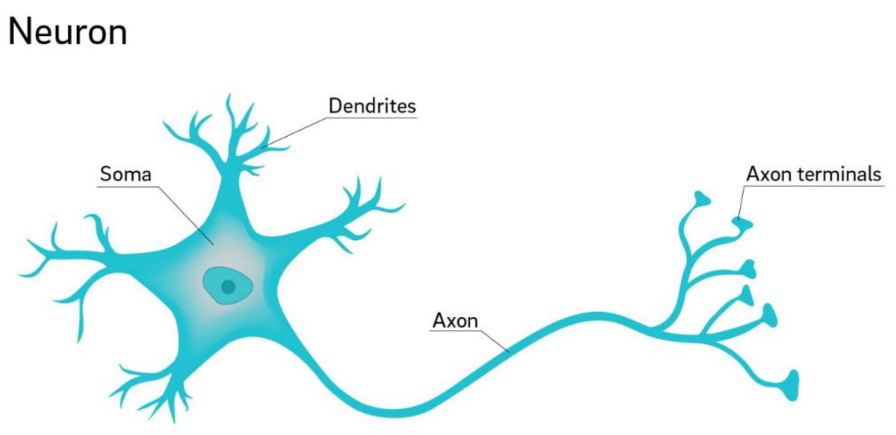


Figure I.4: Biological Neuron [17]

- Dendrites: group of extensions that receive information other neurons
- Cell body (also known as “soma”): the part responsible for producing the activation by processing all the information that comes from the dendrites.
- Axon: Its goal is to use synaptic terminals to send stimulation to other neurons.

Neurons communicate via electrical signals that are short-lived impulses or "spikes" in the voltage of the cell wall or membrane. The interneuron connections are mediated by electrochemical junctions called synapses, which are located on branches of the cell referred to as dendrites. [6]

I.4.1 Artificial Neuron

The computational components or processing units, called artificial neurons, are simplified models of biological neurons. [4]

Artificial neural networks (ANNs) are a connection of artificial neurons that simulate the process of biological learning.

The artificial neural network (ANN) structures were developed from known models of biological nervous systems and the human brain itself [4]

The artificial neurons in ANNs are implemented as shown in **Figure I.5**

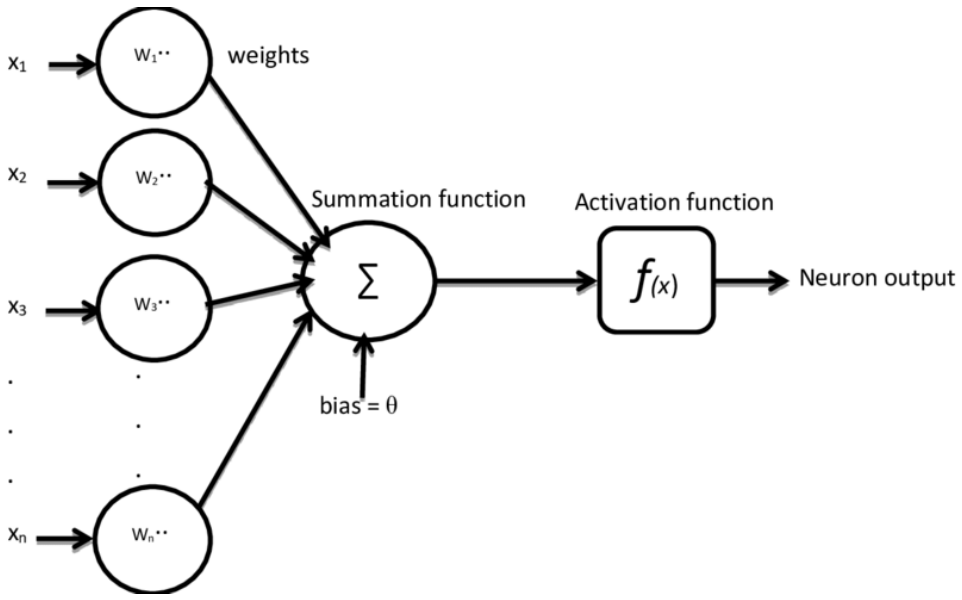


Figure I.5: Artificial neuron in an ANN [16]

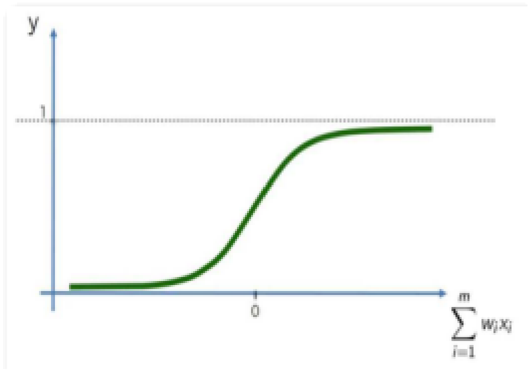
- Input connections (inputs): Signals (X1 through Xn) are multiplied by their weights (W1 through Wn) accordingly
- Summation function (∑): sums inputs after being multiplied by their own associated weight, and adds the bias.
- Activation function: the activation function decides, whether a neuron should be activated or not according to the summation function. The purpose of this function is to introduce non-linearity into the output of the neuron.

I.4.2 Activation Functions

Are nonlinear mathematical functions that calculate the level of neuron activation. The function is chosen according to the problem and outputs; we list the most popular activation function:

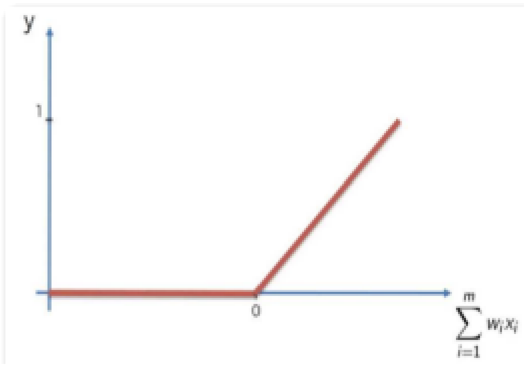
• **Sigmoid**: Normalizes the output of each neuron, its output value is in the range [0,1].

$$F(x) = \frac{1}{1 + e^{-x}} \text{ …… Eq I.1}$$



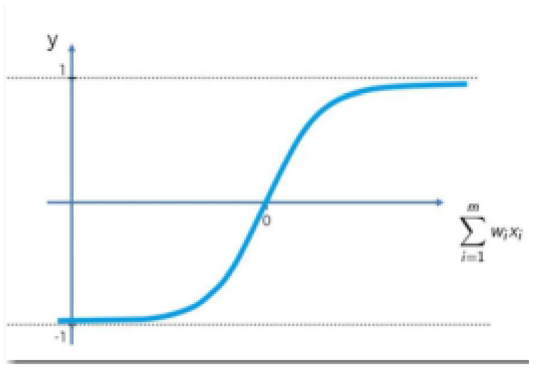
• **Rectified Linear Unit (ReLU):** It's gives 0 if it receives a negative input, and returns the same positive value x otherwise.

$$F(x) = \max(x, 0) \text{ …… Eq I.2}$$



• **Hyperbolic Tangent (TanH):** Normalizes the output of each neuron, its output value is in the range [-1,1].

$$F(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \text{…… Eq I.3}$$



• **Softmax:** Used when outputs are multi class (multi-classification) and its output value is in the range [0,1]

$$F(xi) = \frac{e^{xi}}{\sum_{j=0}^{n} e^{xj}}, \text{ i= 1,2,3,……,n …… Eq I.4}$$

I.5 Deep Learning (DL)

Deep learning (DL) can be considered as a subset of machine learning. It is a field that is focused on computer algorithms learning and developing on their own. Deep learning (DL) uses artificial neural networks (ANNs), which are supposed to mimic how humans think and learn, as opposed to machine learning, which uses simpler principles. Deep learning (DL) is especially useful when you're trying to learn patterns from unstructured data. [2]

Deep learning (DL) algorithms can be regarded both as a sophisticated and mathematically complex evolution of machine learning (ML) algorithms.

Deep learning (DL) is one of the most important branches of machine learning (ML) and its most important features are:

- Depth: so that the term 'deep' refers to the number of hidden layers that make up the deep neural network and this is what distinguishes it from normal neural networks.
- Special ability to extract features: by processing a large amount of data without the need for external intervention, unlike traditional machine learning (ML) algorithms.

DL has witnessed great prosperity in recent years due to a combination of factors:

- Big data: it is one of the most important things that deep learning (DL) needs. Where digitization led to the presence of a large amount of data, which has become easy to access and use.
- Graphics Processing Unit (GPU) and cloud computing: Deep Learning (DL) relies on processing a large set of data, which requires computers with high capabilities. Therefore, the increase in computing power encouraged the use of deep learning (DL).

I.6 Reinforcement Learning (RL)

Reinforcement Learning (RL) is a particular area of Machine Learning (ML) where an agent interacts with an environment through executing a singular or a sequence of actions with the goal to solve a certain problem, the agent is not told which actions to take, but instead must discover which actions yield the most reward by trying them [9]. Following these actions, the agent receives a certain number of rewards and observations that play a part in defining the next actions that should be executed.

The sequences of rewards and observations that the agent gets define the behavior of the agent, which is also known as the policy This may lead the behavior through positive and negative reinforcement, reward and punishment respectively.

In our case for example the environment can be the parking lot and the agent can be the car and the actions can be represented as acceleration/reversing and steering, while the observations can be the location and speed of the car and the parking spot we want to reach.

Basic terminology to know in reinforcement learning:

- **Environment**: the surroundings in which the agent learns, the agent cannot change nor control the environment, it only has control on its own actions.
- **Agent**: the entity we train by interacting with the environment to learn a certain behavior.
- **Action**: The movement the agent chooses to execute at a certain timestep.
- **Reward**: The feedback the agent receives from the environment to assess how well the actions taken were.
- **State**: The state is returned by the environment describing the current situation of the agent.
- **Policy**: Is the strategy or the “thinking process” that the agent went through to choose an action.

I.6.1 Markov Decision Process (MDP)

MDPs are a mathematically idealized form of the reinforcement learning (RL) problem for which precise theoretical statements can be made. [9] MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision maker are called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. [9]

The environment and the agent interact across time in sequential ways. At each timestep, the agent receives a representation of the state of the environment and bases its next move on that.

The agent is finally rewarded for its prior actions and the environment changes to a new state.

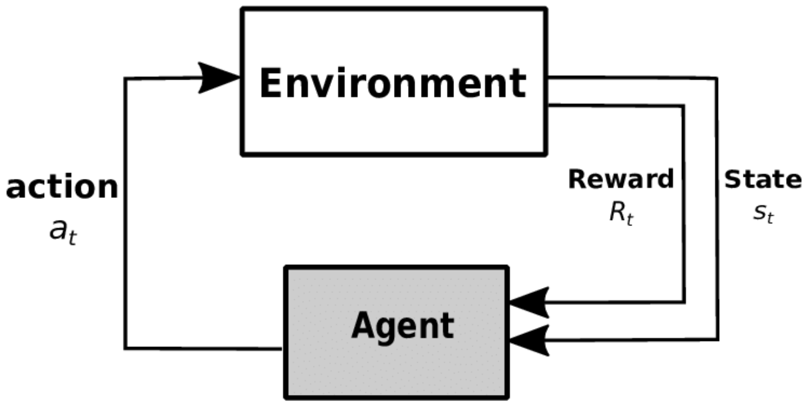


Figure I.6: Reinforcement learning (RL) process [9]

We define MDP components as:

- **The Environment.**
- **The Agent.**
- **States (S):** Every possible state of the environment.
- **Actions(A):** All possible actions the agent can take in the environment.
- **Rewards(R):** All possible rewards the agent can get from taking certain actions in the environment.

The process of selecting an action from one state, moving into some other state, and receiving a reward happens in sequence repeatedly. The goal of the agent is to maximize the number of Rewards it receives from taking actions in certain states of the environment.

I.6.1.1 Policies and Value Functions

Value functions are state or state-action pair functions that calculate the value of an agent's current state or the value of carrying out a certain action in a specific state using the expected rewards from future states.

The actions the agent will take result in these expected rewards, so value functions are defined in relation to specific acting patterns, also called as policies.

A policy is responsible of controlling the agent’s behavior, it is a mapping of certain states of the environment to available actions that can be taken in these states.

For an agent's policy π at time t the probability that the agent's action will be A(t) = a when in a state S(t) = s is π (a| s).

For policy π the value function of a state s is the expected result of being in that state and then following that policy:

$$\mathbf{v_{\pi}(s) = E_{\pi}[G(t) \mid S(t) = s] = E_{\pi}\left[\sum_{K=0}^{\infty} \gamma^{k} R(t+k+1) \mid S(t) = s\right], \text{for all } s \in S} \quad \text{....Eq I.5}$$

$\mathbf{E}_{\pi}$ Represents the expected random variable value for an agent following policy π.

$\boldsymbol{\gamma}$ Represents the discount factor [0,1], this factor reflects how much does the agent value distant future rewards compared to immediate future results.

We also express the value of taking action a in a certain state s following a policy π as the expected return beginning from state s, executing the action a and then continuing to follow policy π:

$$\mathbf{q_{\pi}(s,a) = E_{\pi}[G(t) \mid S(t) = s, A(t) = a] = E_{\pi}\left[\sum_{K=0}^{\infty} \gamma^{k} R(t+k+1) \mid S(t) = s, A(t) = a\right]} \quad \text{...Eq I.6}$$

I.6.1.2 Optimal Policies and Optimal Value Functions

Creating a policy that produces a lot of rewards is the main objective of reinforcement learning (RL). In an MDP, we say that a policy π is superior to another policy π' if its expected return v_{π} (s) is higher for each state s in S. The best policy is referred to as an optimal policy π*, and its state-value function is referred to as the optimal state-value function v*.

$$\mathbf{v_{*}(s) = \max_{\pi} v_{\pi}(s) \ \text{ for all } s \in S} \quad \text{......Eq I.7}$$

I.7 Unity

Video game engines are defined as software development tools mainly used to create different types of video games.

Unity is considered one of the most used video game engines in the field of making games and some of the main reasons for that are:

- Its support both 2D and 3D.
- Being a cross-platform game engine.
- The easy-to-use UI.
- Large community that finds solutions, share experiences and exchange ideas.

Unity was released back in 2005 with the aim of being only a video game engine; it is now used to achieve a plethora of products varying from games to animations and even used in engineering.

Figure I.7: Unity Logo

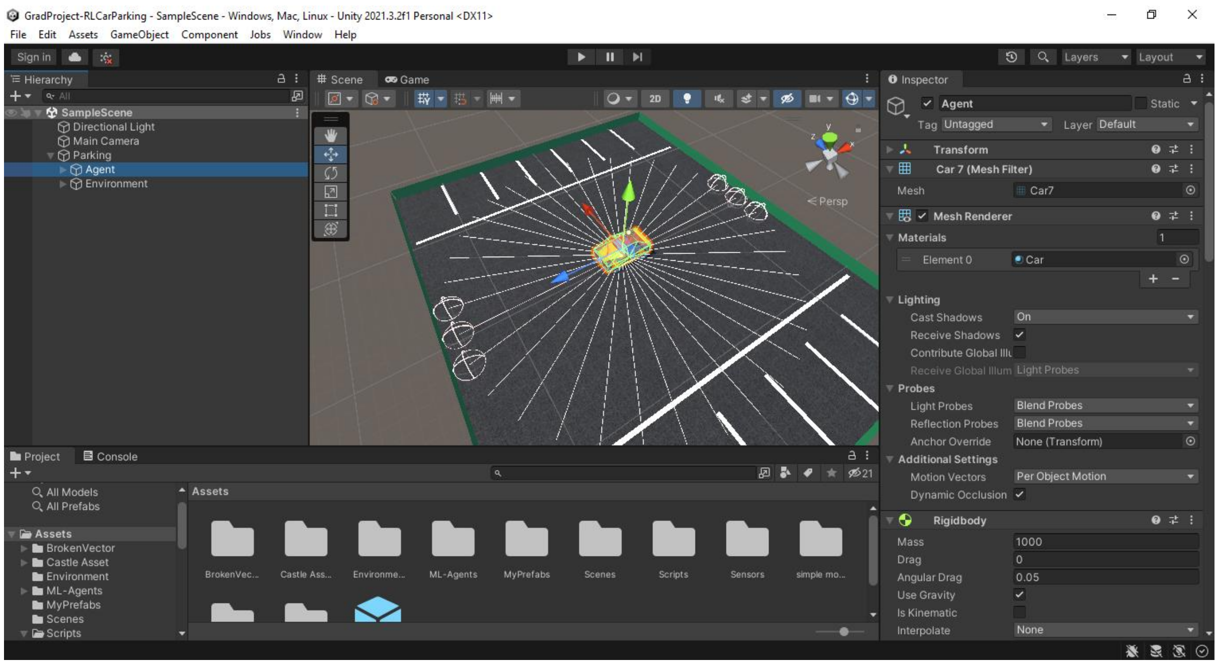
It is a free product that can be used by anyone with a rich and easy to understand documentation for every single expression, mechanic and system that the engine provides.

Unity is still being developed and updated so that it can deliver the best tools possible to the user.

I.7.1 Unity Editor

The main unity editor shown in **Figure I.8** consists of several modules, windows and tools that can be rearranged by the user as he desires.

The default interface layout can be modified quickly and easily using the toolbar by choosing the ideal windows arrangement under Window > Layouts.



Agent ✓ Static
Tag Untagged Layer Default
Transform
Car 7 (Mesh Filter)
Mesh Car7
Mesh Renderer ✓
Materials 1
Element 0 Car
Lighting
Cast Shadows On
Receive Shadows ✓
Contribute Global Illu
Receive Global Illum Light Probes
Probes
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)
Additional Settings
Motion Vectors Per Object Motion
Dynamic Occlusion ✓
Rigidbody
Mass 1000
Drag 0
Angular Drag 0.05
Use Gravity ✓
Is Kinematic
Interpolate None

Figure I.8: Unity Editor

Users can also easily transfer an object from one window to another using the drag-and-drop function.

I.7.1.1 Core elements

Every unity project is built on a couple of core elements, these elements need to be explained in order to better understand how unity works.

- **GameObjects:** Any object found in a project scene is referred to as a GameObject, they are considered the most crucial part of Unity projects.
  These GameObjects can be anything and usually hold multiple components in their properties.

- **Components:** Components are additions to a GameObject's properties that control how it behaves. They can be changed, added, or removed out as needed.
  A wide range of components for various game object types are included with Unity.
  For example: a transform component is included with every GameObject, allowing the user to move, rotate, and resize the GameObject.

- **Scenes:** A scene is a 2D or 3D environment in which all of the GameObjects we use to create a game are placed. Depending on the concept, a Unity game might have a single scene or multiple scenes.
- **Assets:** Any object used in a Unity project is referred to as an asset. Assets can be created inside of Unity using several GameObjects or imported from another software.
  The Unity Asset Store is a collection of both free and paid-for items made by Unity Technologies or by the community and it includes a wide variety of assets including audio, images, 2D/3D models, textures, and more.
  Assets make game production much easier, allowing you to concentrate on other aspects of the project.

- **Prefab:** Prefabs represent GameObjects that can be used again in scenes or projects.
  When a prefab is used in a scene, the original GameObject is preserved while an instance of the GameObject is created and added to the scene.
  Changing a prefab's properties will modify every instance, making it simpler to use the same GameObject across many scenes or projects.

- **Scripts:** Unity enables us to write scripts that we attach to a GameObject as a custom component when the GameObject's basic components are unable to achieve a specific behavior. Unity supports C# as a programming language.

Typically, multiple scenes make up a Unity game. These scenes all contain a variety of GameObjects, ranging from simple geometric shapes to complex objects.

These GameObjects contain components that tells them how to behave in the scene and how to interact with other GameObjects.

The general architecture of games made in Unity is shown in **Figure I.9**.

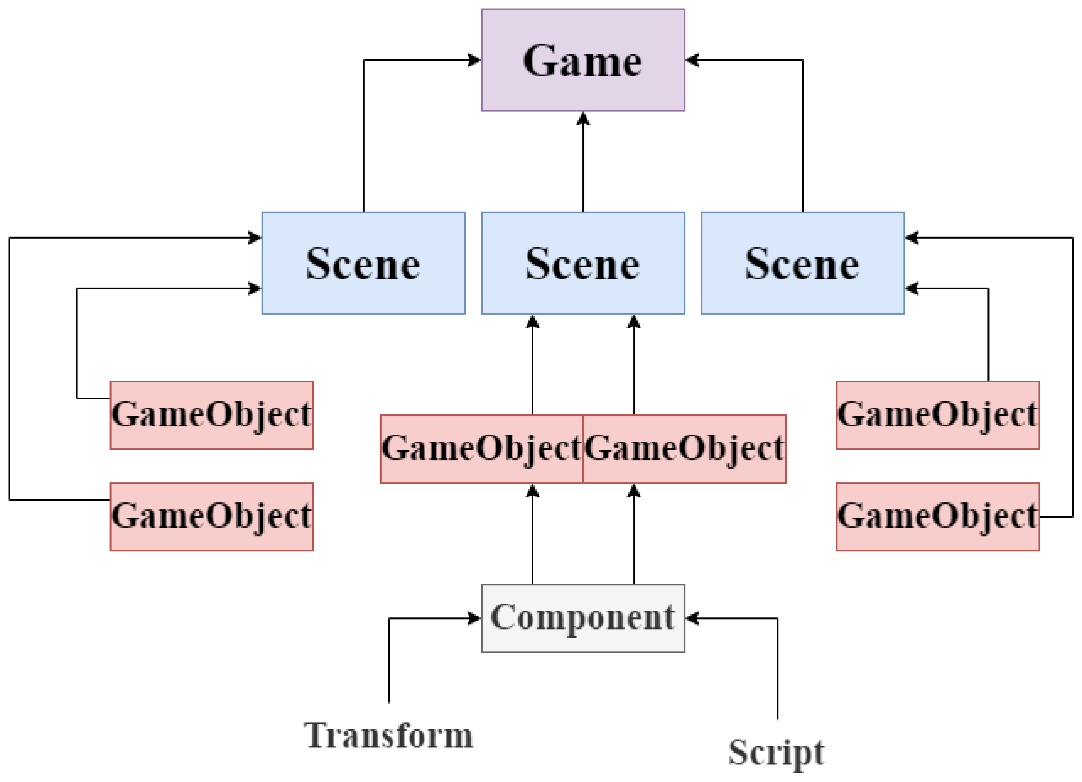


Figure I.9: Unity game architecture

I.7.1.2 Rigidbody and colliders

There are two main components that allow us to simulate real life physics on our GameObjects:

- **Rigidbody:** Rigidbody must be included for our GameObjects to behave realistically and react realistically in the scene. It is the main component that enables physical behavior for a **GameObject** with a Rigidbody attached, the object will immediately respond to gravity. [11]

- **Colliders:** Colliders represent the physical boundaries of a GameObject for a collision.
  Due to the variety of types of GameObjects Unity provides multiple types of collider components.
  Simple shape colliders for example include:
  - Box Collider
  - Sphere Collider
  - Capsule Collider

  However, for complex 3D GameObjects these simple shape colliders are not efficient, in 3D, you can use Mesh Colliders to match the shape of the GameObject's mesh exactly. [12]

I.7.1.3 Unity Editor Interface

Unity interface is made up of four main windows:

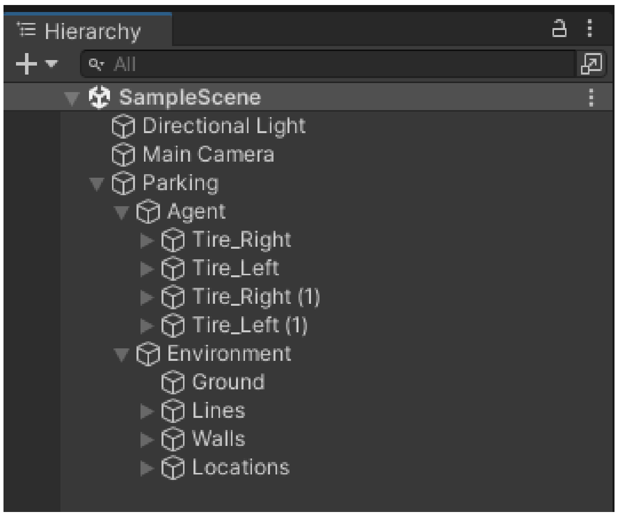
- ➢ Hierarchy window
- ➢ Scene and Game view window
- ➢ Inspector window
- ➢ Project window

Hierarchy window

In the hierarchy window (shown in **Figure I.10**), each GameObject that is present in the scene is listed.

We can use it to group GameObject anyway we like and to add new ones as well.

The GameObject that contains other GameObjects is known as the parent GameObject in this window, and the GameObject it contains is known as the child GameObject.



```
Hierarchy
All
SampleScene
  Directional Light
  Main Camera
  Parking
    Agent
      Tire_Right
      Tire_Left
      Tire_Right (1)
      Tire_Left (1)
    Environment
      Ground
      Lines
      Walls
      Locations
```

Figure I.10: Hierarchy window

Scene and Game window

Every GameObject utilized in the game can be viewed in the scene window, it allows us to navigate through the scene and modify it by moving, resizing, deleting …etc every GameObject alone or multiple at once.

The game window displays the final result of the game from the perspective of the player.

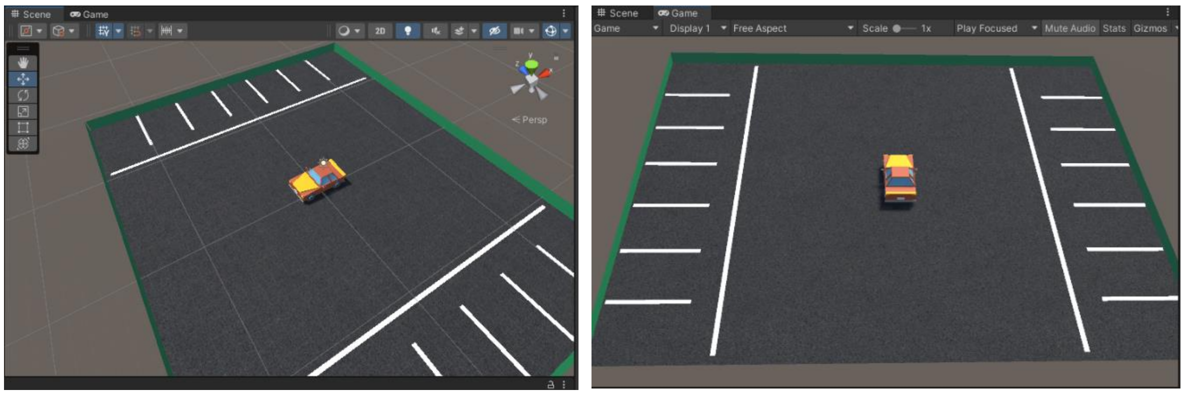
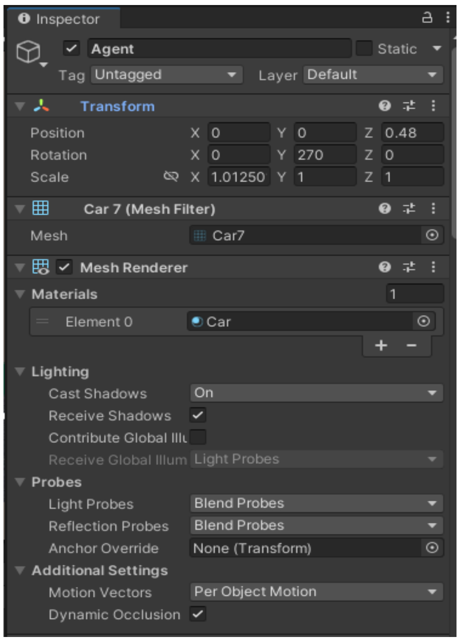


Figure I.11: Scene and game window

Inspector window

The inspector window lets us change, remove, or add components to a GameObject's properties after choosing it from the hierarchy window. **Figure I.12**

Unity comes with a variety of components for various GameObject types, and we can even add our own custom components by utilizing scripts.



Agent
Tag Untagged
Layer Default

Transform
Position X 0 Y 0 Z 0.48
Rotation X 0 Y 270 Z 0
Scale X 1.01250 Y 1 Z 1

Car 7 (Mesh Filter)
Mesh Car7

Mesh Renderer
Materials 1
Element 0 Car

Lighting
Cast Shadows On
Receive Shadows ✓
Contribute Global Illu
Receive Global Illum Light Probes

Probes
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)

Additional Settings
Motion Vectors Per Object Motion
Dynamic Occlusion ✓

Figure I.12: Inspector window

Project window

The project window contains all the project files used in that specific project, it can include Assets, GameObjects or scripts. **Figure I.13**

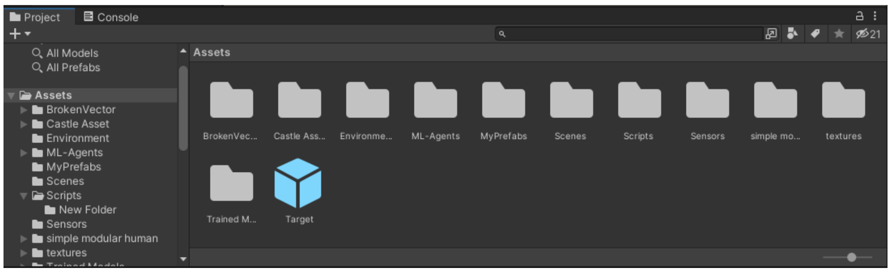


Figure I.13: Project window

I.8 ML-Agents

ML-Agents is a framework for machine learning (ML) designed by Unity to be used in the Unity Editor for the development of capable AI in games.

It is used for multiple types of machine learning (ML)including Reinforcement learning (RL) where an agent is taught to gather information through observations and take action in order to develop a policy, which is essentially a set of instructions to follow in certain situations. This policy is set up to obtain the most rewards possible from performing the actions.

The details of reinforcement learning (RL) have been discussed in the first part of this chapter.

There are four primary parts to Unity ML-Agents. The Learning Environment, the External Communicator, and the Python API.

Learning environment: This represents our project scenes developed in the Unity editor, they usually include both the environment and the agent.

The project scenes for training can be multiple or a single scene.

The agent is usually assigned to a GameObject using a script and special components. This agent runs on a cycle where it collects observations from the environment and execute actions, the rewards are also assigned to this agent.

Agents in the environment operate in steps [13] and for each step that cycle repeats.

External communicator: connects the Learning Environment with the Python Low-Level API. It lives within the Learning Environment. [8]

Python Trainer: The trainer is not part of the environment, and it holds all machine learning (ML) algorithms and techniques used in the training of the agent.

Python API: It is also not part of the learning environment and it serves as the communication interface between the communicator and the and the Python trainer.

During the first step of training, the agent gathers its initial observations. The external communicator and Python API then passes these observations to the Python trainer, which analyzes them using the desired reinforcement learning (RL) algorithm to get the optimal policy. After that, it sends the agent the best action/actions to perform at the moment, and the cycle repeats for the next step. (The process is shown in **Figure I.14)**

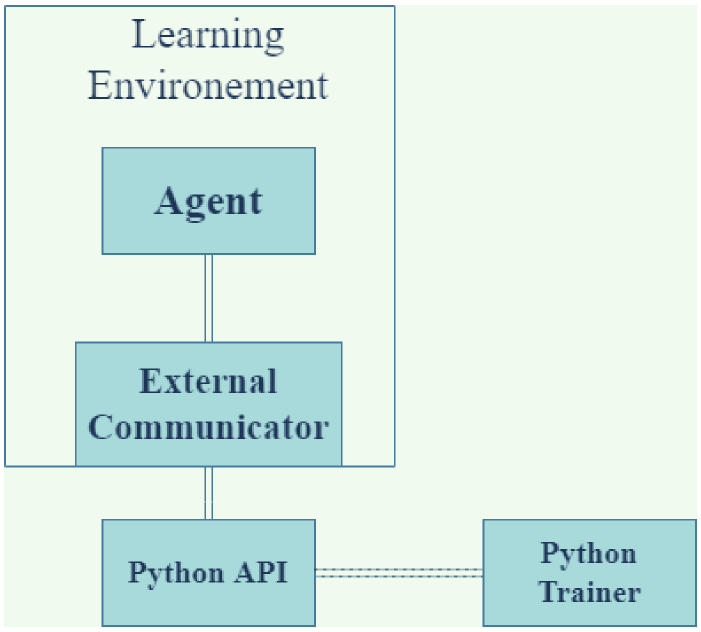


Figure I.14: ML-Agents learning process

Every agent in a learning environment has a specific Behavior set to it and that behavior will determine the type of actions the agent can take.

Agent behaviors can fall under one of three types:

Heuristic: The actions are chosen by the player through an input, such as a keyboard.

At this phase, neither the reward signals nor the observation states have any effect.

Learning: The trainer sends actions to the agent after receiving observations from him using the communicator. The model is still learning during this phase.

Inference: A model that finished training controls the behavior of the agent.

Multiple agents could be involved in a learning environment, and these agents may all have the same behavior type or a separate behavior type as shown in the **Figure I.15**.

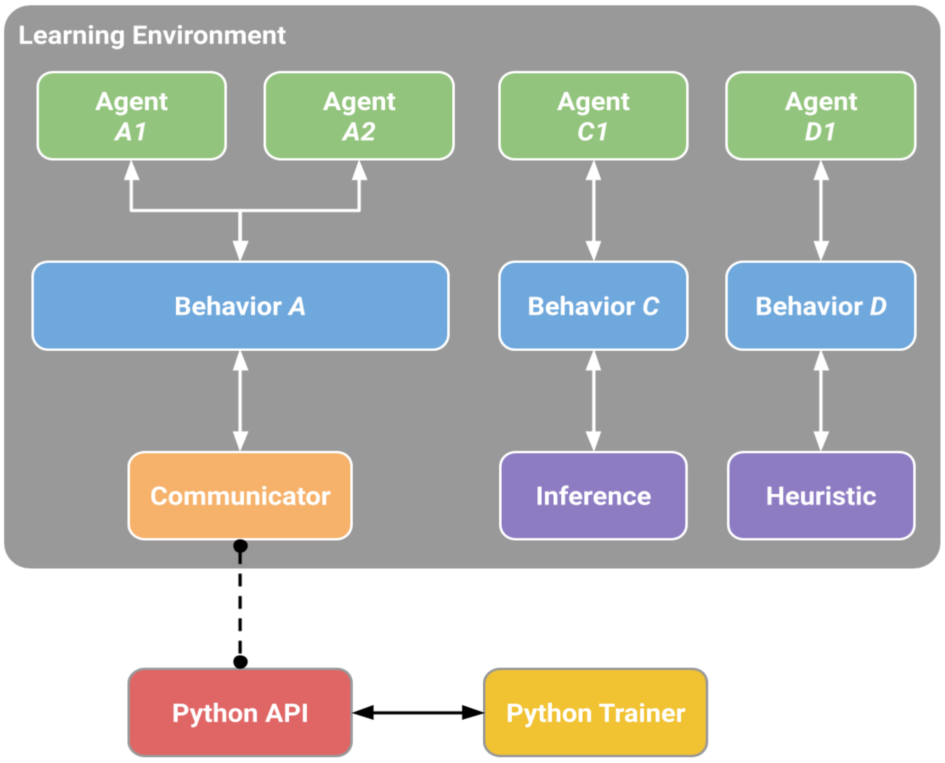


Figure I.15: Agents behaviors in ML-Agents [8]

One of the main benefits of ML-Agents is that it enables us to train the same policy in parallel utilizing many instances of the learning environment.

The trainer can update the policy more effectively and quickly because all instances of the learning environment are connected, training time is reduced the more we add instances.

Goals should be defined in advance in order to get the best training results. Then, these goals should be transformed into three elements: observations, actions, and rewards (negative and positive).

The easiest way to guarantee a seamless training process is to configure the environment and the agent around these three elements.

I.8.1 Agent Configuration

As we highlighted earlier in the chapter the agent uses three components to develop the best policy: Observations, actions and appropriate rewards and punishments.

In this sub-section we discuss how these components are set up with ML-Agents.

<u>I.8.1.1 Behavior Parameters</u>

Every agent in a learning environment has a behavior parameters component in its properties (shown in **Figure I.16)**, this component also known as the brain, defines particular characteristics for particular agent behaviors mentioned earlier.

Behavior Name: This name identifies the unique behavior we're trying to train.

Vector Observation: Space size is the number of observations the agent receives in its environment.

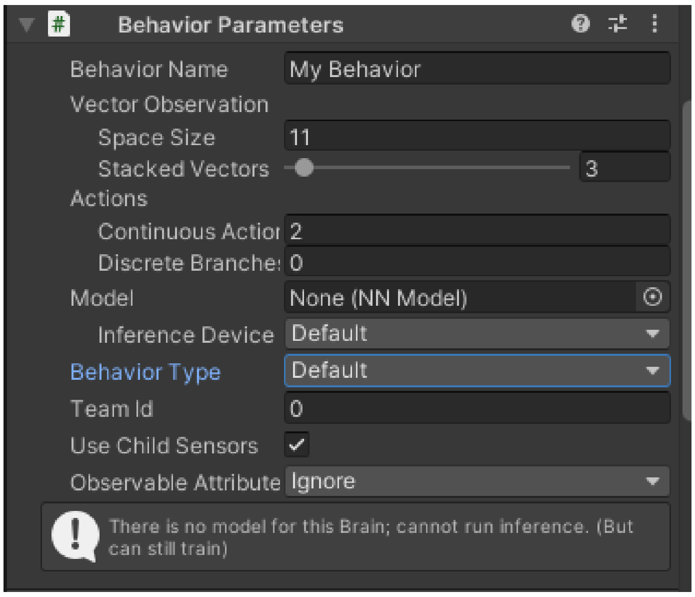
Stacked vectors is the number of previous steps' observations used for the calculation of the next action.

Actions: Choosing either continuous or discrete actions and the number of actions the agent can take.

Model: Using a trained model to control the agent (inference behavior).

Behavior Type: Type of behavior the agent is following; they can be:

- Heuristic.
- Inference.
- Default(Learning**).**



Behavior Parameters

Behavior Name	My Behavior
Vector Observation	
Space Size	11
Stacked Vectors	3
Actions	
Continuous Actior	2
Discrete Branche:	0
Model	None (NN Model)
Inference Device	Default
Behavior Type	Default
Team Id	0
Use Child Sensors	✓
Observable Attribute	Ignore

There is no model for this Brain; cannot run inference. (But can still train)

Figure I.16: Agent Behavior Parameters

I.8.1.2 Observations

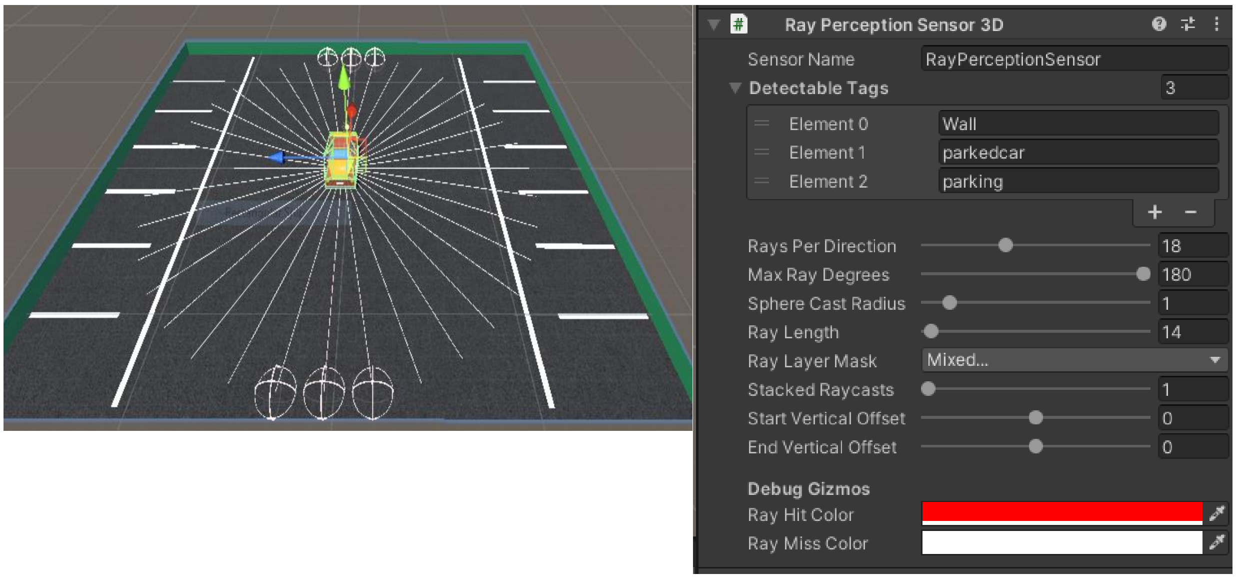
Observations are a way for the agent to collect information in the environment this information is used to decide the agent’s next action.

Observations are split into two categories: Sensors and custom observations.

Some examples of sensors include ray perception sensors and camera sensors. We’ll only explain Ray perception sensors since it’s the only type used in our project.

Ray perception sensors 3D: is a ray-based type of observation that shoots out rays in various directions and angles, these rays interact with objects in the environment when they hit them. It uses tags to identify the object it hits and automatically adds these observations to the agent.

This sensor can simply be added as a component to the agent and configured as needed. (as can be seen in **Figure I.17)**



Ray Perception Sensor 3D

Sensor Name	RayPerceptionSensor
Detectable Tags	3
Element 0	Wall
Element 1	parkedcar
Element 2	parking
Rays Per Direction	18
Max Ray Degrees	180
Sphere Cast Radius	1
Ray Length	14
Ray Layer Mask	Mixed...
Stacked Raycasts	1
Start Vertical Offset	0
End Vertical Offset	0
Debug Gizmos	
Ray Hit Color	🟥
Ray Miss Color	⬜

Figure I.17: Ray Perception Sensors 3D Setup

Custom Observations: Custom observations can be added to the agent using the ML-Agents special function CollectObservations.

Inside this CollectObservations function we can use the sensor. AddObservation function to observe a specific element in the environment. For example: a car’s position in an environment and its velocity.

It is important to keep track of the size of the observations when using the CollectObservations function to properly configure the behavior parameters for the agent.

The size of a single observation can be greater than 1. For instance, if an agent is observing its own location in an environment, it will be searching for its x, y, and z coordinates, which equate to a space size of 3.

I.8.1.3 Actions

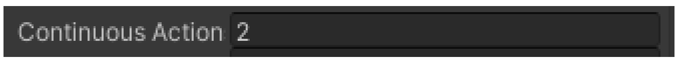
Following the relay of the observations to the trainer, the agent is given actions to take. Good actions configuration enables better environment exploration, which in return results in faster and more effective learning.

Actions in ML-Agents are set up using the unique method OnActionReceived, where we can set up any action we want the trainer to explore.

There are two types of actions supported: Continuous and Discrete [10]

Continuous actions: This type of action is represented using float values ranging from -1 to 1. It is used to express complex movements like acceleration for example, where the trainer learns to manage it by passing float values in that range

The agent's behavior parameter must be configured with the desired number of continuous actions. For instance, if we have accelerating and turning actions we set the value of continuous actions to 2, The trainer would learn to control the acceleration and the turning angle.

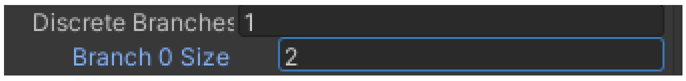

Continuous Action 2

Discrete actions: Discrete actions are best described as binary actions because they can either be active or not.

Discrete actions are represented as branches with each branch being a certain size. The size of the branch is the possible movements in that branch, for example: the branch is turning and the size of the branch is 2 for left and right.

For each branch only a single movement is executed at a time.

Number of branches and their size is configured in the agent's behavior parameter.


Discrete Branches 1
Branch 0 Size 2

It is important to note that during the initial phase of the learning, the trainer doesn't have a reference for the actions. The training algorithm simply tries different values for the action list and observes the effect on the accumulated rewards over time and many training episodes. [14]

I.8.1.4 Rewards

In order for the agent to develop and improve its behavior, rewards have to be setup correctly to control the optimization of the policy.

The agents can earn rewards throughout the training process in the form of floats; positive values are called rewards and reinforce the actions taken to gain them, while negative values are called penalties and punish the actions taken to obtain them.

These rewards are configured by code using the ML-Agents AddReward() or SetReward functions ().

I.8.2 Proximal Policy Optimization Algorithm(PPO)

Developed by the OpenAI team and launched in 2017, Proximal Policy Optimization (PPO) is a Reinforcement learning (RL) algorithm. It is the default algorithm used in ML-Agents.

PPO is a policy gradient method which means that the algorithm modifies its policy by using a limited batch of experiences gathered by interacting with the environment.

When the policy is modified using that batch, the experiences are discarded and a newer batch of experiences is gathered using the new modified policy.

PPO is viewed as a stable method due to the fact that was designed to make sure that the policy update does not significantly alter the previous policy.

I.8.3 Hyperparameters

Hyperparameters have a significant impact on the learning process, and occasionally fine-tuning particular parameters might be the difference between accomplishing the desired training objective and not.

There are various hyperparameters associated with various training methodologies; in this subsection, we only include those associated with the PPO methodology.

Hyperparameters are used by ML-Agents in a configuration file, and the trainer can only identify these parameters if the configuration file follows a specific format.

```
behaviors:
  My Behavior:
    trainer_type: ppo

    hyperparameters:
      batch_size: 512
      buffer_size: 5120
      learning_rate: 3.0e-4
      beta: 0.01
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear

    network_settings:
      normalize: True
      hidden_units: 512
      num_layers: 3
      vis_encode_type: simple

    reward_signals:
      extrinsic:
        strength: 1.0
        gamma: 0.99

             curiosity:
        strength: 0.01
        gamma: 0.99
        encoding_size: 256

    max_steps: 12000000
    time_horizon: 512
    summary_freq: 30000
```

All of the hyperparameters are explained in detail in the configuration section of the ML-Agents documentation [7].

I.9. Conclusion

This chapter focused on explaining the fundamentals of reinforcement learning (RL) and the usage of the Unity video game engine to create simulations and video games. We also covered ML-Agents and how it functions, including how agents are set up.

We discuss how these components will be used to accomplish our goal in the following chapter

Chapter II: Autonomous parking using Unity and Reinforcement Learning (RL)

II.1 Introduction

We covered the fundamentals of reinforcement learning (RL) and game design with the Unity engine in the previous chapter.

In this chapter, we discuss how we created our project using the Unity editor and then integrated the ML-Agents Toolkit to achieve our objective.

II.2 Working Environment

We used a variety of tools in the creation of this project; thus, we feel it is important to provide a quick overview of each one

II.2.1 Unity Game Engine

Unity game engine was used to create the simulation environment to both train the Reinforcement learning model and also test it.

Unity Game Engine was explained previously in chapter 1 section 1.7.

II.2.2 C-sharp (C#) Programming Language

C# is a general-purpose programming language that Microsoft developed in the year 2000 under the direction of Anders Hejilsberg with the goal of making it a modern C-like object-oriented language and a competitor to Java.

It is one of the most widely used programming languages today since it is simple to learn and has a ton of documentation. It is used to create a range of software, including desktop, mobile, and web apps as well as videogames for PCs and consoles.

Given that Unity uses it as its programming language, it was used in our project to write scripts that control how the environment and the agent function.

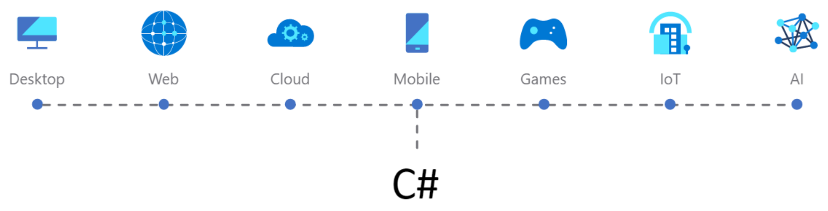


Figure II.1: Domains that C# is used in.

II.2.3 Visual Code Studio

Visual studio code is a powerful open-source Integrated Development Environments (IDE) that was designed by Microsoft and first released in 1997.

On top of having a simplistic design and offering various extensions, it supports hundreds of programming languages including every major one.

Given that Microsoft made sure Visual Code Studio's documentation was excellent and made the IDE free to download from: code.visualstudio.com it is no surprise that it has been voted as the most preferred IDE to use according to StackOverflow's 2022 Developers survey. [14]

It is worth noting that Visual Code Studio is supported on various platforms including Linux, Windows and MacOS.

Because Unity has built-in support for Visual Studio code as an external script editor, it was used in our project. [15]

We used it to create the scripts and to edit the configuration file for the hyperparameters.

II.2.4 Google Drive

Google drive is cloud storage service that enables its users to store and access files of various types on servers online; created by Google and released in the early part of 2012 it quickly became the most used cloud service in the world.

Some of the reasons that make it incredibly popular is that it is free and everyone who has a Google account automatically gets a Drive account with the 15GB of storage space therefore, if you already use Google, you avoid the effort of setting up a new account just for your online storage.

Additionally, when files are stored in the cloud, you may access them from any location using any internet-connected device including phones, tablets and PCs.

We used it to be able to upload our files to Google Colaboratory for training.

II.2.5 Google Colaboratory

Developed by Google Research, Google colaboratory or Colab for short is a cloud-based environment that allow to write, run, store and share code with the help of Google Drive.

Google Colab is the best tool for working with machine learning; it provides free Graphical processing units (GPUs) and Tensor processing units (TPUs) in addition to essential deep learning (DL) and data analysis tools like Tensorflow, Keras, Pytorch, Numpy, Scipy, and Pandas

Colab requires no setup and runs fully in the cloud and offers free powerful hardware which is why we decided to use it for training.

II.3 Concept Introduction

In this project, we create a controllable car that can make the most fundamental movements, such as moving forward and backward, turning left and right, and stopping.

II.3.1 Working Concept

The car spawns in a parking lot with four walls surrounding it, seven parking spots on two sides, every parking spot has a car parked in it except for one vacant spot.

Our objective is to train the AI to locate the available parking spot and approach it.

The way we achieve this is by rewarding the car every time it enters the available parking spot, and punish it every time it hits a wall or a parked car.

In reinforcement learning an episode is the length of the simulation in which the agent interacts with the environment until it reaches a certain terminal state that initiates a reset to its initial state.

In our case the episode starts when the car spawns and ends when the car makes contact with a wall or a parked car. The same happens if the car successfully finds the vacant spot. Each time a new episode of learning starts, the car and the available parking spot spawn in a random position in the parking lot to maximize the amount of learning and avoid having the same scenario each episode.

The process is shown in the flowchart below.

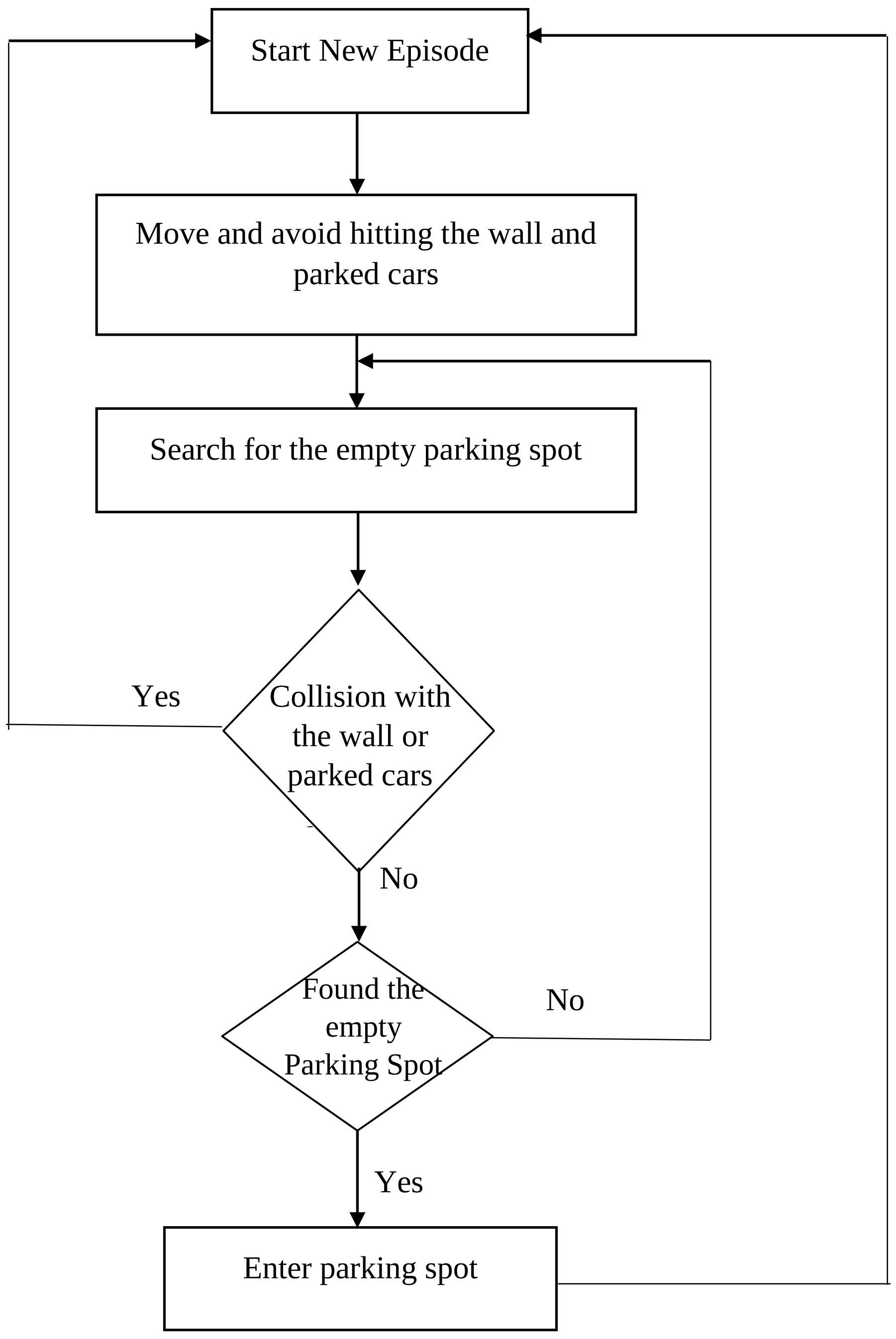

Start New Episode
Move and avoid hitting the wall and parked cars
Search for the empty parking spot
Collision with the wall or parked cars
Yes
No
Found the empty Parking Spot
No
Yes
Enter parking spot

Figure II.2: Flowchart showing the agent working concept

II.3.2 Design Steps

The project design is split into two parts: designing the Environment and designing the Agent.

Environment: For the environment we first develop a parking lot in Unity (Described in Section II.5.1) and then we configure the random respawn of the parked cars and the parking spot which is done via script using VS Code (Described in Section II.6.2.1).

Agent: for the agent we first set up a car in Unity (Described in Section II.5.2) and also specify the behavior parameters, after that we set up a script using VS code to spawn the car in a random position in the parking lot and implement functions for getting observations from the environment, taking actions, and getting rewards. (Described in Section II.6.2.2).

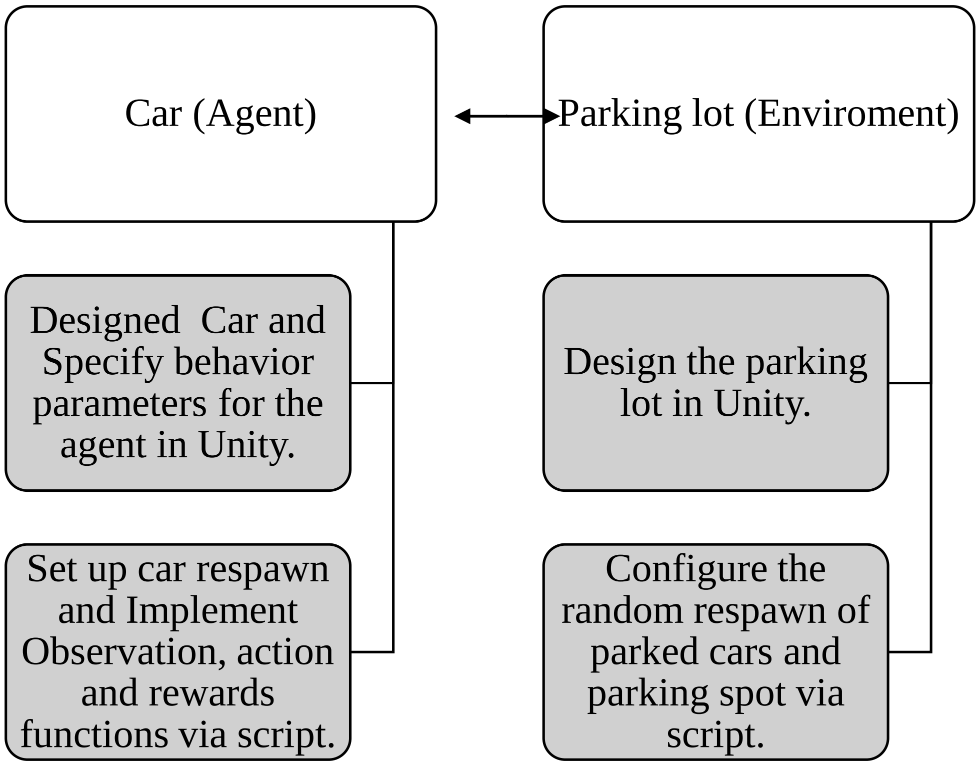

Car (Agent)
Parking lot (Enviroment)
Designed Car and Specify behavior parameters for the agent in Unity.
Design the parking lot in Unity.
Set up car respawn and Implement Observation, action and rewards functions via script.
Configure the random respawn of parked cars and parking spot via script.

Figure II.3: Project Design

When the training starts the trainer creates a Neural Network model. As car interacts with the environment it receives a reward value and a new state that include observations, these observations are then fed into the input layer of the neural network model, the output layer then produces an action to take and the cycle repeats.

The inputs of the neural network can be modified using Behavior Parameters component by changing the Vector observations space size and stacked vectors value.

The outputs of the neural network can also be modified by modifying the type and size of the actions in the Behavior Parameters component.

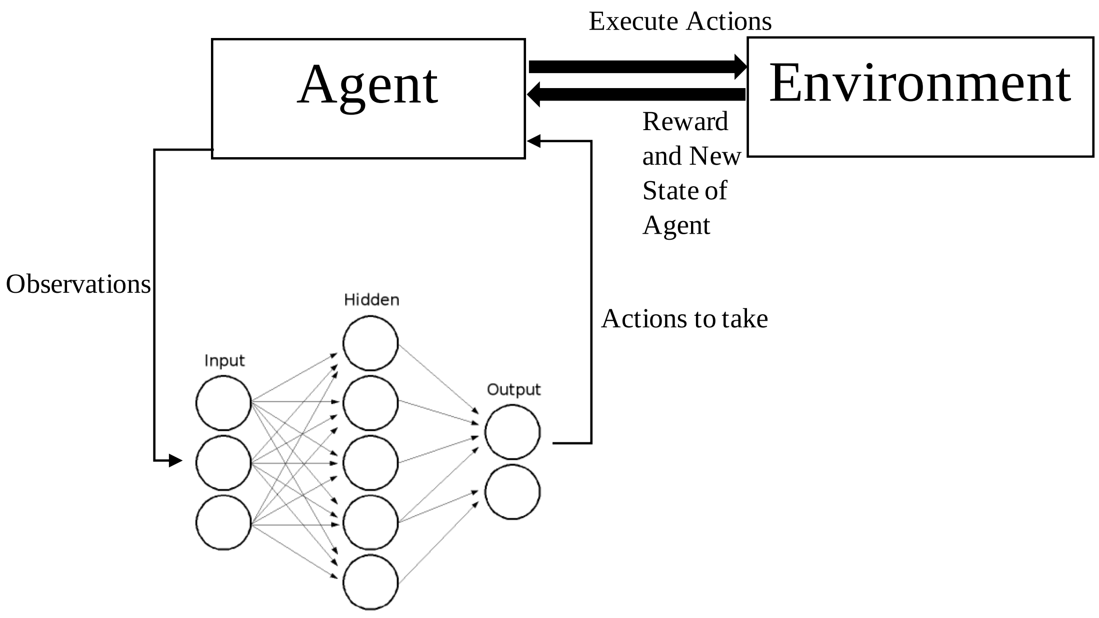


Figure II.4: Agent/Environment/NN Model interaction

II.4 Assets Used

During the planning stages of our project, we discovered that we could benefit from unity's extensive collection of assets found in the unity asset store and community websites.

We used an asset called 3D Low Poly Cars [7] which was available for free. This pack contains models of 10 different cars, a limousine and a truck. **(Figure II.5)**

Figure II.5: 3D Low Poly Cars Asset

II.5 Scene Overview

As depicted in **Figure II.6**, our scene titled" SampleScene" contains 2 main parts: the environment and the agent both consisting of different game objects and have different properties and to which we attached different script.

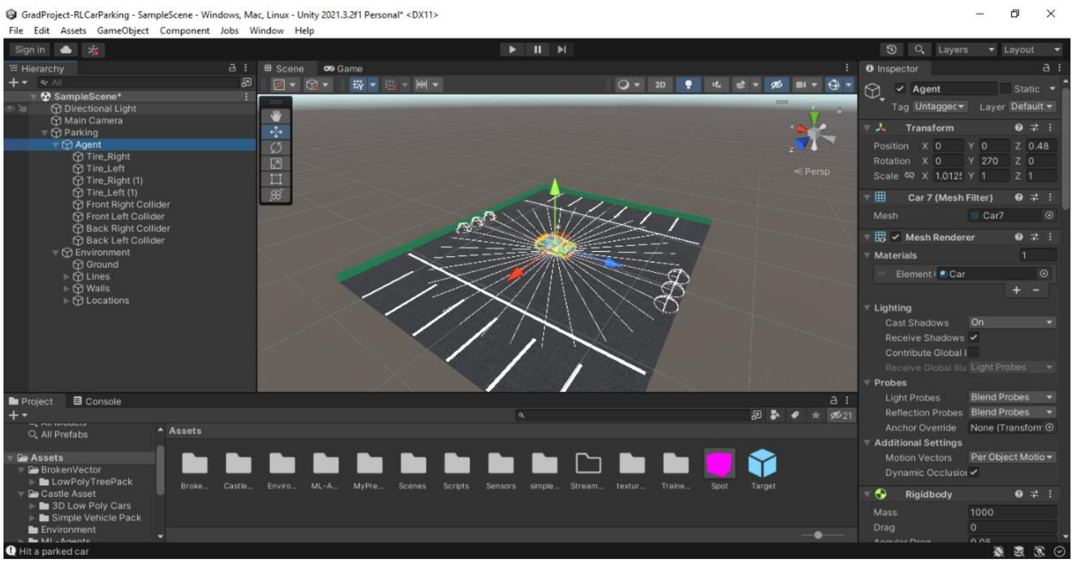


Figure II.6: Unity Editor Overview

Our scene also contains a camera and a directional light, the camera is the point of view we get once we start the simulation it can be angled and positioned as wanted.

When selecting the camera from the sample scene we get a small preview of the view angle we have during simulation this allows the adjustment of the camera to be easier. (**Figure II.7**)

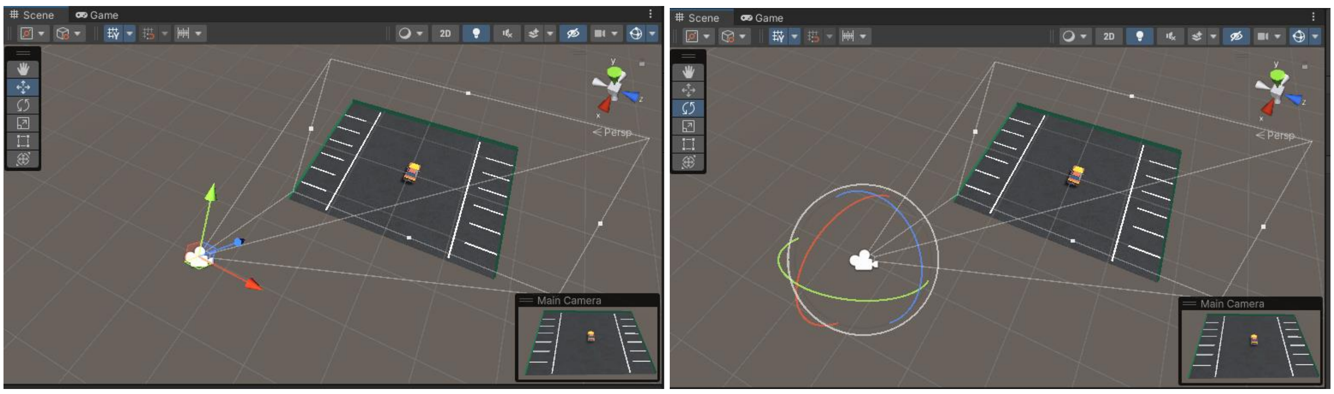
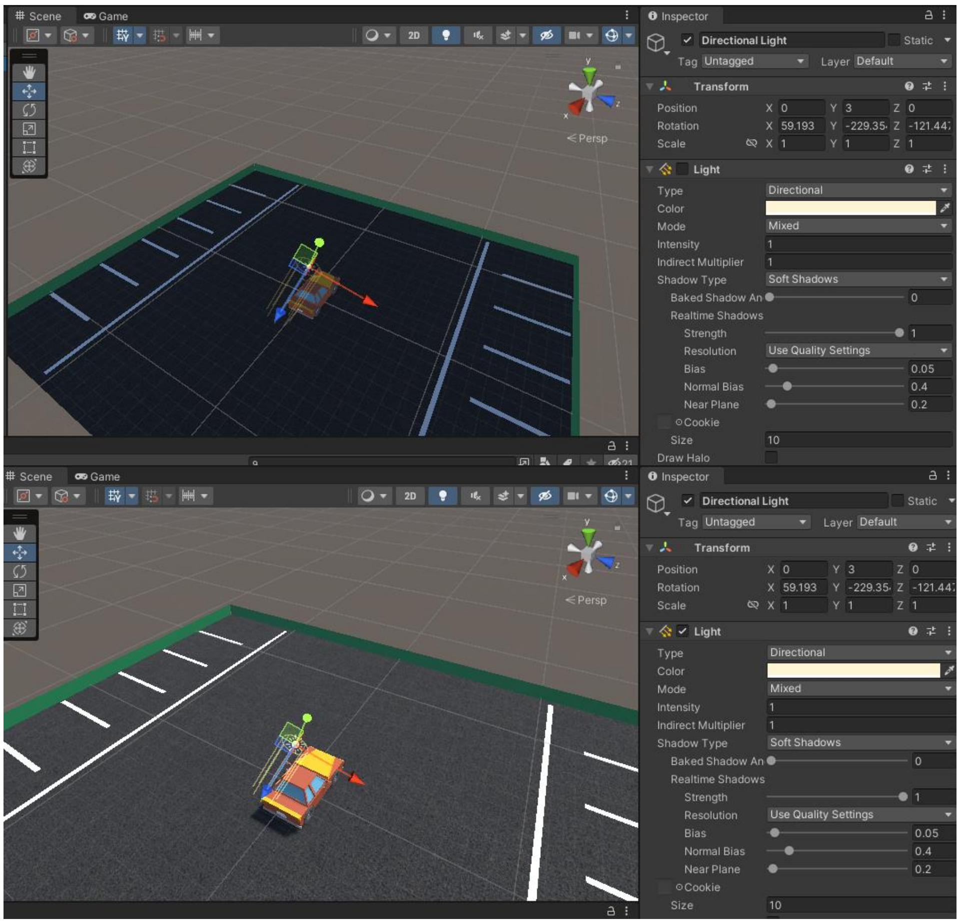


Figure II.7: Camera in Unity

The directional light allows for better visibility and can also be angled, adjusted or turned off easily.



```
✓ Directional Light                Static
Tag Untagged        Layer Default
Transform
Position      X 0        Y 3         Z 0
Rotation      X 59.193   Y -229.35   Z -121.44
Scale         X 1        Y 1         Z 1
Light
Type                 Directional
Color
Mode                 Mixed
Intensity            1
Indirect Multiplier  1
Shadow Type          Soft Shadows
  Baked Shadow An    0
  Realtime Shadows
    Strength         1
    Resolution       Use Quality Settings
    Bias             0.05
    Normal Bias      0.4
    Near Plane       0.2
  Cookie
  Size               10
Draw Halo
```

```
✓ Directional Light                Static
Tag Untagged        Layer Default
Transform
Position      X 0        Y 3         Z 0
Rotation      X 59.193   Y -229.35   Z -121.44
Scale         X 1        Y 1         Z 1
✓ Light
Type                 Directional
Color
Mode                 Mixed
Intensity            1
Indirect Multiplier  1
Shadow Type          Soft Shadows
  Baked Shadow An    0
  Realtime Shadows
    Strength         1
    Resolution       Use Quality Settings
    Bias             0.05
    Normal Bias      0.4
    Near Plane       0.2
  Cookie
  Size               10
```

Figure II.8: Example of how light works in Unity.

As noticed from **Figure II.8,** disabling the light makes the scene much darker.

<u>II.5.1 The Parking Environment</u>

The environment is composed of a parking lot with 14 parking spots divided onto two sides, 13 parked cars and a vacant parking spot for the agent to find. (**Figure II.9)**

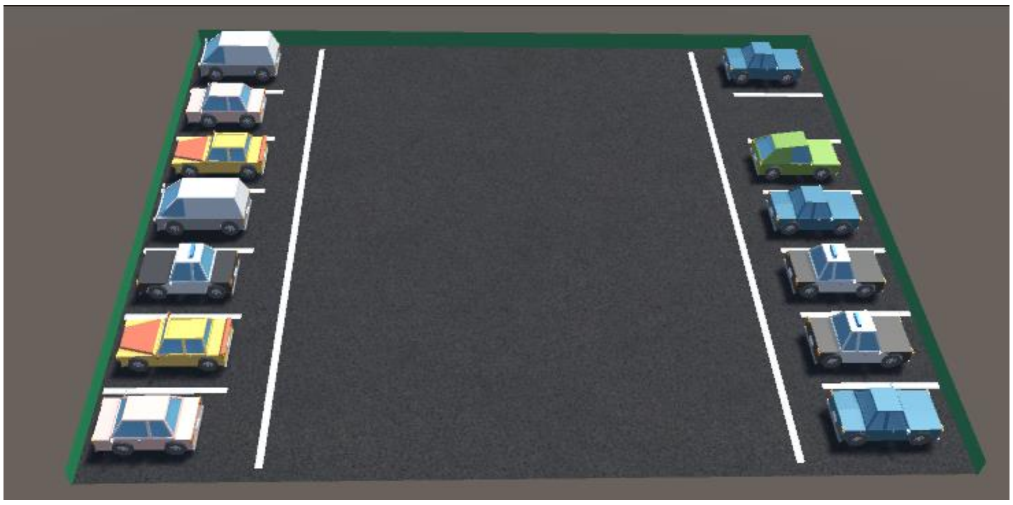


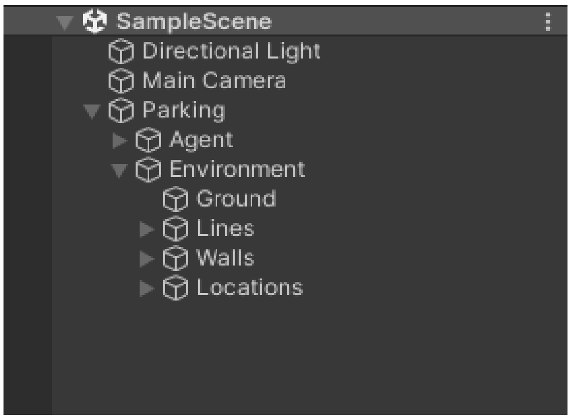
Figure II.9: The parking environment.

We divided the environment into three important components in order to better understand it:

- The parking lot
- The parked cars
- The available parking spot

II.5.1.1 The parking lot

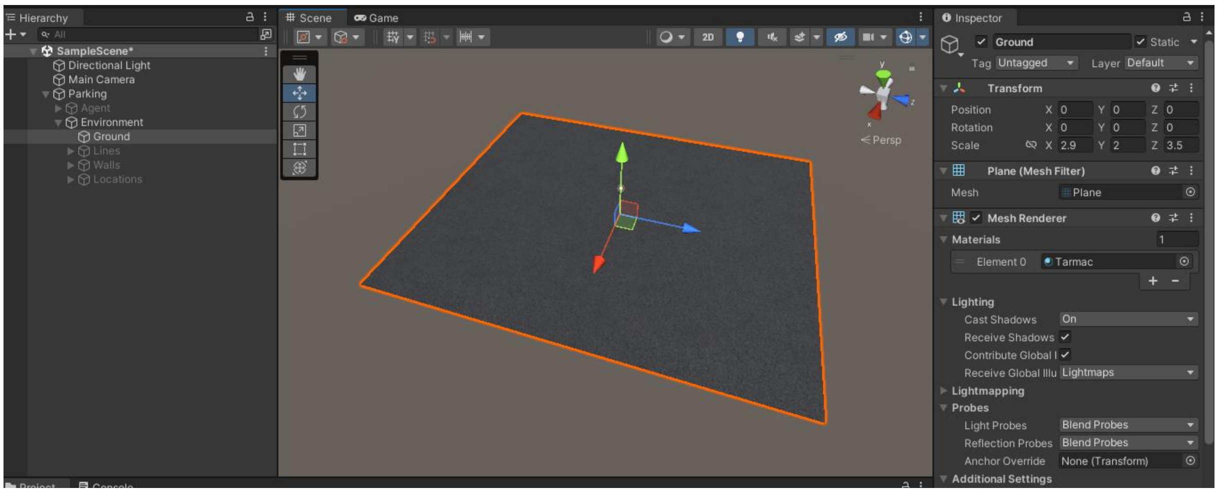
for the parking lot we had to create several game objects, grouped into 4 components as seen in **Figure II.10**.



```
SampleScene
  Directional Light
  Main Camera
  Parking
    Agent
    Environment
      Ground
      Lines
      Walls
      Locations
```

Figure II.10: Hierarchy window of the project

- **Ground:** The ground is a simple plane in the shape of a square with a Tarmac texture and a Mesh collider added to make it interactable with every element on it, this makes it so we can put Game Objects on it. (**Figure II.11)**

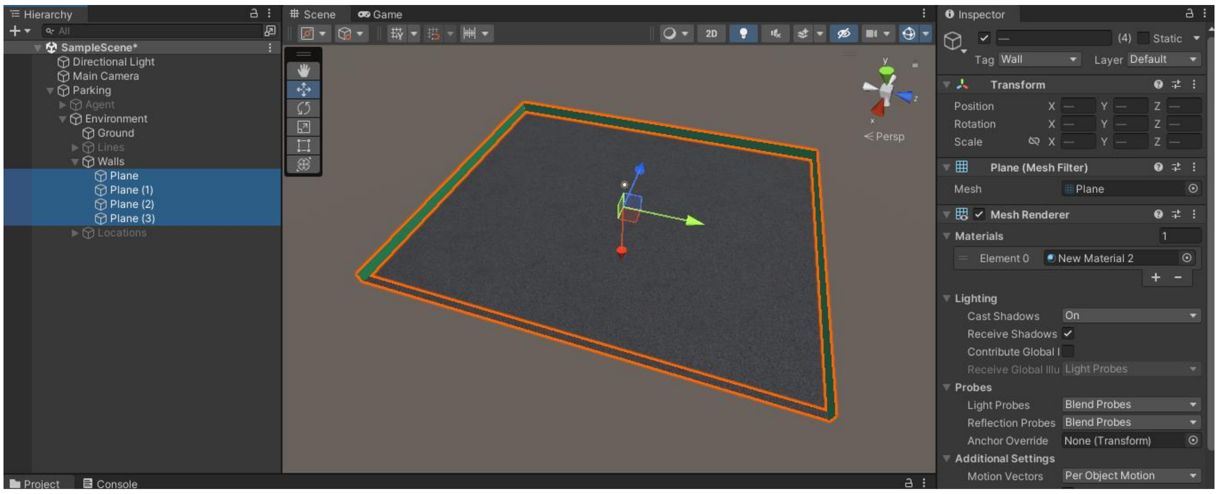


```
Inspector
Ground  ✓ Static
Tag Untagged   Layer Default
Transform
Position  X 0    Y 0  Z 0
Rotation  X 0    Y 0  Z 0
Scale     X 2.9  Y 2  Z 3.5
Plane (Mesh Filter)
Mesh  Plane
Mesh Renderer
Materials  1
Element 0  Tarmac
Lighting
Cast Shadows  On
Receive Shadows  ✓
Contribute Global I  ✓
Receive Global Illu  Lightmaps
Lightmapping
Probes
Light Probes  Blend Probes
Reflection Probes  Blend Probes
Anchor Override  None (Transform)
Additional Settings
```

Figure II.11: Ground of the environment.

- **Walls:** The walls are 4 planes in the shape of a rectangle covering the width and length of the ground so the agent cannot fall from the environment.

A box collider was also added to each wall so when the agent hit the wall it is detected. **(Figure II.12)**



(4) Static
Tag Wall Layer Default
Transform
Position X — Y — Z —
Rotation X — Y — Z —
Scale X — Y — Z —
Plane (Mesh Filter)
Mesh Plane
Mesh Renderer
Materials 1
Element 0 New Material 2
Lighting
Cast Shadows On
Receive Shadows ✓
Contribute Global I
Receive Global Illu Light Probes
Probes
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)
Additional Settings
Motion Vectors Per Object Motion

Figure II.12: Walls of the environment.

Every GameObject that is a wall has a "Wall" tag on it, making it easy for the agent to identify the object as one and for the script to quickly access grouped items under this tag. **(Figure II.13)**

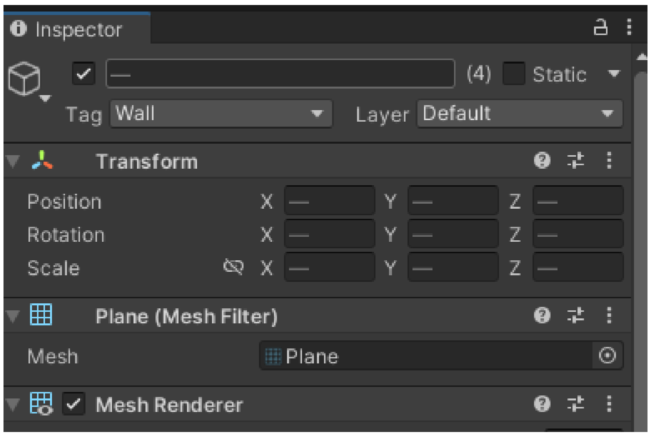

Inspector
(4) Static
Tag Wall Layer Default
Transform
Position X — Y — Z —
Rotation X — Y — Z —
Scale X — Y — Z —
Plane (Mesh Filter)
Mesh Plane
Mesh Renderer

Figure II.13: Wall Tag

- **Lines:** The lines are a number of thin planes that have been placed on the ground to divide the parking spaces. **(Figure II.14)**

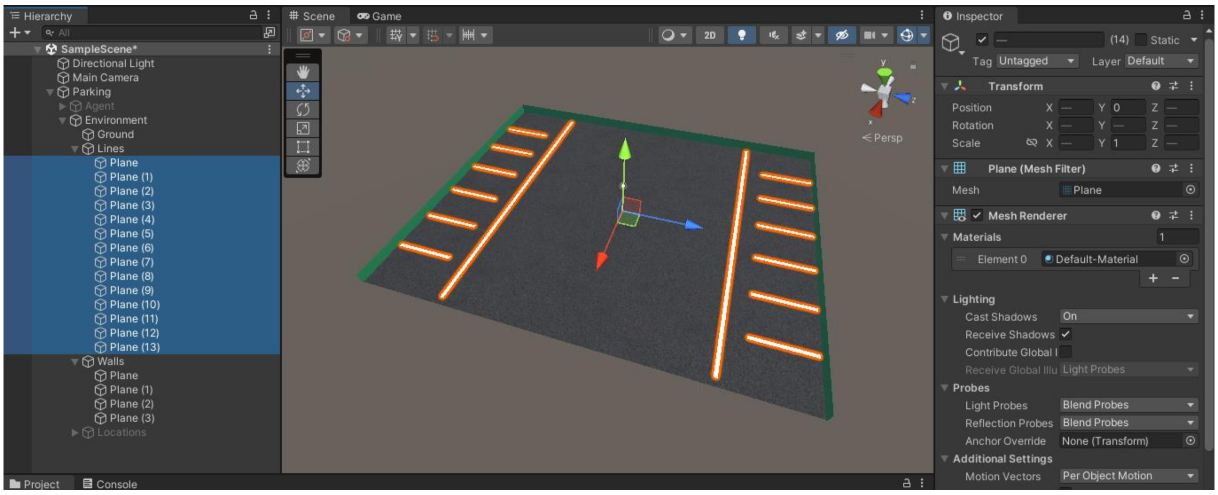


Figure II.14: Lines separating parking spaces.

- **Locations:** Locations is made of 14 empty game objects that are placed in the middle of each parking space. These game objects are used by a script to indicate where vacant parking spot spawns, as well as the random positions in which parked cars spawn. **(Figure II.15)**

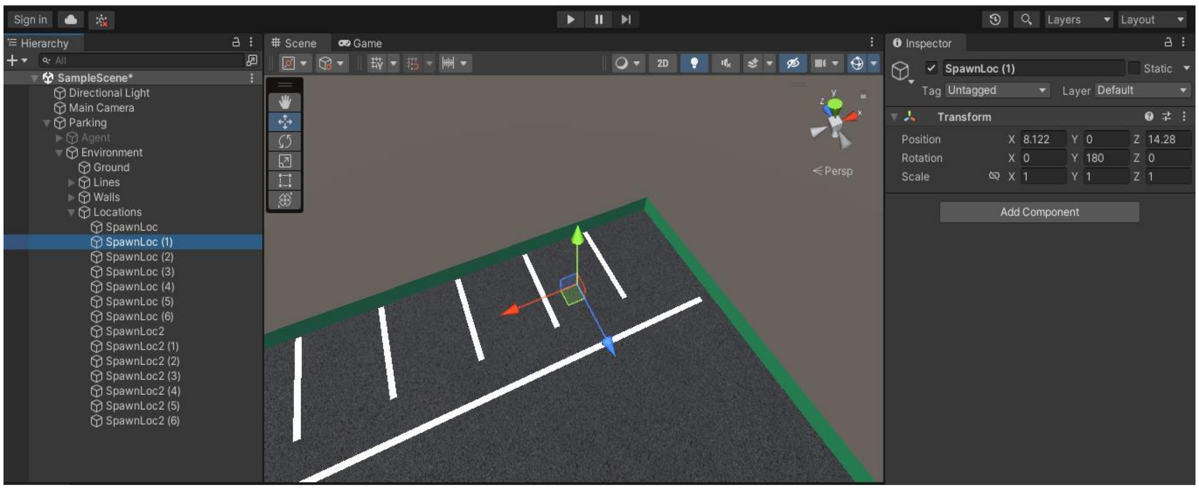


Figure II.15: Parked cars and parking spot spawn locations.

II.5.1.2 Parked Cars

Parked cars occupy 13 of the parking lot's 14 slots; however, because each episode uses a random spawn, they are not permanently placed in the scene, but rather spawned in using a script.

In order to do this, we utilized the 3D Low Poly Cars asset cars that we previously mentioned. **(Figure II.16)**

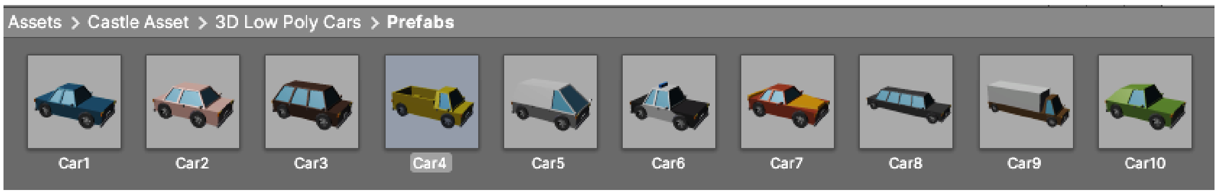
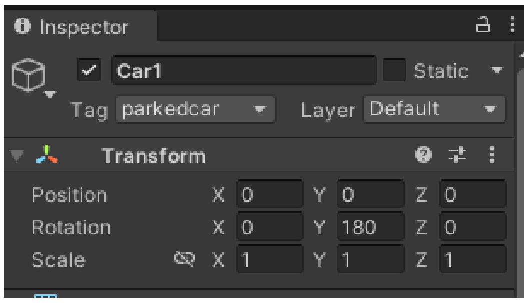


Figure II.16: Prefabs from 3D Low Poly Cars asset.

To help the agent identify the cars from this asset, "parkedcar" tags were placed on them.


Inspector

Car1 Static

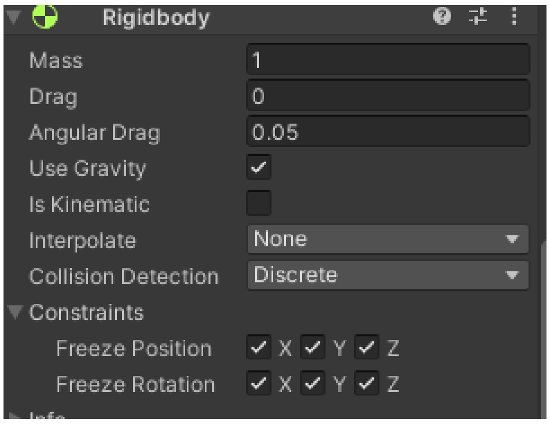
Tag parkedcar Layer Default

Transform

Position	X	0	Y	0	Z	0
Rotation	X	0	Y	180	Z	0
Scale	X	1	Y	1	Z	1

Figure II.17: parkedcar Tag.

If our agent collides with one of these vehicles, it sets it into motion, forcing it to move out of its spot, to avoid this we froze the x, y, and z positions of these cars in the environment so that they would stay in the location that they spawned in.


Rigidbody

Mass	1
Drag	0
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	✓ X ✓ Y ✓ Z
Freeze Rotation	✓ X ✓ Y ✓ Z

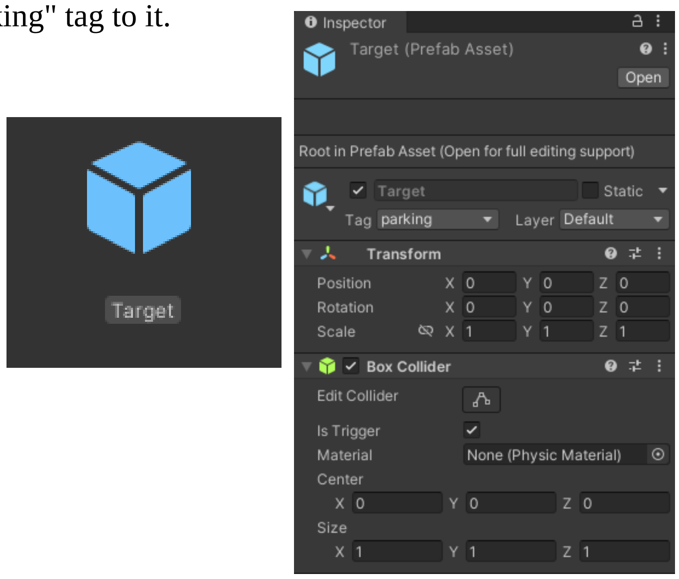
Figure II.18: Parked Cars rigidbody component

II.5.1.3 Parking spot

Since an interaction with another object must be detected by the agent before it can recognize that it has parked.

We created an empty GameObject called "Target," to which we added a box collider component. When the agent collides with this object, it will detect that it has arrived at the vacant parking spot. (**Figure II.19)**

In order for the agent to identify this object as the open parking spot when it collides with it, we added a "open parking" tag to it.



Target (Prefab Asset)

Root in Prefab Asset (Open for full editing support)

✓ Target Static

Tag parking Layer Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

✓ Box Collider

Edit Collider

Is Trigger ✓

Material None (Physic Material)

Center X 0 Y 0 Z 0

Size X 1 Y 1 Z 1

Figure II.19: Target GameObject

II.5.2 The Agent

Our agent is a car that maneuvers in our environment and searches for the empty parking spot, for this we used the "Car7" Prefab from the 3D Low Poly Cars asset show in **Figure II.20**.

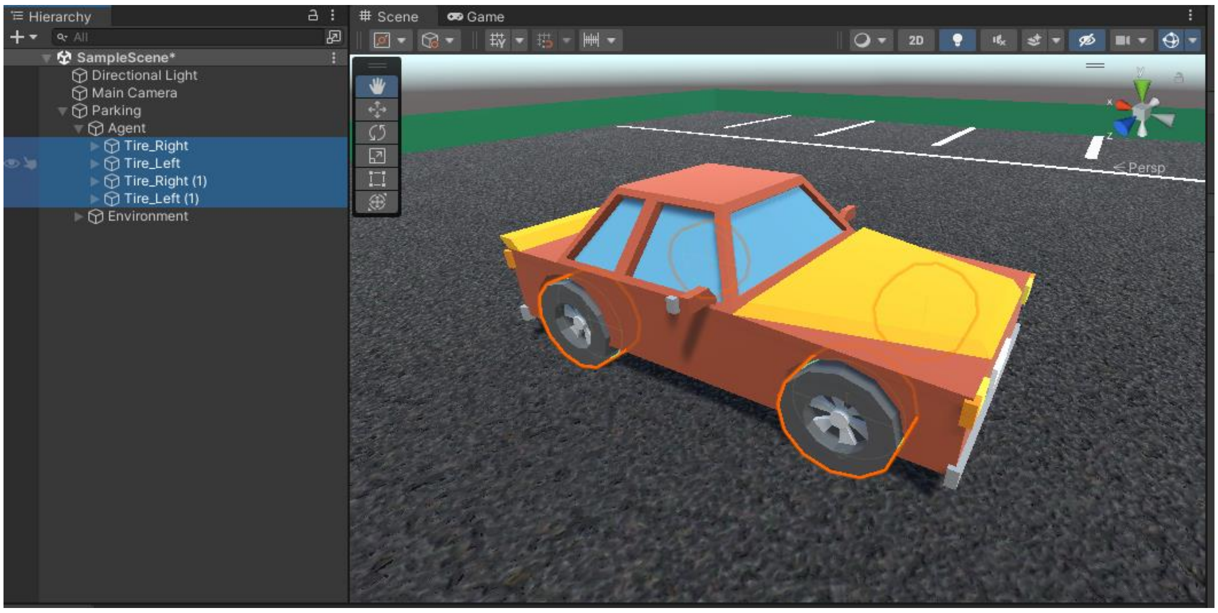
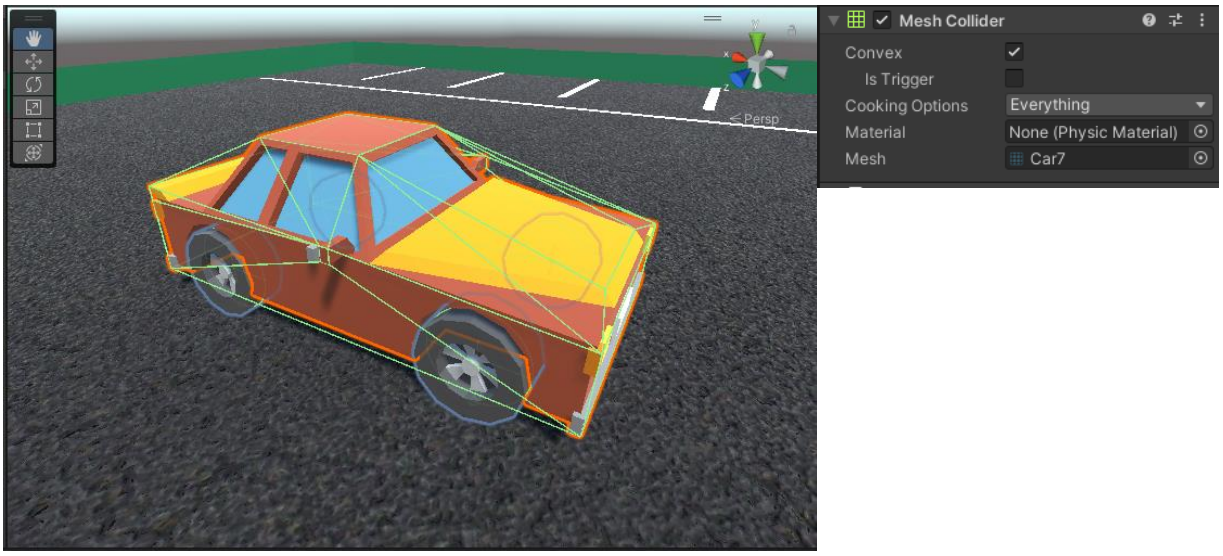


Figure II.20: Agent Overview

The car was equipped with a mesh collider to enable collisions with parked vehicles, walls, and the empty parking spot.

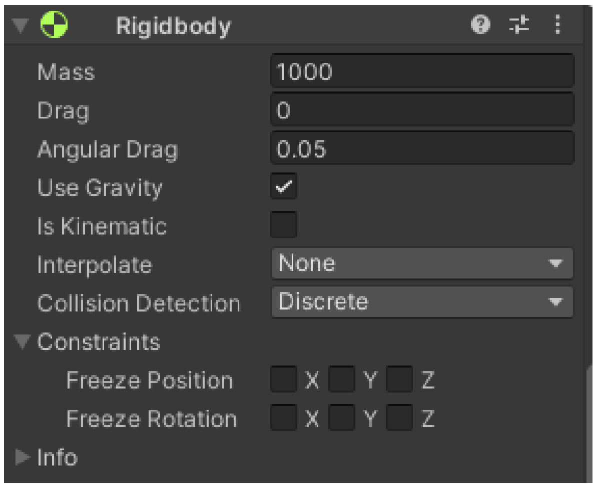


Mesh Collider

Convex	✓
Is Trigger	
Cooking Options	Everything
Material	None (Physic Material)
Mesh	Car7

Figure II.21: Agent Meshcollider component

The RigidBody component shown in **Figure II.22** enables us to change the car's mass to give the movement a sense of realism, therefore the mass was set at 1000 kg.

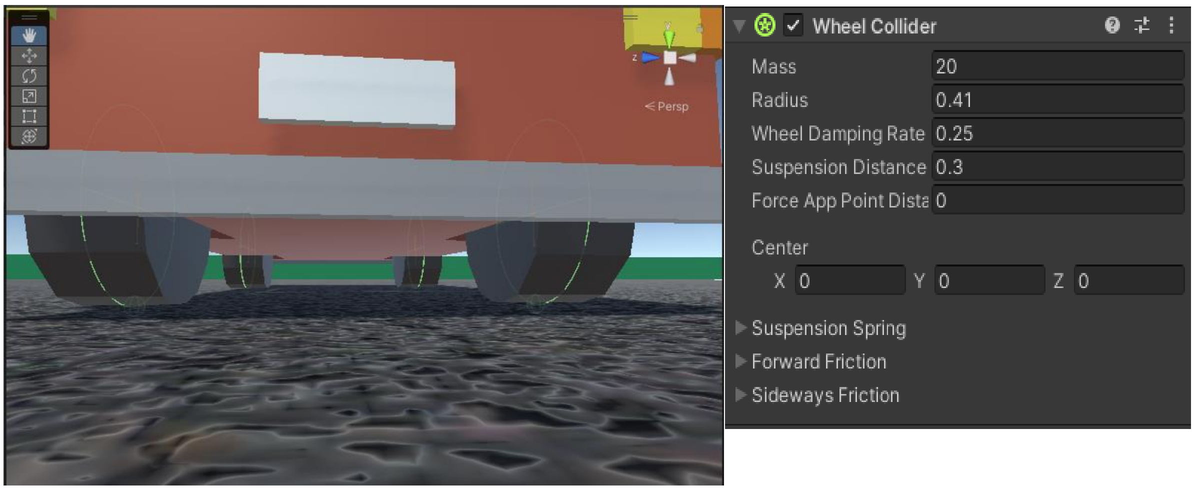


Rigidbody

Mass	1000
Drag	0
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	X Y Z
Freeze Rotation	X Y Z
Info	

Figure II.22: Agent Rigidbody component

The car also comes with four wheels, to which we added the wheel collider component to. These wheel colliders enable the car's four wheels to have contact with the ground, and thus are responsible for the movement of the car. (**Figure II.23)**



Wheel Collider

Mass	20
Radius	0.41
Wheel Damping Rate	0.25
Suspension Distance	0.3
Force App Point Dista	0

Center
X 0 Y 0 Z 0

Suspension Spring
Forward Friction
Sideways Friction

Figure II.23: Agent Wheel Colliders

II.6 Scripts

The most crucial component of our project is the code, it controls everything from the spawning of the environment and the agent to the movements of the agent and the rewards it should receive.

II.6.1 VS Code Setup

Before we start coding in Unity we have to setup our IDE of choice, for this we went with VS code studio because we were comfortable with the interface and found it simple to set up C# on it.

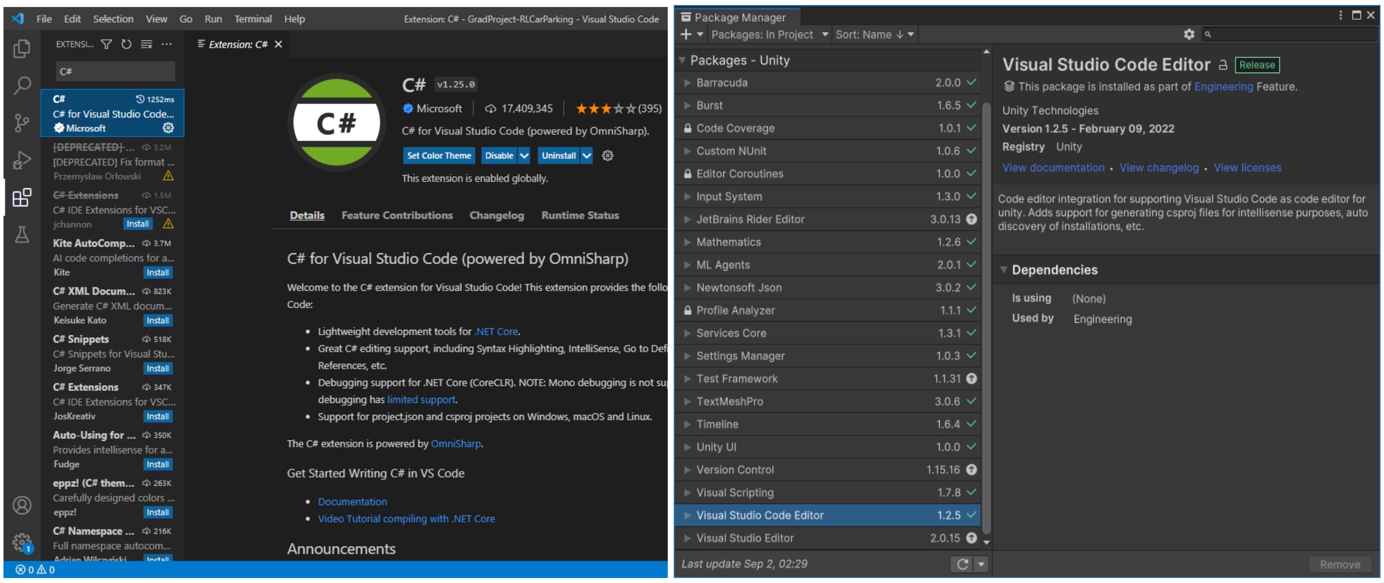
To use C#, VS Code Studio requires us to install an extension. The extension is simple to install from within VS Code Studio by following the instructions below:

- Open Visual Studio Code.
- Click on the extension button found on the left panel.
- Search for the C# extension published by Microsoft.
- Install the extension.

To use this IDE, Unity requires the VS Code Studio package to be installed; fortunately, newer versions of Unity already have this package set up in the editor.

Its is possible to check if the package is installed by following the instructions below:

- Open Unity Editor.
- From the toolbar we click on “Window” and then “Package Manager”
- In the top right of the Package Manager window, we search for “Visual Studio Code Editor”


File Edit Selection View Go Run Terminal Help
Extension: C# - GradProject-RLCarParking - Visual Studio Code
EXTENSI...
C#
Extension: C#
Set Color Theme
Disable
Uninstall
Details Feature Contributions Changelog Runtime Status
Install
Package Manager
Packages: In Project
Sort: Name
Release
Remove

C#	1252ms	C# for Visual Studio Code... / Microsoft
~~[DEPRECATED]~~ ...	3.2M	[DEPRECATED] Fix format ... / Przemysław Orłowski
~~C# Extensions~~	1.5M	C# IDE Extensions for VSC... / jchannon
Kite AutoComp...	3.7M	AI code completions for a... / Kite
C# XML Docum...	823K	Generate C# XML docum... / Keisuke Kato
C# Snippets	518K	C# Snippets for Visual Stu... / Jorge Serrano
C# Extensions	347K	C# IDE Extensions for VSC... / JosKreativ
Auto-Using for ...	350K	Provides intellisense for a... / Fudge
eppz! (C# them...	263K	Carefully designed colors ... / eppz!
C# Namespace ...	216K	Full namespace autocom... / Adrian Wilczyński

C# v1.25.0

Microsoft | 17,409,345 | ★★★☆☆ (395)

C# for Visual Studio Code (powered by OmniSharp).

This extension is enabled globally.

C# for Visual Studio Code (powered by OmniSharp)

Welcome to the C# extension for Visual Studio Code! This extension provides the follo
Code:

- Lightweight development tools for .NET Core.
- Great C# editing support, including Syntax Highlighting, IntelliSense, Go to Defi
References, etc.
- Debugging support for .NET Core (CoreCLR). NOTE: Mono debugging is not su
debugging has limited support.
- Support for project.json and csproj projects on Windows, macOS and Linux.

The C# extension is powered by OmniSharp.

Get Started Writing C# in VS Code

- Documentation
- Video Tutorial compiling with .NET Core

Announcements

0 0

Packages - Unity	
Barracuda	2.0.0 ✓
Burst	1.6.5 ✓
Code Coverage	1.0.1 ✓
Custom NUnit	1.0.6 ✓
Editor Coroutines	1.0.0 ✓
Input System	1.3.0 ✓
JetBrains Rider Editor	3.0.13 ↑
Mathematics	1.2.6 ✓
ML Agents	2.0.1 ✓
Newtonsoft Json	3.0.2 ✓
Profile Analyzer	1.1.1 ✓
Services Core	1.3.1 ✓
Settings Manager	1.0.3 ✓
Test Framework	1.1.31 ↑
TextMeshPro	3.0.6 ✓
Timeline	1.6.4 ✓
Unity UI	1.0.0 ✓
Version Control	1.15.16 ↑
Visual Scripting	1.7.8 ✓
Visual Studio Code Editor	1.2.5 ✓
Visual Studio Editor	2.0.15 ↑

Last update Sep 2, 02:29

Visual Studio Code Editor

This package is installed as part of Engineering Feature.

Unity Technologies
Version 1.2.5 - February 09, 2022
Registry Unity
View documentation • View changelog • View licenses

Code editor integration for supporting Visual Studio Code as code editor for unity. Adds support for generating csproj files for intellisense purposes, auto discovery of installations, etc.

Dependencies

Is using	(None)
Used by	Engineering

Figure II.24: C# installation in VS Code

To create a new C# script:

- in Unity Editor we Go to the projects tab located at the lower half of the editor and we click on “Assets”.

- After selecting “Assets” we open the “Scripts” folder and right click inside the folder, inside the window that opens we hover on “Create” and we click on “C# Script”

The VS Code Studio IDE is automatically launched when we open a script from the Unity Editor.

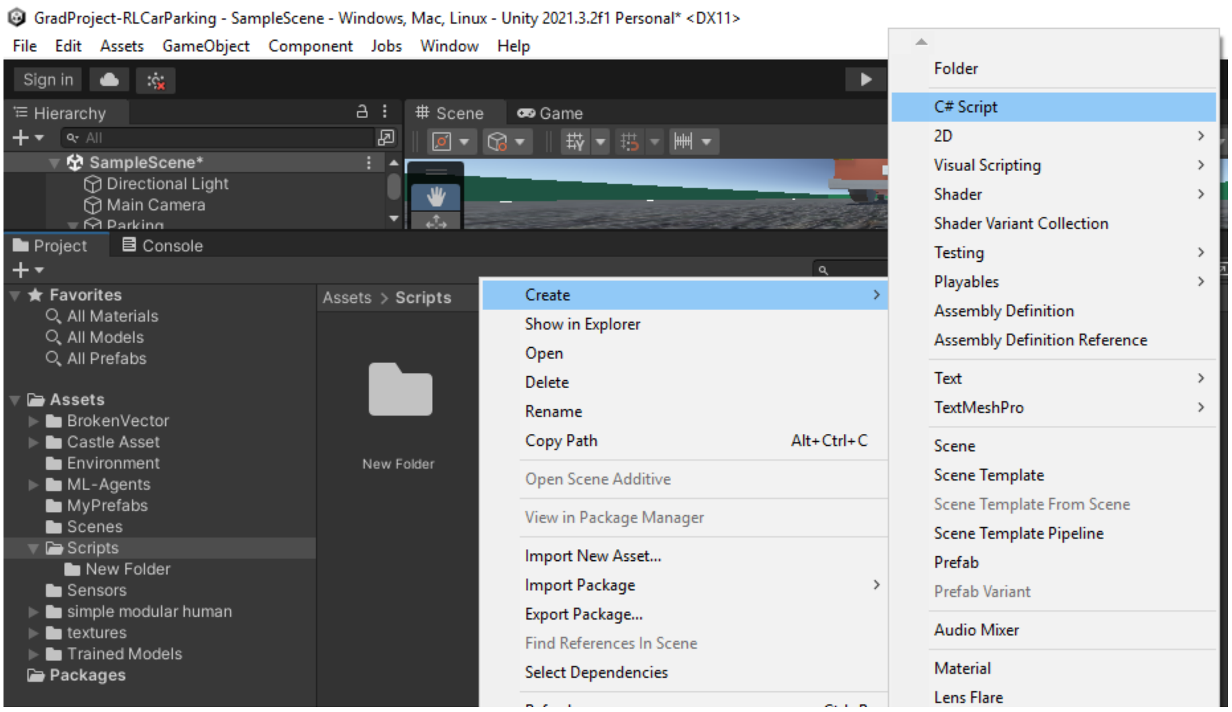


Figure II.25: Script creation in Unity Editor

<u>II.6.2 Environment and Agent Scripts</u>

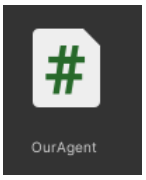
Each script has to be assigned to the appropriate GameObject in the editor since scripts in Unity determine how each object behaves. Due to this, we had to write separate scripts for the environment and the agent; the purpose of each function will be described in this sub-section.

The project includes two scripts:

- **AreaSpawn:** This script focuses on the environment; each time it runs, it resets the environment, giving the parked cars and the empty spot a random spawn.

- **OurAgent**: This is the main script for the agent, and it controls the beginning and finish of each episode as well as movement, observation, and reward functions. It also spawns the car at a random position in the middle of the parking lot.

  When this function is executed, it executes the previous function.



Both scripts uses the following namespaces, so it's critical that we define them.

- **System.Collections**: In C# the Collections namespace is used to store, alter, delete, search, and sort groups of data.
  Using System.Collections.Generic in our script we get access to the following classes of the Collections namespace:
  - List
  - Stack
  - Queue
  - LinkedList
  - HashSet
  - SortedSet
  - Dictionary
  - SortedDictionary
  - SortedList

- **UnityEngine:** This namespace allows the use of Unity elements in the script like GameObjects or positions of objects … etc

  Using UnityEngine allows us access to a lot of classes for example:
  - MonoBehaviour: Each Unity script must inherit from this base class; it offers multiple predefined functions.
  - GameObject: Type of objects in the scene.
  - Transform: Provides the use of Object positions, rotations and scales.

- **Unity.MLAgents:** Provides the essential MLAgents functions for learning for example:
  - CollectObservations.
  - OnActionReceived.
  - OnEpisodeBegin.

In the following sub-sections we highlight the purpose of each function from the scripts.

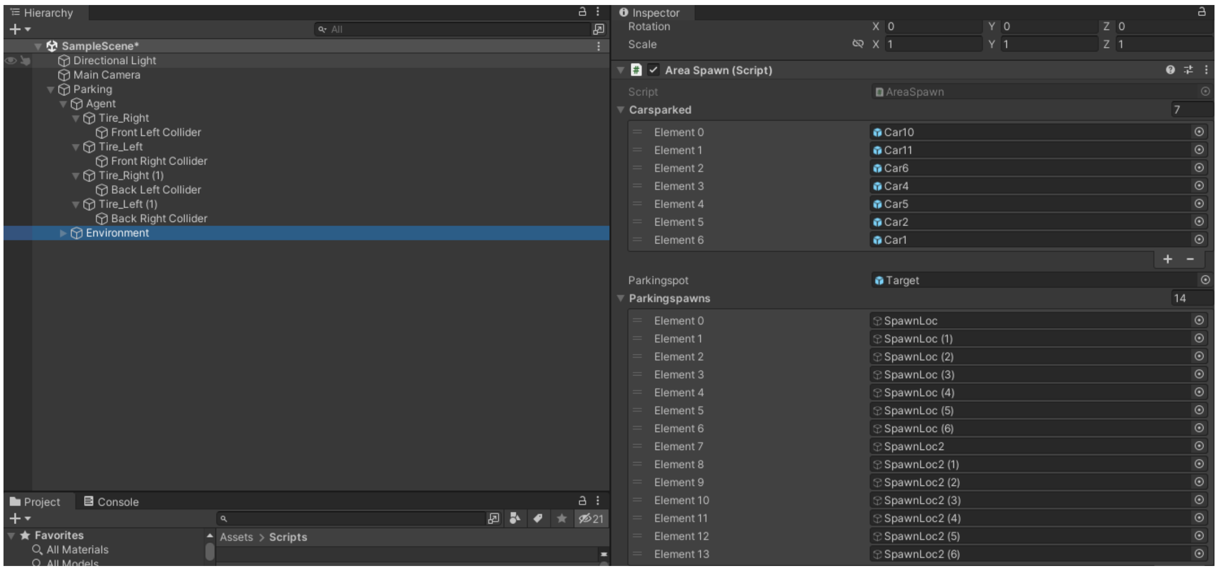
II.6.2.1 AreaSpawn Script

This script is for the environment, it controls the random spawn of the parked cars and the parking spot.

```
public List<GameObject> carsparked;  Represents the list of cars that will be parked in the environment.
public GameObject  parkingspot; This variable represents the parking spot that will be spawned in the vacant spot.
public List<GameObject> parkingspawns; The list of spawn locations we created in the environment.
```

After adding the script to the environment, these three variables are manually set in the Unity Editor.

We drag and drop each item into the right position as shown in **Figure II.26**.


```
Hierarchy
SampleScene*
  Directional Light
  Main Camera
  Parking
    Agent
      Tire_Right
        Front Left Collider
      Tire_Left
        Front Right Collider
      Tire_Right (1)
        Back Left Collider
      Tire_Left (1)
        Back Right Collider
    Environment

Project    Console
Favorites                 Assets > Scripts
  All Materials
  All Models
```

```
Inspector
Rotation      X 0    Y 0    Z 0
Scale         X 1    Y 1    Z 1
Area Spawn (Script)
Script            AreaSpawn
Carsparked                          7
  Element 0       Car10
  Element 1       Car11
  Element 2       Car6
  Element 3       Car4
  Element 4       Car5
  Element 5       Car2
  Element 6       Car1
Parkingspot       Target
Parkingspawns                       14
  Element 0       SpawnLoc
  Element 1       SpawnLoc (1)
  Element 2       SpawnLoc (2)
  Element 3       SpawnLoc (3)
  Element 4       SpawnLoc (4)
  Element 5       SpawnLoc (5)
  Element 6       SpawnLoc (6)
  Element 7       SpawnLoc2
  Element 8       SpawnLoc2 (1)
  Element 9       SpawnLoc2 (2)
  Element 10      SpawnLoc2 (3)
  Element 11      SpawnLoc2 (4)
  Element 12      SpawnLoc2 (5)
  Element 13      SpawnLoc2 (6)
```

Figure II.26: Attaching AreaSpawn script to environment

```
public Vector3 parkingspot_cords;
```

A variable used to obtain the location of the new parking spot after it spawns. It is of type Vector3 because the location is specified in the form of an x, y, and z coordinate. This variable is used in the agent script.

```
public void RandomPlaceEnvir(){

GameObject a=parkingspawns[Mathf.FloorToInt(UnityEngine.Random.Range(0,parkingspawns.Count))];
      foreach (GameObject obj in parkingspawns)
         {   if (obj == a)
                {placeObject(parkingspot,a);
                 parkingspot_cords = obj.transform.localPosition;}
```

```
        else
          { placeObject(carsparked[Mathf.FloorToInt(UnityEngine.Random.Range(0,carsparked.Count))],obj);}
    }}
```

- This function chooses a random spawn location from the list of parkingspawns.
- Following that it places the parking spot in the selected random spawn location and saves the location in the parkingspot_cords variable.
- The remaining spots are then filled by cars selected at random from the carsparked list.

The Flowchart shown in **Figure II.27** shows the execution process of the function.

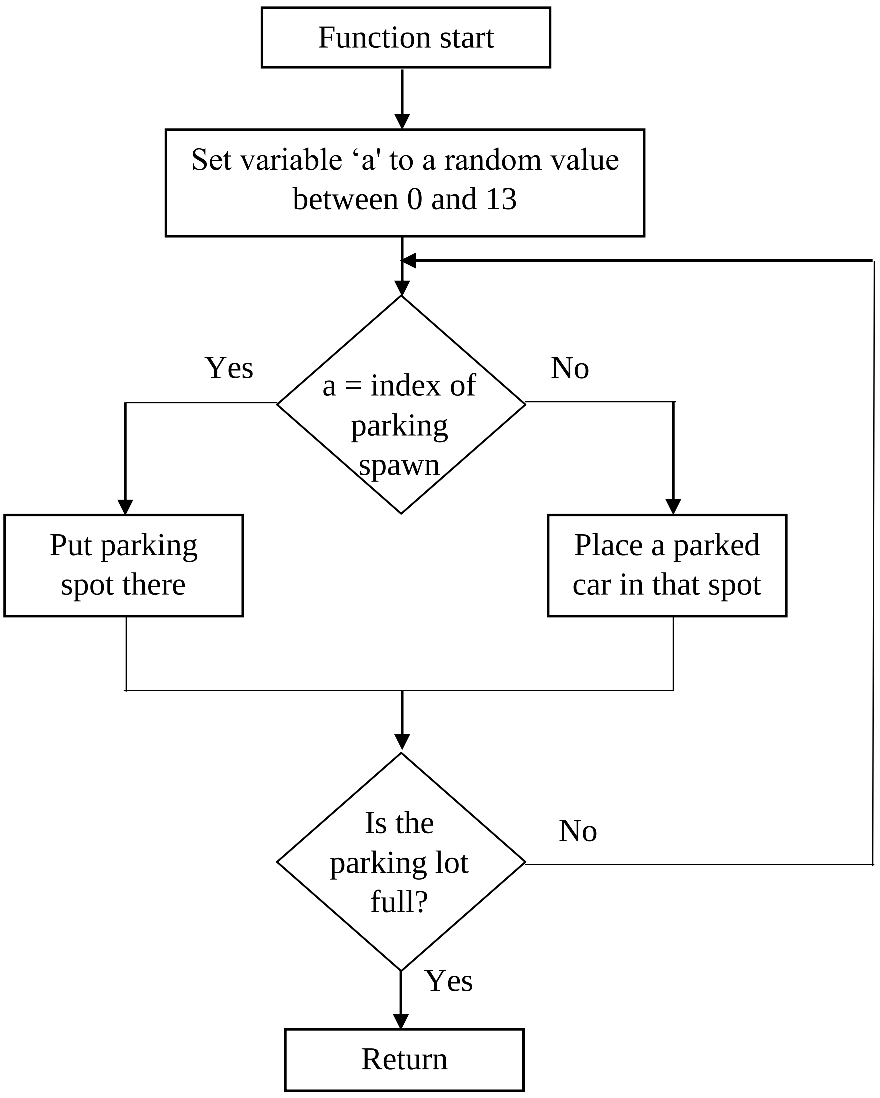

Function start
Set variable ‘a' to a random value between 0 and 13
a = index of parking spawn
Yes
No
Put parking spot there
Place a parked car in that spot
Is the parking lot full?
No
Yes
Return

Figure II.27: Flowchart explaining ‘RandomPlaceEnvir()’ execution process

```
public void resetparking()
  {
    Transform[] allChildren = GetComponentsInChildren<Transform>(true);
    foreach (Transform child in allChildren)
    {
      if (child.CompareTag("parkedcar") || child.CompareTag("parking") )
        Destroy(child.gameObject);
      }
    }
```

Every time it is called, this function obtains a list of all the GameObjects in our scene, searches for those that contain the tags "parkedcar" or "parking," and then destroys them.

We utilize this function to ensure that the previous spawning of parked cars and vacant parking spot are removed each time the environment is reset, preventing spawn overlap.

```
public void placeObject(GameObject parkedCarObject, GameObject gameObject)
  {
    Transform spawnLocation = gameObject.transform;
    Instantiate(parkedCarObject, spawnLocation);
  }
```

With the help of this function, we are able to place objects in the environment without permanently putting them in the scene.

It clones the original object and places it in the target position.

The RandomPlaceEnvir() function makes use of this function.

```
public void NewSpawn()
  {
    resetparking();
    RandomPlaceEnvir();
  }
```

When called, this simple function calls the RandomPlaceEnvir() and resetparking() functions in that sequence.

It destroys the previous environment and spawns a fresh one.

The Agent script makes use of this function.

```
private void Update()
  {
    if (Input.GetKeyDown(KeyCode.R))
    {
      NewSpawn();
    }}
```

We gave users the option to manually reset the environment by hitting the "R" key with the help of this function.

Each time the AreaSpawn executes it destroys the parked cars and parking spot and creates new ones placing them in an order different to the previous one as we notice from **Figure II.28.**

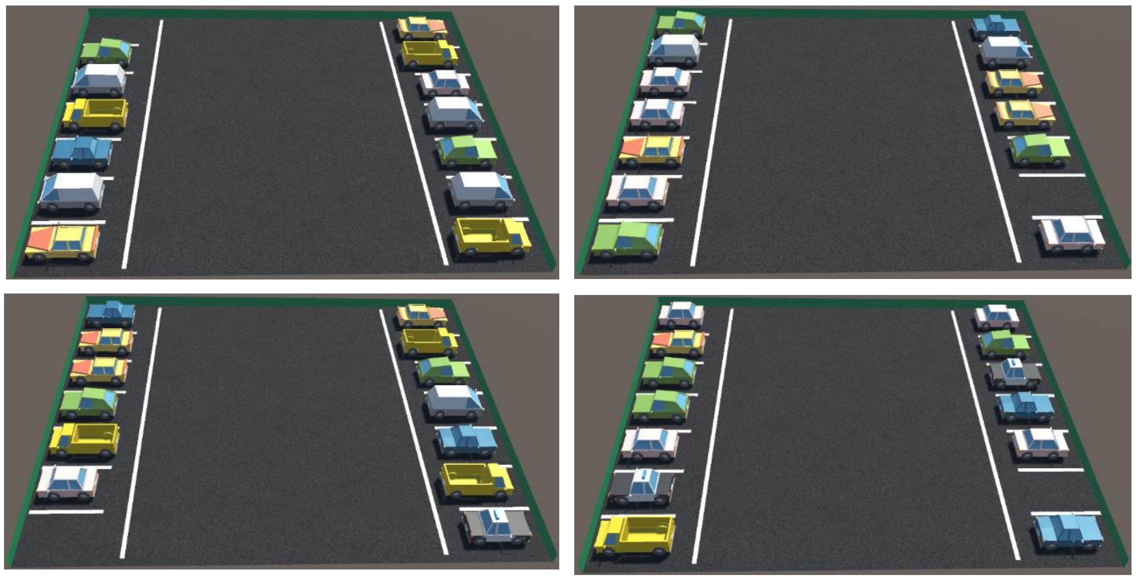


Figure II.28: AreaSpawn function example.

It should be noted that this script is executed through the Agent script.

II.6.2.2 OurAgent Script

This is the agent script, it controls everything from the spawn of the spawn of the car to the rewards it should receive and actions it can take.

```
public WheelCollider Back_right_collider ;
public WheelCollider Back_left_collider ;
public WheelCollider Front_right_collider ;
public WheelCollider Front_left_collider ;
```

These wheel colliders will be used later to manage car movements by exerting a force on and rotating them.

After attaching the script to the agent, we set these wheel colliders in Unity Editor.

Each wheel collider is moved into place by being dragged and dropped.

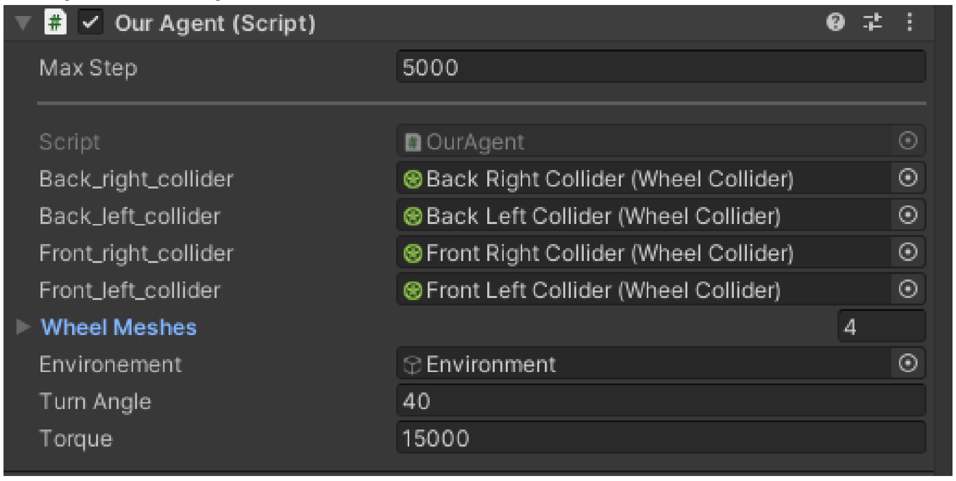
```
public GameObject Environement;
private AreaSpawn ParkingSettings;
```

In order for the agent script to access the AreaSpawn features, ParkingSettings is created as an instance to the AreaSpawn script.

ParkingSettings requires an environment in order to use functions from the AreaSpawn script on, thus we created a variable named Environement and set it in our Unity Editor to our environment.

```
private Vector3 orig; This variable is used to help randomize the agent's spawn location.
public float TurnAngle = 0f; TurnAngle is used to set the maximum turn angle of the wheels.
public float Torque = 0f; Torque is the force value applied on the wheel to accelerate it.
Rigidbody CarRb; This represents the car body in the script; it allows us to manipulate the car directly from the script.
```

The TurnAngle and Torque variables as set to 0 in the script as initial values, we can set their values directly from Unity Editor.


Our Agent (Script)

Max Step	5000
Script	OurAgent
Back_right_collider	Back Right Collider (Wheel Collider)
Back_left_collider	Back Left Collider (Wheel Collider)
Front_right_collider	Front Right Collider (Wheel Collider)
Front_left_collider	Front Left Collider (Wheel Collider)
Wheel Meshes	4
Environement	Environment
Turn Angle	40
Torque	15000

Figure II.29: Attaching OurAgent script to Agent

```
void Awake()
    {
        ParkingSettings = Environement.GetComponent<AreaSpawn>();
        CarRb = GetComponent<Rigidbody>();
    }
```

This function is the first to run when the script launches; it initializes the environment and the car agent so that following functions can operate on them.

```
public override void Initialize(){
        orig=new Vector3 (0,0,0);
        transform.localPosition=orig; }
```

We put the initial spawn position to the middle of the parking lot in order to be able to randomize the car's location.

```
private void RandomPlaceAgent(){

        float rx = UnityEngine.Random.Range((orig[0]-7f),(orig[0]+7f));
        float rz = UnityEngine.Random.Range((orig[2]-3f),(orig[2]+3f));

        CarRb.transform.localPosition=new Vector3 (rx,orig[1],rz);}
```

The new x axis position is assigned to a random value between -7 and 7 units (Which is the length of the parking lot), and the new z axis position is set to a random value between -3 and 3 units (which is the width of the parking lot but limited to the empty space in the middle).

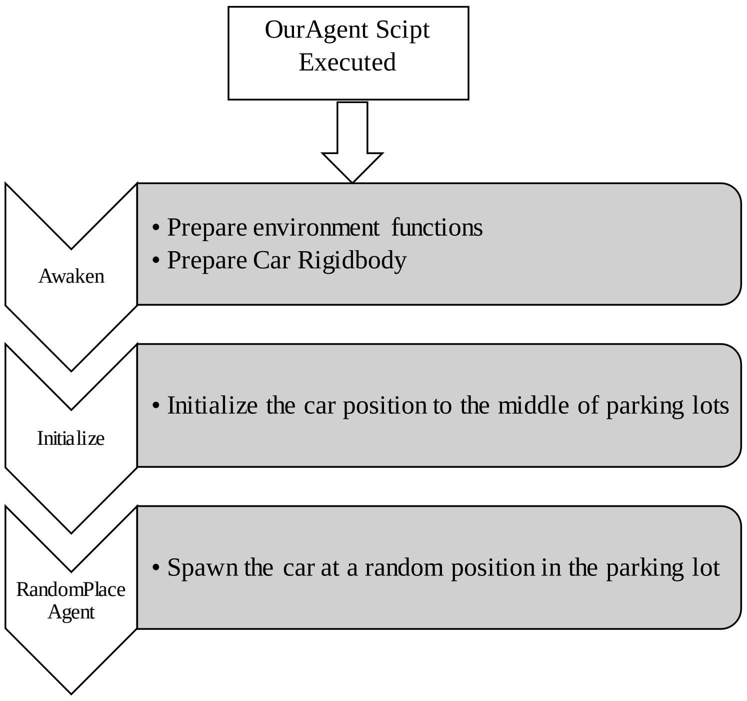

OurAgent Scipt Executed
Awaken
• Prepare environment functions
• Prepare Car Rigidbody
Initialize
• Initialize the car position to the middle of parking lots
RandomPlace Agent
• Spawn the car at a random position in the parking lot

Figure II.30: Execution process of functions that randomly spawn the agent.

That x and z coordinate is now the new Car spawn position.

As can be seen from **Figure II.31** each new episode the car spawns in a different position in the parking lot.

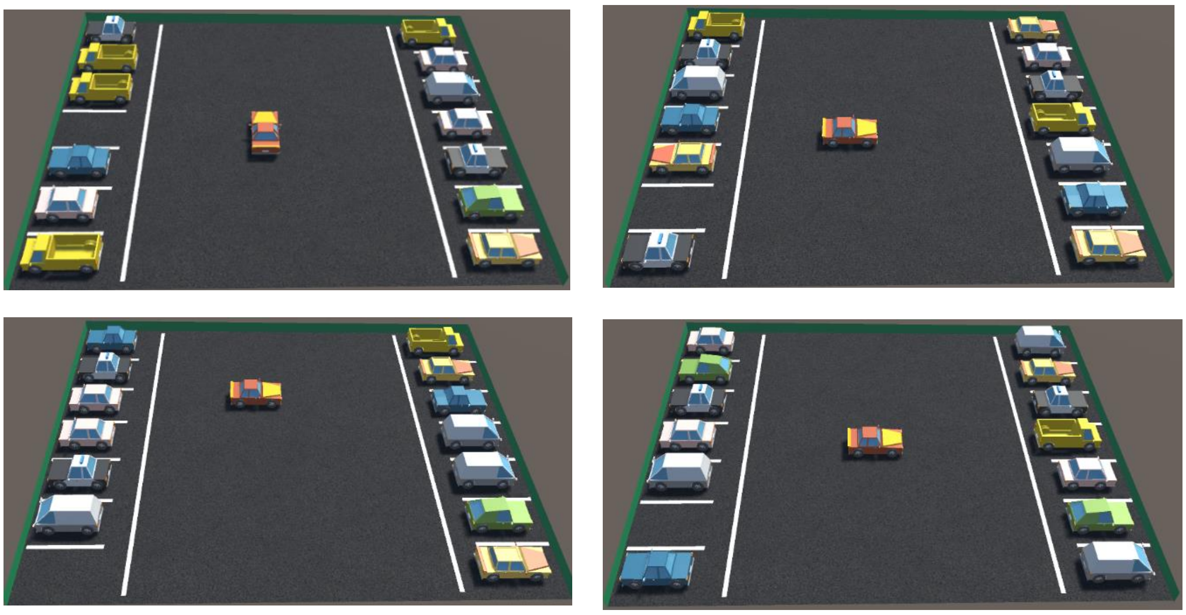


Figure II.31: Example of agent spawn.

```
private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.CompareTag("parkedcar"))
        {
            hitACar();
        }

        if (collision.gameObject.CompareTag("Wall"))
        {
            hitAWall();
        }
    }
    public void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("parking"))
        {
            parked();
        }
    }
```

The OnCollisionEnter method is called when an agent collides with an object in the environment. If the object hit has the tag "parkedcar," the hitACar() function is called; alternatively, the hitAWall() function is used if the object hit has the tag "Wall."

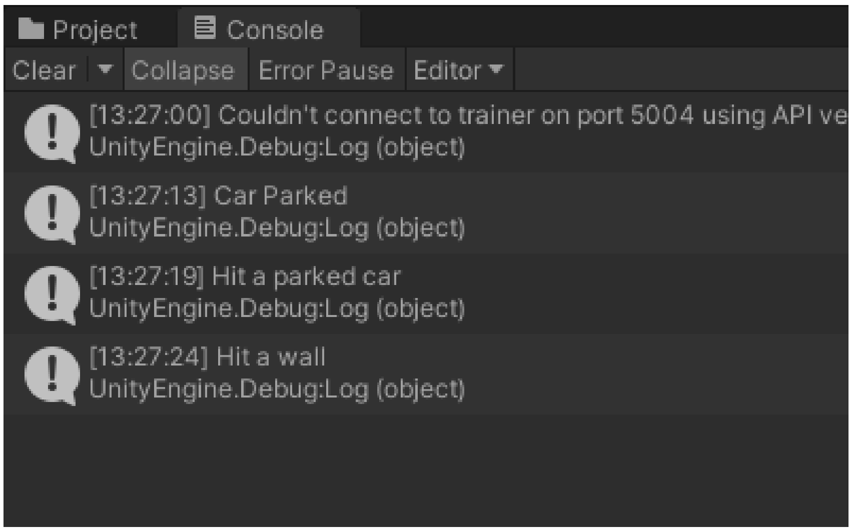
In a similar manner, whenever the car reaches the parking spot, the OnTriggerEnter method detects it and determines whether the tag of the spot is "parking," in which case the parked() function is called.

Rewards

```
public void hitACar()
    {Debug.Log("Hit a parked car");
        AddReward(-0.1f);
        EndEpisode();}


public void hitAWall()
    {
        Debug.Log("Hit a wall");
        AddReward(-0.1f);
        EndEpisode();
    }

public void parked(){

        Debug.Log("Car Parked");
        AddReward(3f);
        EndEpisode();}
```

Since rewards and punishments are the foundation of reinforcement learning (RL), they must be included in the script in order to prepare the agent for training using Unity ML-Agents.

Each time we enter one of the three functions listed above, a message letting us know which one was executed appears in the Unity Editor console.



```
[13:27:00] Couldn't connect to trainer on port 5004 using API ve
UnityEngine.Debug:Log (object)
[13:27:13] Car Parked
UnityEngine.Debug:Log (object)
[13:27:19] Hit a parked car
UnityEngine.Debug:Log (object)
[13:27:24] Hit a wall
UnityEngine.Debug:Log (object)
```

Figure II.32: Example of Rewards functions execution

The agent then receives either a negative or a positive response, and the episode then restarts.

Three different rewards were set up:

- The agent gets a -0.1 if it hits a parked car.
- The agent gets a -0.1 if it hits a wall.
- The agent gets a +3 if it reaches the parking spot.

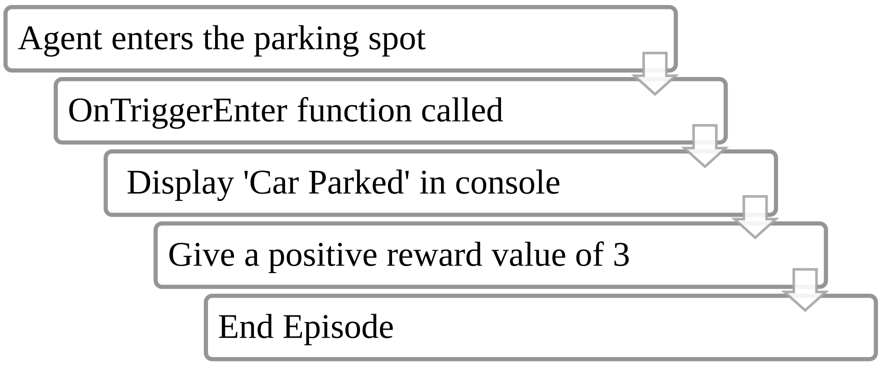

Agent enters the parking spot
OnTriggerEnter function called
Display 'Car Parked' in console
Give a positive reward value of 3
End Episode

Figure II.33: Execution process after agent parks successfully

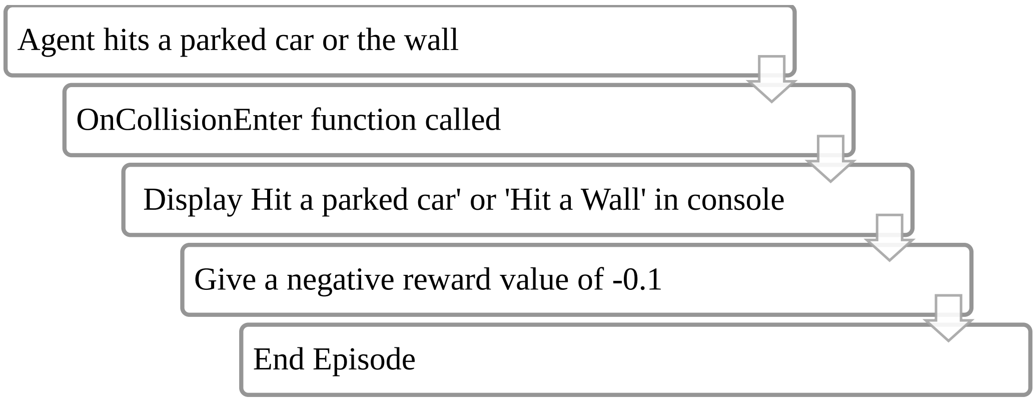

Agent hits a parked car or the wall
OnCollisionEnter function called
Display Hit a parked car' or 'Hit a Wall' in console
Give a negative reward value of -0.1
End Episode

Figure II.34: Execution process after agent collides with parked car or wall

Observations

There are two types of observations that our agent relies on:

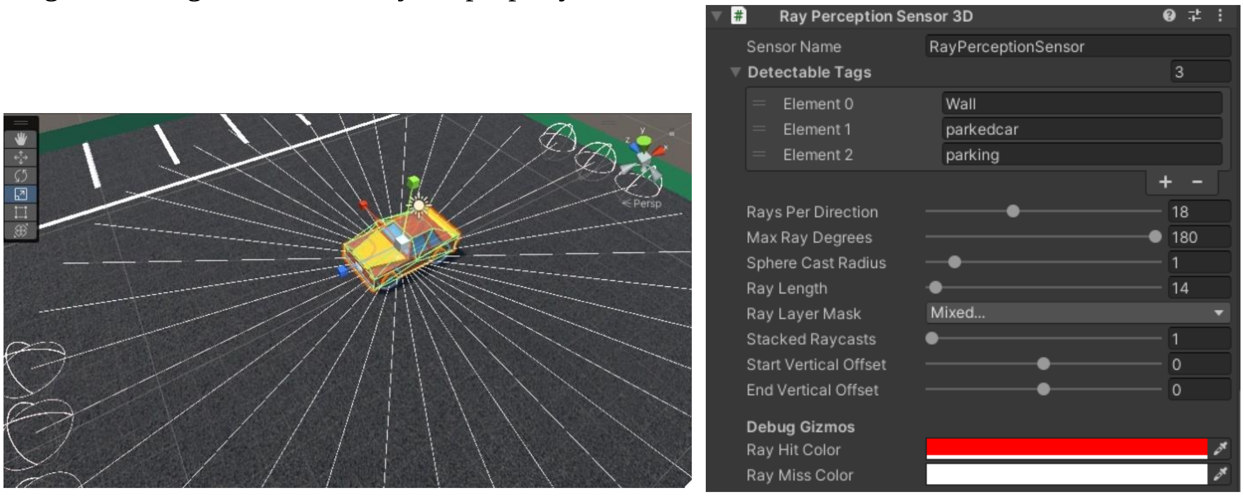
- Ray Perception Sensors
- CollectObservations Function

```
public override void  CollectObservations (VectorSensor sensor) {
        sensor.AddObservation(CarRb.velocity);
        sensor.AddObservation(Vector3.Distance(ParkingSettings.idk ,
transform.localPosition));
        sensor.AddObservation((ParkingSettings.idk -
transform.localPosition).normalized);
        sensor.AddObservation(transform.rotation); }
```

Each observation helps the agent in learning something new and supports navigation to its destination. Details of each observation are provided below.

- 1st Observation: Helps the agent pay attention to its movement speed.
- 2nd Observation: Know the distance between the agent and the parking spot.
- 3rd Observation: Know where the parking spot is located in the environment using x y z coordinates.
- 4th Observation: Helps the agent be aware of its rotation and which way it's facing.

For Ray Perception Sensors we set up the rays length, angle, spread and we add the detectable tags for the agent to detect objects properly.



Ray Perception Sensor 3D

Sensor Name	RayPerceptionSensor
Detectable Tags	3
Element 0	Wall
Element 1	parkedcar
Element 2	parking
Rays Per Direction	18
Max Ray Degrees	180
Sphere Cast Radius	1
Ray Length	14
Ray Layer Mask	Mixed...
Stacked Raycasts	1
Start Vertical Offset	0
End Vertical Offset	0
Debug Gizmos	
Ray Hit Color	🟥
Ray Miss Color	⬜

Figure II.35: Ray perception sensors 3D setup.

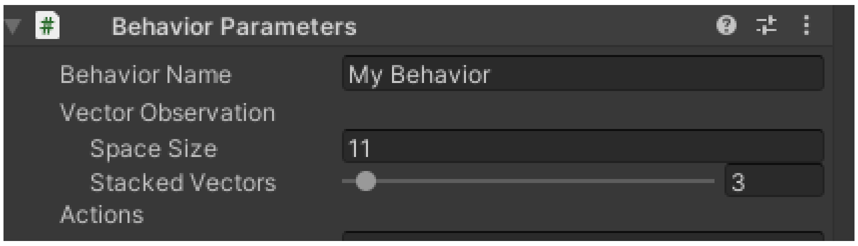
In the Behavior Parameters the stacked vectors value is set to 3 and the space size value is set to 11 because for each observation we have in the CollectObservations a specific observation space size is needed.

1st Observation: Takes 1 observation space for the X axis.

2nd Observation: Takes 3 observation spaces for the X, Y and Z axis.

3rd Observation: Takes 3 observation spaces for the X, Y and Z axis.

4th Observation: Takes 4 observation spaces for the X, Y, Z and W because it's a Quaternion.



Behavior Parameters

Behavior Name	My Behavior
Vector Observation	
Space Size	11
Stacked Vectors	3
Actions	

Figure II.36: Vector Observation setup.

Actions

The type of action the agent can take will have a significant impact on how the training will go.

Since driving involves a lot of movement changes and occasionally accelerating or reversing while turning, we chose Continuous actions to achieve more fluid movement.

```
public override void OnActionReceived(ActionBuffers actions)
    {
        var steer = actions.ContinuousActions[0];
        var throttle = actions.ContinuousActions[1];

        Back_right_collider.motorTorque=Torque * Time.fixedDeltaTime *
throttle;
        Back_left_collider.motorTorque=Torque * Time.fixedDeltaTime *
throttle;
        Front_right_collider.steerAngle=TurnAngle*steer;
        Front_left_collider.steerAngle=TurnAngle*steer;
        AddReward(-1f/ MaxStep);
    }
```

Continuous actions are explained in the previous chapter.

- The throttle variable will get a value from the AI that will be a float in the range of [-1,1] this value multiplied by the Torque variable we set will be applied as a torque force on the back wheel colliders to either move the car forward if the throttle value is positive or backwards if the value is negative.
- The acceleration will gradually increase as the time passes which is why we multiplied by the Time.fixedDeltaTime.
- The steer variable will also get a value from the AI in the form of a float ranging from -1 to 1 and this value multiplied by the TurnAngle variable will be applied as a steering angle to the front wheel colliders.
- A negative reward was added at the end of the function to prevent the agent from staying still and encourage moving and exploring the environment.
  This negative reward is given when the episode time runs out and the agent haven't collided with a parked car, a wall or found the parking spot.

The execution process of the OnActionReceived function is shown in **Figure II.37** below

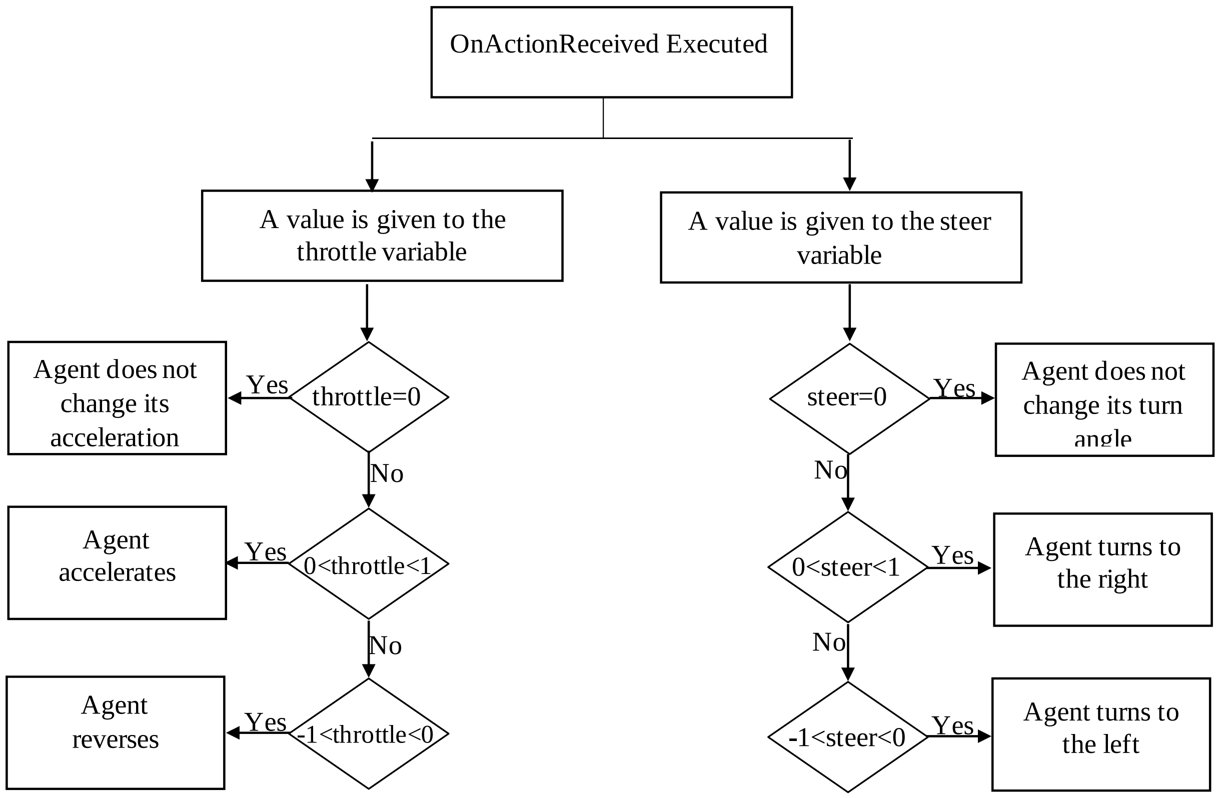
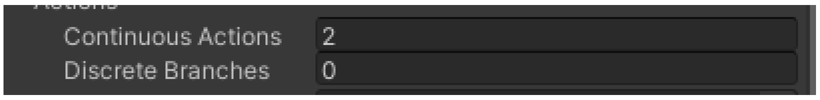

OnActionReceived Executed
A value is given to the throttle variable
A value is given to the steer variable
throttle=0
Yes
Agent does not change its acceleration
No
0<throttle<1
Yes
Agent accelerates
No
-1<throttle<0
Yes
Agent reverses
steer=0
Yes
Agent does not change its turn angle
No
0<steer<1
Yes
Agent turns to the right
No
-1<steer<0
Yes
Agent turns to the left

Figure II.37: OnActionReceived Execution Process

In the Behavior Parameters of the agent we set the number of continuous actions to 2 since we have 2 actions set up in the script.


Continuous Actions 2
Discrete Branches 0

When we are not trying to train the agent and just check the motion of the car, manual control is also an option using the heuristic function.

```
public override void Heuristic(in ActionBuffers actionsOut)
    {

        ActionSegment<float> continuousActionsOut =
actionsOut.ContinuousActions;
        float turn = Input.GetAxis("Horizontal");
        float move = Input.GetAxis("Vertical");
        continuousActionsOut[0] = turn;
        continuousActionsOut[1] = move;
    }
```

We simply control the car using Arrow Keys, with up being acceleration forward and down being accelerating backwards and the left and right keys for turning.

Episode Start

When a training episode starts the OnEpisodeBegin function is called.

```
public override void OnEpisodeBegin()
    {
        ParkingSettings.NewSpawn();
        RandomPlaceAgent();
        var rotationAngle = UnityEngine.Random.Range(0, 4) * 90f;
        transform.Rotate(new Vector3(0f,rotationAngle, 0f));

        CarRb.velocity = Vector3.zero;
        CarRb.angularVelocity = Vector3.zero;


    }
```

- First the environment respawns.
- After that the agent is placed in a random position with a random rotation in the parking lot.
- The car speed and turn angle are set to 0.

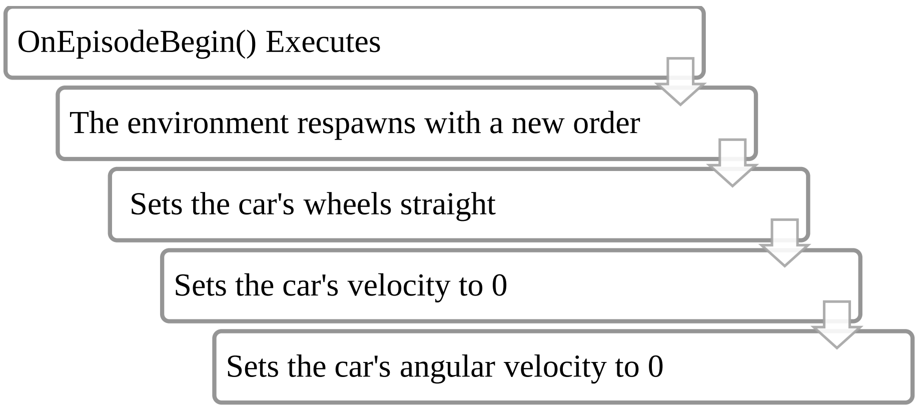
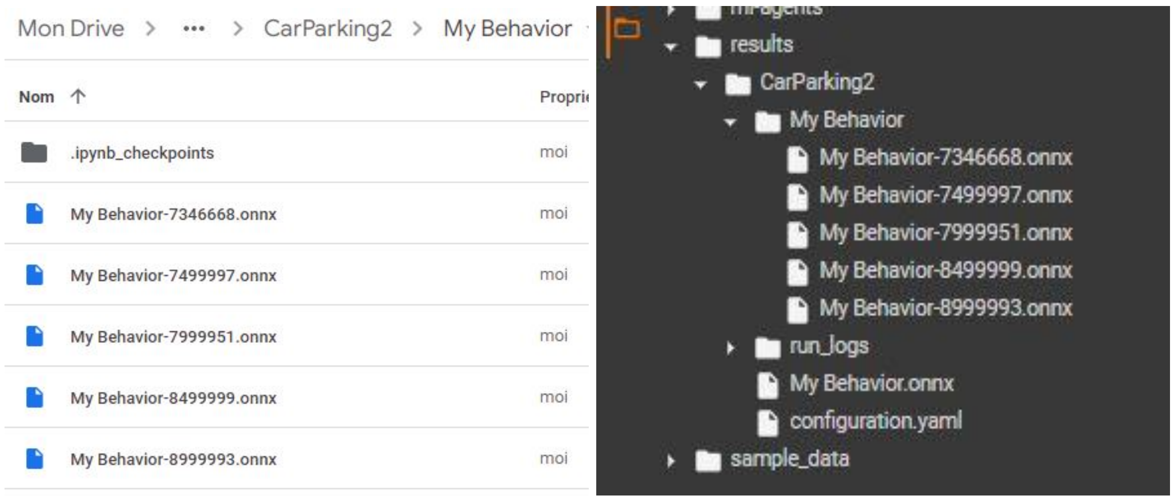

OnEpisodeBegin() Executes
The environment respawns with a new order
Sets the car's wheels straight
Sets the car's velocity to 0
Sets the car's angular velocity to 0

Figure II.38: Execution process of the OnEpisodeBegin( )function

II.7 Training Process

In order for the training of our Agent to be done fully in Colab our project had to be exported and uploaded into Google Drive so Colab can access it.

During training Google Colab saves multiple versions of the Neural Network model as it's being updated.



Mon Drive > ••• > CarParking2 > My Behavior

Nom ↑	Propri
.ipynb_checkpoints	moi
My Behavior-7346668.onnx	moi
My Behavior-7499997.onnx	moi
My Behavior-7999951.onnx	moi
My Behavior-8499999.onnx	moi
My Behavior-8999993.onnx	moi

```
▸ mlagents
▾ results
  ▾ CarParking2
    ▾ My Behavior
        My Behavior-7346668.onnx
        My Behavior-7499997.onnx
        My Behavior-7999951.onnx
        My Behavior-8499999.onnx
        My Behavior-8999993.onnx
    ▸ run_logs
      My Behavior.onnx
      configuration.yaml
▸ sample_data
```

Figure II.39: Example of how results are saved in Colab and uploaded to Drive

Colab is dependent on Google Drive to be able to:

1) Access Project Files for the training of the Agent.
2) To save the Neural Network models and the Graphs of the training.

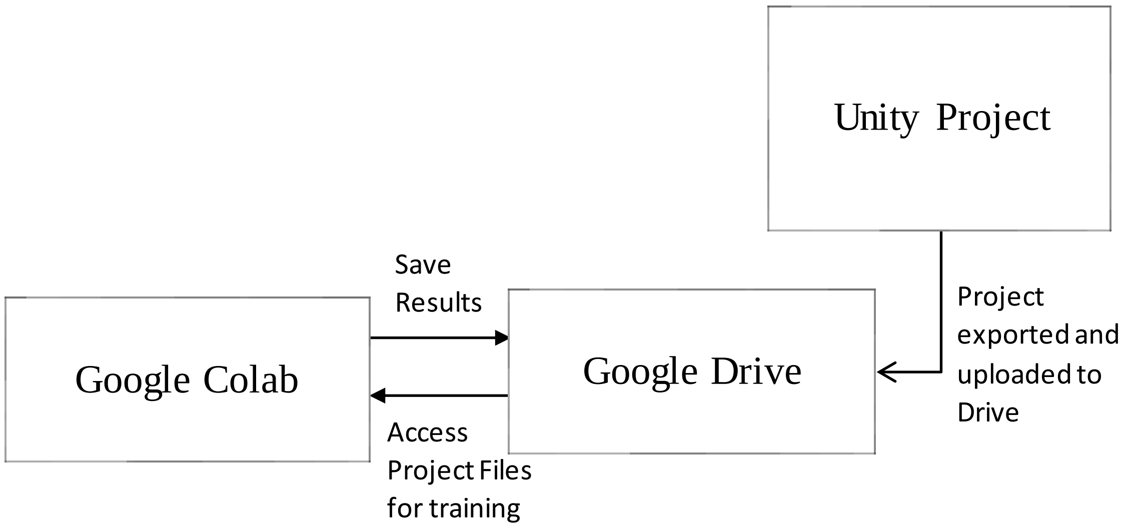
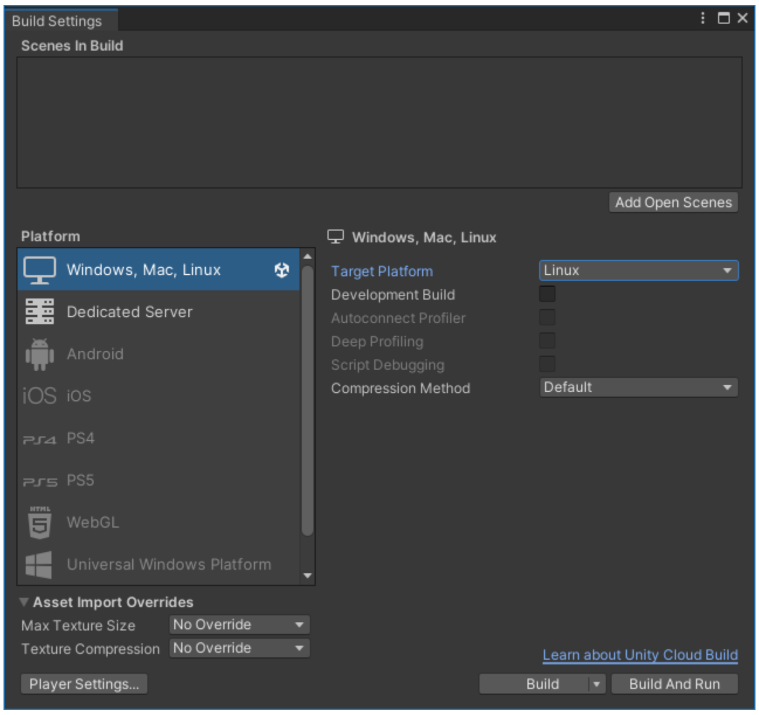

Unity Project
Save Results
Google Colab
Google Drive
Project exported and uploaded to Drive
Access Project Files for training

Figure II.40: Relation between Drive and Colab

To export the project:

- We go to File > Build Settings.
- Choose the target platform we want to build for. In our case since Colab runs Linux systems, we export our project for a Linux platform.



Windows, Mac, Linux
Target Platform: Linux
Development Build
Autoconnect Profiler
Deep Profiling
Script Debugging
Compression Method: Default

Max Texture Size: No Override
Texture Compression: No Override

Figure II.41: Exporting Project

Colab Setup

We create a new notebook in Colab and set up our training environment.

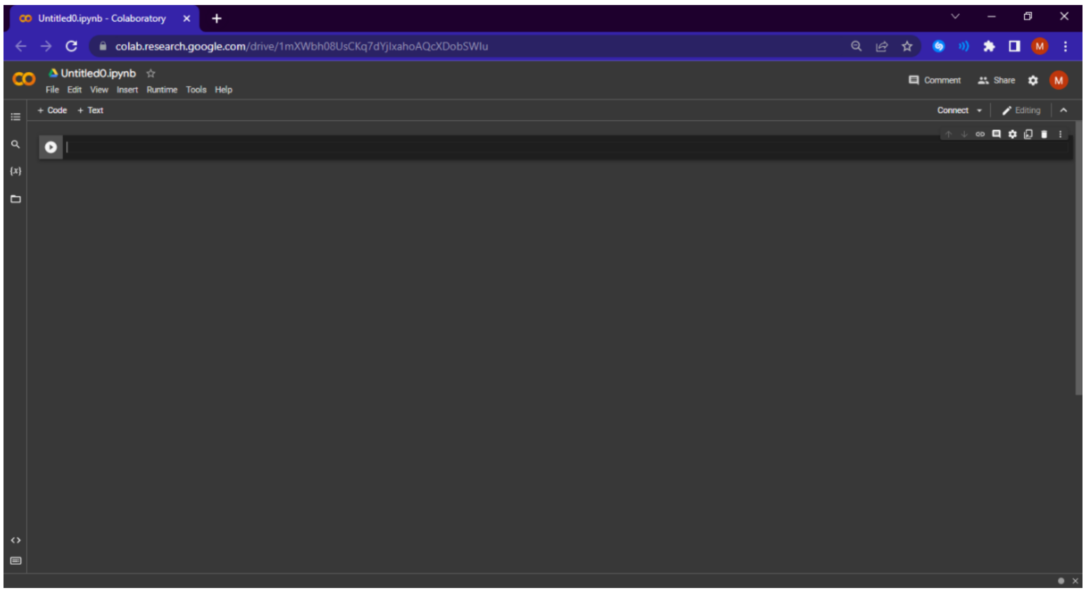
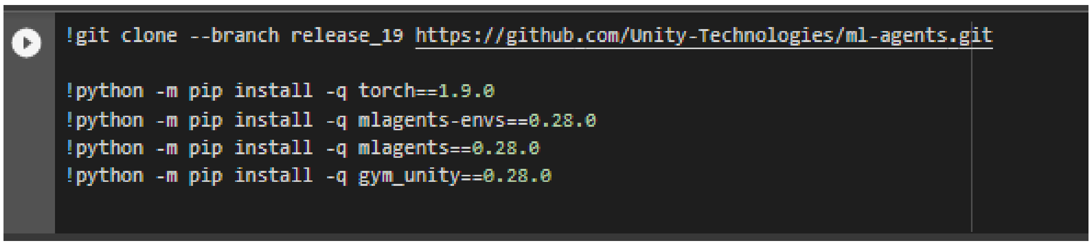
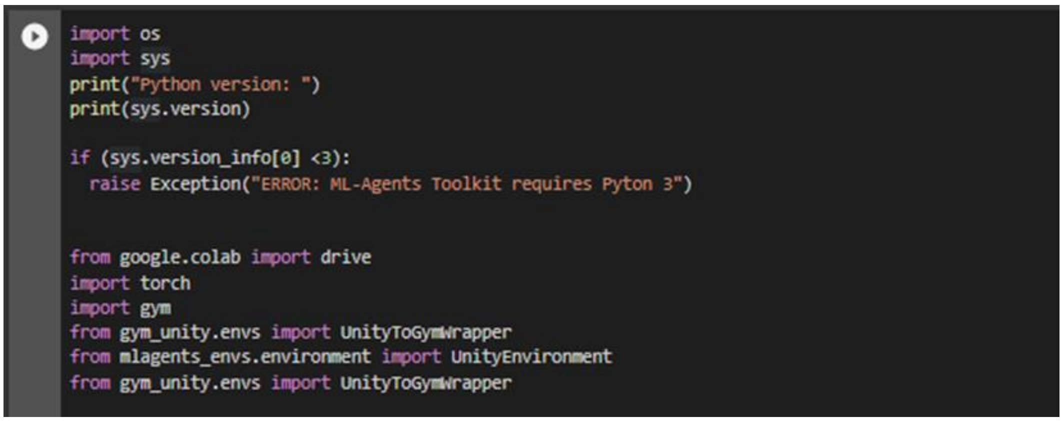

Untitled0.ipynb - Colaboratory
colab.research.google.com/drive/1mXWbh08UsCKq7dYjlxahoAQcXDobSWlu
Untitled0.ipynb
File Edit View Insert Runtime Tools Help
Comment
Share
+ Code
+ Text
Connect
Editing

Figure II.42: Example of Colab notebook

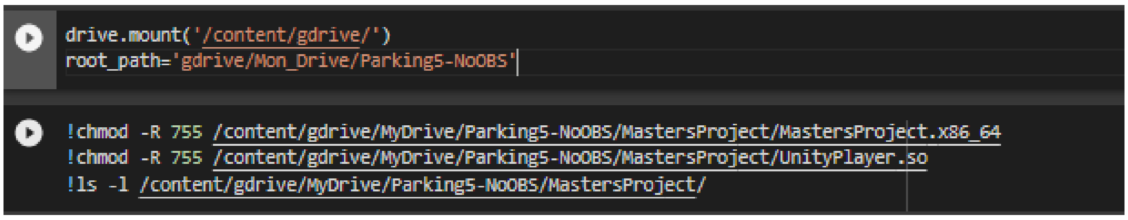
➢ We clone the ML-Agents Toolkit from the github then we install both ML-Agents and PyTorch.


```
!git clone --branch release_19 https://github.com/Unity-Technologies/ml-agents.git

!python -m pip install -q torch==1.9.0
!python -m pip install -q mlagents-envs==0.28.0
!python -m pip install -q mlagents==0.28.0
!python -m pip install -q gym_unity==0.28.0
```

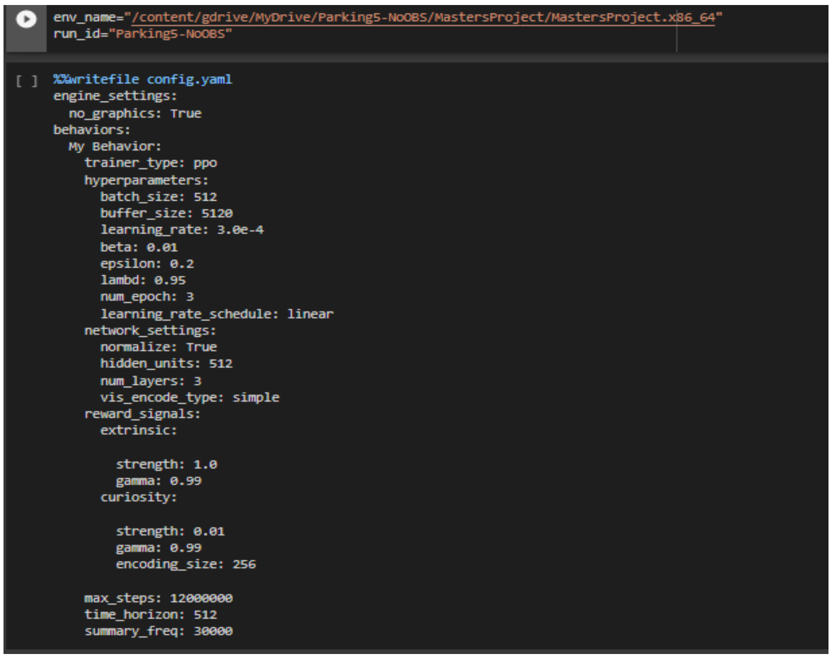
➢ Then we check the Python version and import all the important libraries.


```
import os
import sys
print("Python version: ")
print(sys.version)

if (sys.version_info[0] <3):
  raise Exception("ERROR: ML-Agents Toolkit requires Pyton 3")


from google.colab import drive
import torch
import gym
from gym_unity.envs import UnityToGymWrapper
from mlagents_envs.environment import UnityEnvironment
from gym_unity.envs import UnityToGymWrapper
```

➢ After that we mount our Google Drive so Colab could have access to our project build and we access those files.


```
drive.mount('/content/gdrive/')
root_path='gdrive/Mon_Drive/Parking5-NoOBS'
```
```
!chmod -R 755 /content/gdrive/MyDrive/Parking5-NoOBS/MastersProject/MastersProject.x86_64
!chmod -R 755 /content/gdrive/MyDrive/Parking5-NoOBS/MastersProject/UnityPlayer.so
!ls -l /content/gdrive/MyDrive/Parking5-NoOBS/MastersProject/
```

➢ Set our environment path and the name of run_id we wish to use and load our hyperparameter configuration (yaml configuration file.) It is important to mention that we had to use various configurations to find the optimal one for the training the agent. The shown configuration got us the best results.

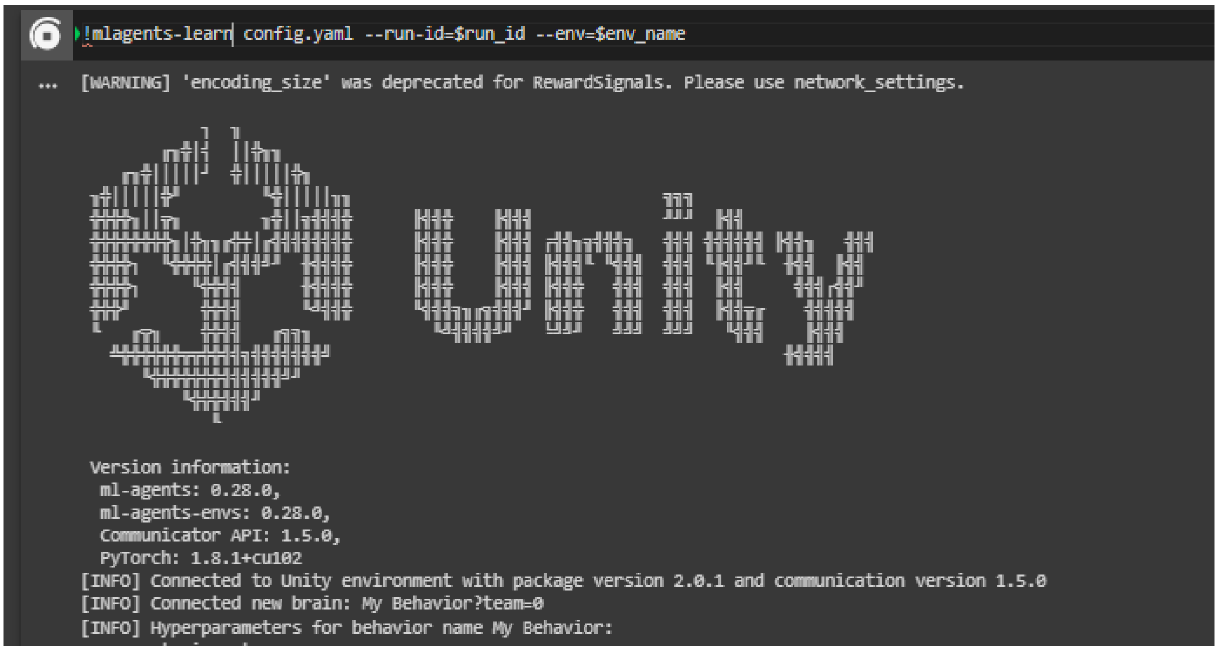


```
env_name="/content/gdrive/MyDrive/Parking5-NoOBS/MastersProject/MastersProject.x86_64"
run_id="Parking5-NoOBS"
```

```
[ ] %%writefile config.yaml
    engine_settings:
      no_graphics: True
    behaviors:
      My Behavior:
        trainer_type: ppo
        hyperparameters:
          batch_size: 512
          buffer_size: 5120
          learning_rate: 3.0e-4
          beta: 0.01
          epsilon: 0.2
          lambd: 0.95
          num_epoch: 3
          learning_rate_schedule: linear
        network_settings:
          normalize: True
          hidden_units: 512
          num_layers: 3
          vis_encode_type: simple
        reward_signals:
          extrinsic:

            strength: 1.0
            gamma: 0.99
          curiosity:

            strength: 0.01
            gamma: 0.99
            encoding_size: 256

        max_steps: 12000000
        time_horizon: 512
        summary_freq: 30000
```

Figure II.43: Loading up Hyperparameter config in colab.

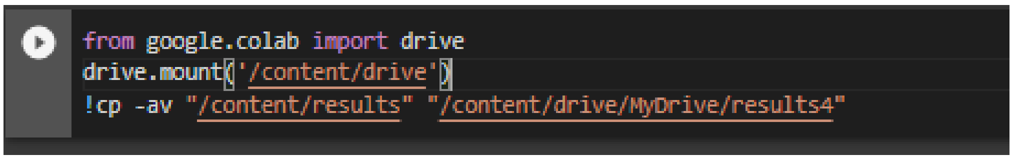
➢ And finally start the training process.



```
!mlagents-learn config.yaml --run-id=$run_id --env=$env_name

... [WARNING] 'encoding_size' was deprecated for RewardSignals. Please use network_settings.

 Version information:
  ml-agents: 0.28.0,
  ml-agents-envs: 0.28.0,
  Communicator API: 1.5.0,
  PyTorch: 1.8.1+cu102
[INFO] Connected to Unity environment with package version 2.0.1 and communication version 1.5.0
[INFO] Connected new brain: My Behavior?team=0
[INFO] Hyperparameters for behavior name My Behavior:
```

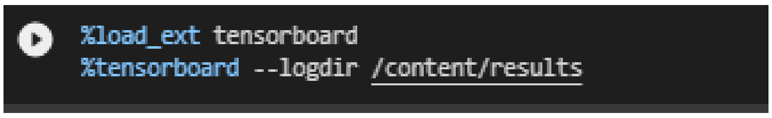
Figure II.44: Training start.

Anytime we need, we can pause the training process and execute the following command to store the training results to Google Drive.



```
from google.colab import drive
drive.mount('/content/drive')
!cp -av "/content/results" "/content/drive/MyDrive/results4"
```

We are also able to obtain graphs showing the full progress of the agent training using tensorboard:



```
%load_ext tensorboard
%tensorboard --logdir /content/results
```

The figure below shows the entire process of Training in Colab

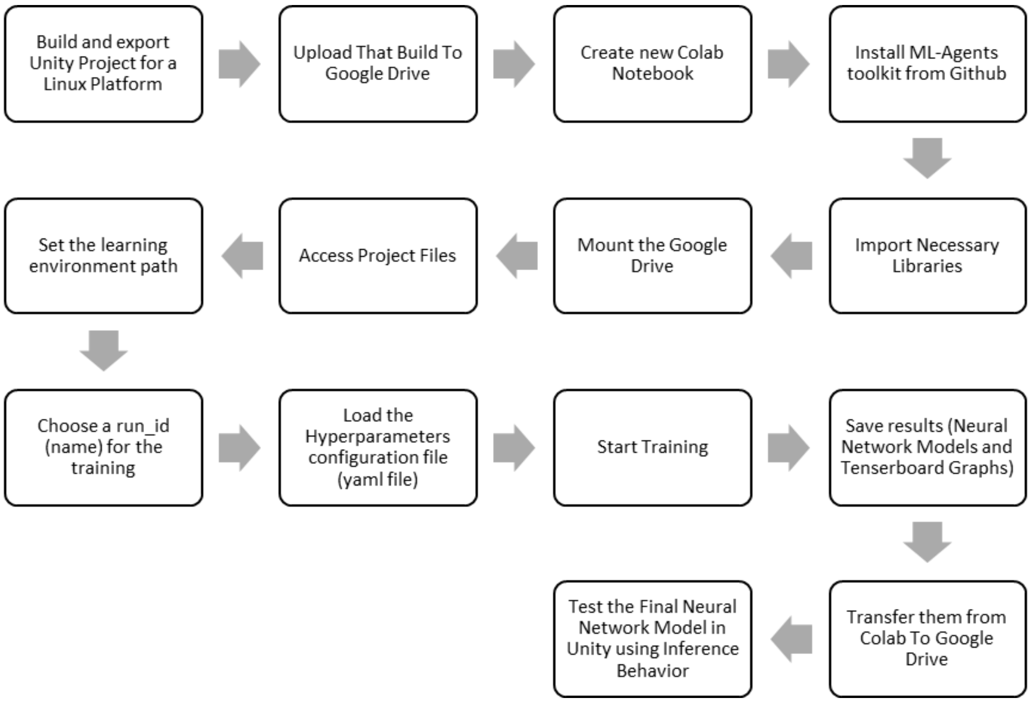

Build and export Unity Project for a Linux Platform
Upload That Build To Google Drive
Create new Colab Notebook
Install ML-Agents toolkit from Github
Import Necessary Libraries
Mount the Google Drive
Access Project Files
Set the learning environment path
Choose a run_id (name) for the training
Load the Hyperparameters configuration file (yaml file)
Start Training
Save results (Neural Network Models and Tenserboard Graphs)
Transfer them from Colab To Google Drive
Test the Final Neural Network Model in Unity using Inference Behavior

Figure II.45: Training Process Using Colab

II.8 Conclusion

In this chapter, we went into great detail about how we created the environment and the agent using Unity and ML-Agents, and we also covered how to set up Google Colab so that our agent could be trained directly on the cloud.

The implementation results are analyzed in the next chapter.

Chapter III: Results

III.1 Introduction

In this chapter we'll be discussing the final results of the training.

Tensorboard graphs and statistics collected throughout the agent's training were used to evaluate the training's results.

Lastly, the trained model will be tested using the inference behavior.

III.2 Tracking progress

For the training we had to modify a couple of hyperparameters in order to get better result:

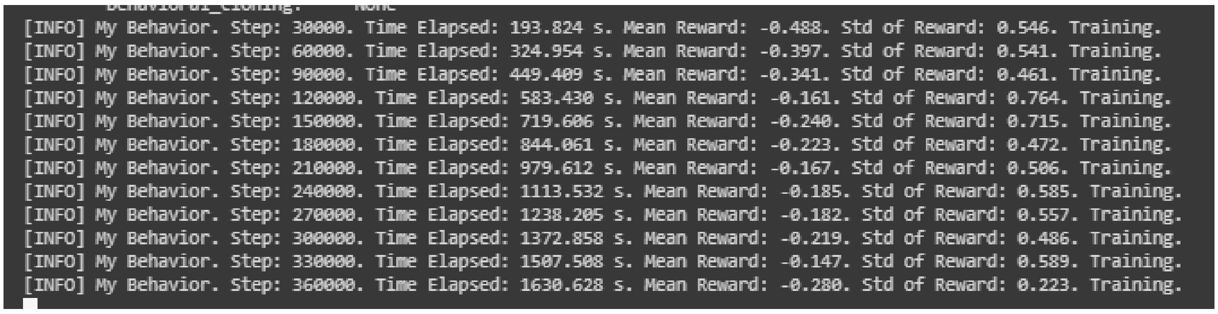
We increased the **hidden_units** parameter to 512, and we changed **num_layer** parameter to 3; this increases the performance of the hidden layer in the neural network.

We also enabled **Curiosity signal** to help the agent explore the environment more.

Finally, we set the **max_steps** parameter to 12000000 to train the agent for 12 million steps.

The progress of the training can be tracked using the generated training statistics; these stats are set by the **summary_freq** hyperparameter to give us updates each 30000 steps.

Figure III.1 shows the statistics from the beginning of the training:



```
[INFO] My Behavior. Step: 30000. Time Elapsed: 193.824 s. Mean Reward: -0.488. Std of Reward: 0.546. Training.
[INFO] My Behavior. Step: 60000. Time Elapsed: 324.954 s. Mean Reward: -0.397. Std of Reward: 0.541. Training.
[INFO] My Behavior. Step: 90000. Time Elapsed: 449.409 s. Mean Reward: -0.341. Std of Reward: 0.461. Training.
[INFO] My Behavior. Step: 120000. Time Elapsed: 583.430 s. Mean Reward: -0.161. Std of Reward: 0.764. Training.
[INFO] My Behavior. Step: 150000. Time Elapsed: 719.606 s. Mean Reward: -0.240. Std of Reward: 0.715. Training.
[INFO] My Behavior. Step: 180000. Time Elapsed: 844.061 s. Mean Reward: -0.223. Std of Reward: 0.472. Training.
[INFO] My Behavior. Step: 210000. Time Elapsed: 979.612 s. Mean Reward: -0.167. Std of Reward: 0.506. Training.
[INFO] My Behavior. Step: 240000. Time Elapsed: 1113.532 s. Mean Reward: -0.185. Std of Reward: 0.585. Training.
[INFO] My Behavior. Step: 270000. Time Elapsed: 1238.205 s. Mean Reward: -0.182. Std of Reward: 0.557. Training.
[INFO] My Behavior. Step: 300000. Time Elapsed: 1372.858 s. Mean Reward: -0.219. Std of Reward: 0.486. Training.
[INFO] My Behavior. Step: 330000. Time Elapsed: 1507.508 s. Mean Reward: -0.147. Std of Reward: 0.589. Training.
[INFO] My Behavior. Step: 360000. Time Elapsed: 1630.628 s. Mean Reward: -0.280. Std of Reward: 0.223. Training.
```

Figure III.1: statistics from the beginning of the training

These statistics show a variety of different data including:

the name of the Agent's Behavior currently in training.

Step: How many steps passed since the beginning of the training.

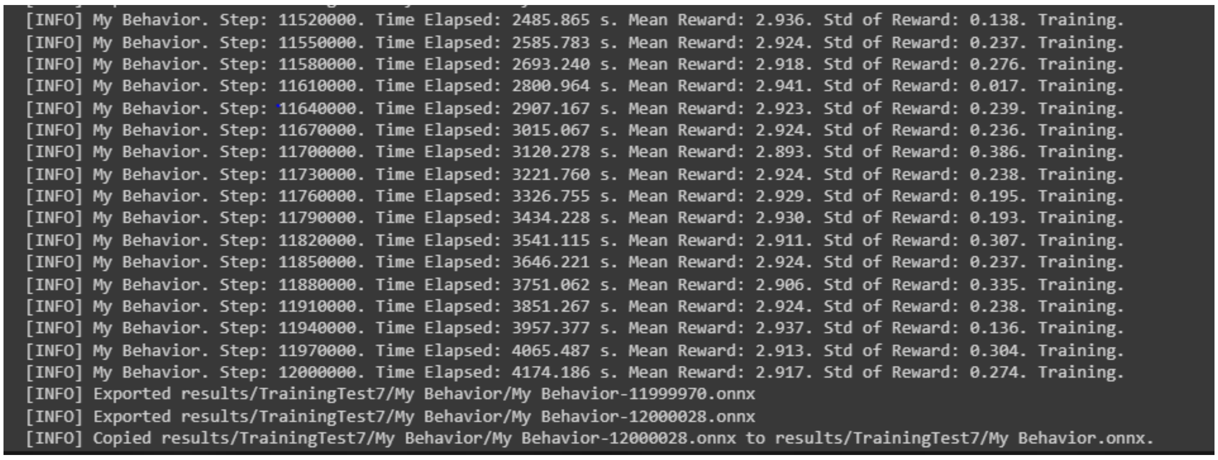
Time Elapsed: Time elapsed since the beginning of the training.

Mean reward: the cumulative number of rewards gotten across all learning environments, a higher value means a better policy and a better result.

Std of Reward: it represents the variations in Mean Rewards, a higher value means a bigger diversity in the type of rewards obtained.

III.3 Final Results

The final training steps are represented in **Figure III.2**; The training took approximately 12 hours to finish 12 million steps.



```
[INFO] My Behavior. Step: 11520000. Time Elapsed: 2485.865 s. Mean Reward: 2.936. Std of Reward: 0.138. Training.
[INFO] My Behavior. Step: 11550000. Time Elapsed: 2585.783 s. Mean Reward: 2.924. Std of Reward: 0.237. Training.
[INFO] My Behavior. Step: 11580000. Time Elapsed: 2693.240 s. Mean Reward: 2.918. Std of Reward: 0.276. Training.
[INFO] My Behavior. Step: 11610000. Time Elapsed: 2800.964 s. Mean Reward: 2.941. Std of Reward: 0.017. Training.
[INFO] My Behavior. Step: 11640000. Time Elapsed: 2907.167 s. Mean Reward: 2.923. Std of Reward: 0.239. Training.
[INFO] My Behavior. Step: 11670000. Time Elapsed: 3015.067 s. Mean Reward: 2.924. Std of Reward: 0.236. Training.
[INFO] My Behavior. Step: 11700000. Time Elapsed: 3120.278 s. Mean Reward: 2.893. Std of Reward: 0.386. Training.
[INFO] My Behavior. Step: 11730000. Time Elapsed: 3221.760 s. Mean Reward: 2.924. Std of Reward: 0.238. Training.
[INFO] My Behavior. Step: 11760000. Time Elapsed: 3326.755 s. Mean Reward: 2.929. Std of Reward: 0.195. Training.
[INFO] My Behavior. Step: 11790000. Time Elapsed: 3434.228 s. Mean Reward: 2.930. Std of Reward: 0.193. Training.
[INFO] My Behavior. Step: 11820000. Time Elapsed: 3541.115 s. Mean Reward: 2.911. Std of Reward: 0.307. Training.
[INFO] My Behavior. Step: 11850000. Time Elapsed: 3646.221 s. Mean Reward: 2.924. Std of Reward: 0.237. Training.
[INFO] My Behavior. Step: 11880000. Time Elapsed: 3751.062 s. Mean Reward: 2.906. Std of Reward: 0.335. Training.
[INFO] My Behavior. Step: 11910000. Time Elapsed: 3851.267 s. Mean Reward: 2.924. Std of Reward: 0.238. Training.
[INFO] My Behavior. Step: 11940000. Time Elapsed: 3957.377 s. Mean Reward: 2.937. Std of Reward: 0.136. Training.
[INFO] My Behavior. Step: 11970000. Time Elapsed: 4065.487 s. Mean Reward: 2.913. Std of Reward: 0.304. Training.
[INFO] My Behavior. Step: 12000000. Time Elapsed: 4174.186 s. Mean Reward: 2.917. Std of Reward: 0.274. Training.
[INFO] Exported results/TrainingTest7/My Behavior/My Behavior-11999970.onnx
[INFO] Exported results/TrainingTest7/My Behavior/My Behavior-12000028.onnx
[INFO] Copied results/TrainingTest7/My Behavior/My Behavior-12000028.onnx to results/TrainingTest7/My Behavior.onnx.
```

Figure III.2: statistics from the end of the training

We can observe that the agent's mean reward value stabilized at around 2.9. Since the highest reward value that could be obtained in our learning environment was set at a value of 3, using the following equation we can find the success rate:

$$\frac{\boldsymbol{Average\ Mean\ Reward\ value}}{\mathbf{3}} * \mathbf{100} = \frac{\mathbf{2.92}}{\mathbf{3}} * \mathbf{100} = \mathbf{97\%}$$

III.3.1 Graph Analysis

The Environment/Cumulative Reward graph shows the progress of the value of cumulative mean reward obtained throughout the training process.

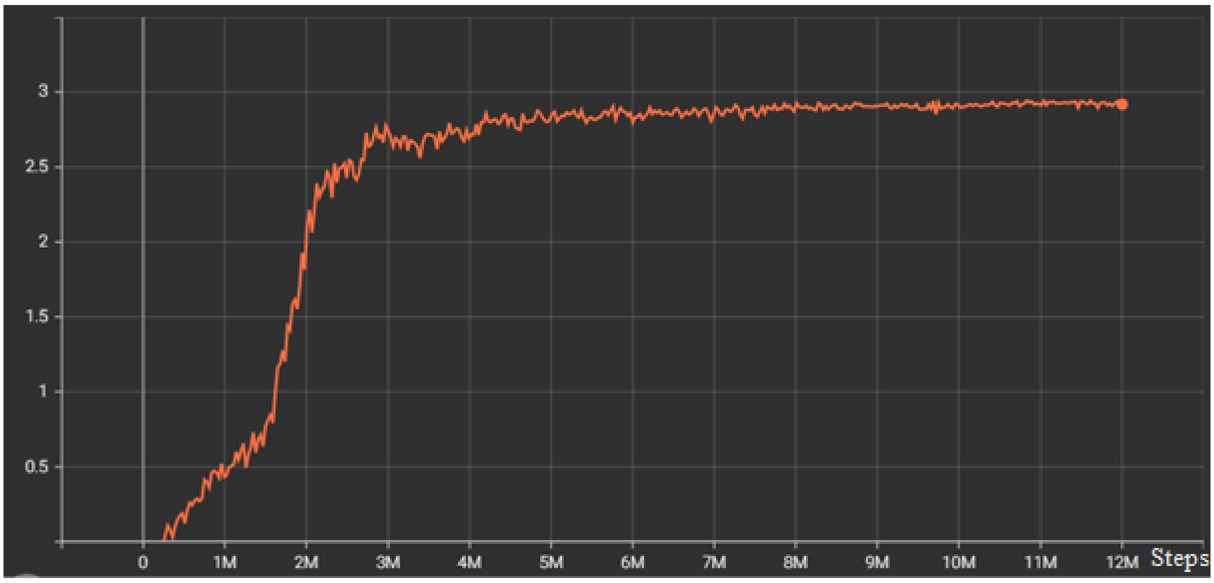


Figure III.3: Environment/Cumulative Reward graph

The graph shows that at the beginning of training, the cumulative reward value was low and rising gradually, reaching a value of 0.5 at 1 million steps. The value continued to rise gradually until reaching 0.75 at 1.5 million steps, at which point it started to improve quickly, reaching a value of 2.5 at 2 million steps. Following that, the progress slowed down and the cumulative mean reward value began to rise gradually once again peaking at 2.9 at 8 million steps, where it stabilized for the rest of the training.

The Environment/Episode Length graph shows the progress of the length of episodes.

The length of the episode is measured in the amount of times the RequestDecision() function was called before the EndEpisode() function is called.

In our case the DecisionRequester component calls the RequestDecision() function every 5 steps.

The longer the episode is, the more the RequestDecision() function is called.

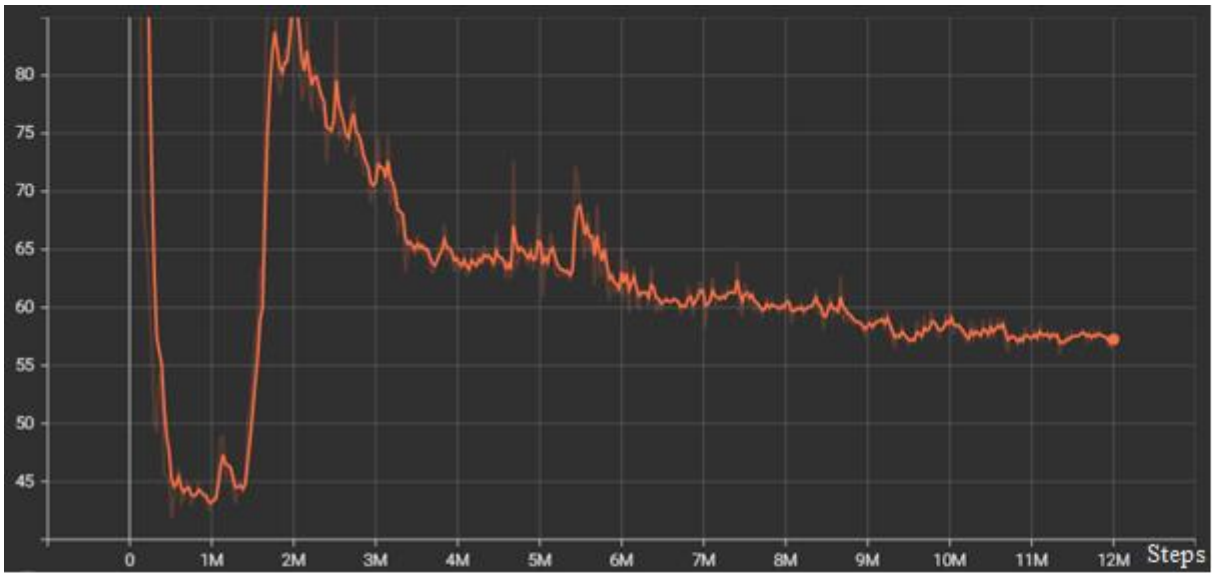


Figure III.4: Environment/Episode length

According to the above graph, the episodes were lengthy at first before decreasing to a value of 45 at 500000 steps, where it stabilized for a while before increasing again to reach 85 at 1.3 million steps. From there, it began to gradually decrease until reaching a value of around 57, where it then stabilized for the remainder of the training process.

The Policy/Entropy graph indicates the randomness of the algorithm's decisions during the training process.

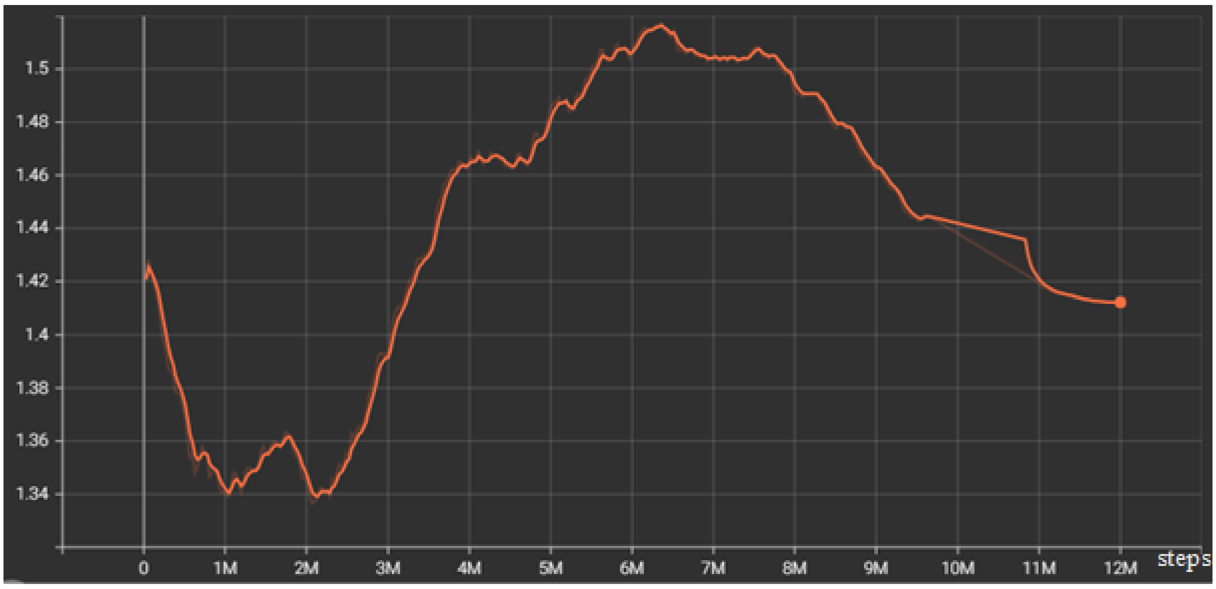


Figure III.5: Policy/Entropy graph

As can be seen from the graph, the entropy value starts at 1.42 and declines to 1.34 after 1 million steps. It then quickly rises again after 2 million steps, reaching 1.51 at 6.1 million steps where it briefly holds, before slowly declining until the end of the training, where it finishes at a value of 1.41.

III.3.2 Results discussion

The agent does not have a defined policy to follow at the start of the training, so it begins acting randomly and exploring the actions it can take. As a result, the agent does not complete the episode and receives a negative reward of -1.

After exploring the actions, the agent starts exploring the environment getting different rewards and slowly developing a policy, it discovers that it is better to collide with a wall or a parked car and receive a negative reward of -0.1 rather than not hitting anything and getting a -1 for reaching max step and not finishing the episode.

According to **Figure III.4**, this leads to a decrease in episode length. It also reduces entropy, as shown in **Figure III.5**, and as a result, the mean rewards value begins to increase.

As the agent follows with its current policy and learns that entering the parking space can result in a positive reward with a value of 3, it begins to explore the environment once more, which causes the entropy in the policy to increase and the episode length to increase rapidly.

The massive increase in cumulative mean rewards shows that the policy greatly improves after this.

As can be observed from **Figure III.3**, the agent reaches the highest cumulative mean reward value that is achievable and stabilizes there until the training is over, proving that it has found the optimal policy available.

This also results in a stabilization of episode length, as shown in **Figure III.4**, and a reduction in policy entropy, as shown by the **Figure III.5** graph.

During the development of our project, we encountered three major problems that we felt needed to be mentioned:

Hyperparameters: The training hyperparameters were challenging to tune; we had to test a number of configurations that produced bad models before finding the one that worked best and was used in our final model.

We had to increase the hidden_units to 512 and the num_layer to 3 and we had to also add the curiosity signal.

Training Time: To complete 10 million steps, training times averaged 10 hours and occasionally even exceeded 12 hours. Because we had to repeat training sessions in order to adjust the hyperparameters, the total training time exceeded 100 hours.

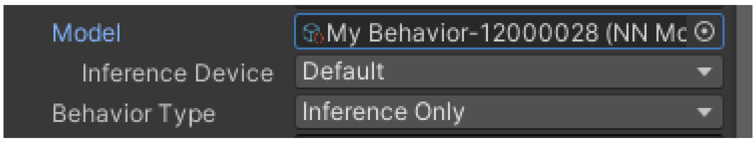
For instance, it took our final model 12 hours to complete 12 million steps.

Action optimization: Since we penalized the agent for taking too long to complete training episodes, it learnt to optimize its movement in order to maximize rewards. As a result, it came to the conclusion that doing a 180-degree turn is faster than reversing.

From model test we noticed that the agent is not able to reverse.

III.4 Model Test

When training is complete, we use inference behavior in the agent behavior parameters to test the trained model.



Model: My Behavior-12000028 (NN Mc
Inference Device: Default
Behavior Type: Inference Only

Figure III.6: Trained model selected in behavior parameters

Testing the model in the environment it trained in:

A- Both the agent and the environment spawn when the simulation begins.

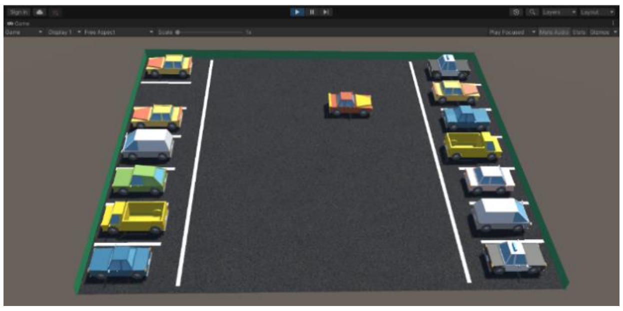
B- The agent moves forward and detects there are multiple parked cars in front of it, it begins to turn in order to avoid a collision.

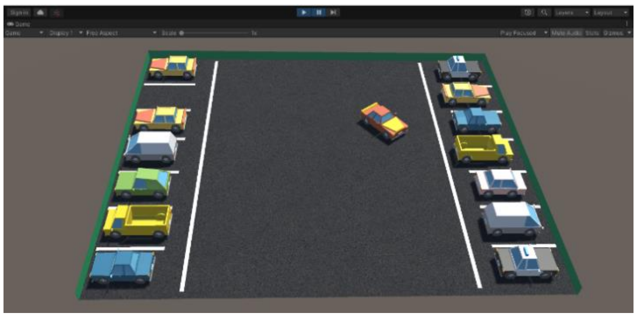
C- After turning the agent detects a wall infront of it again, at which point it continues turning completing a 180-degree turn.

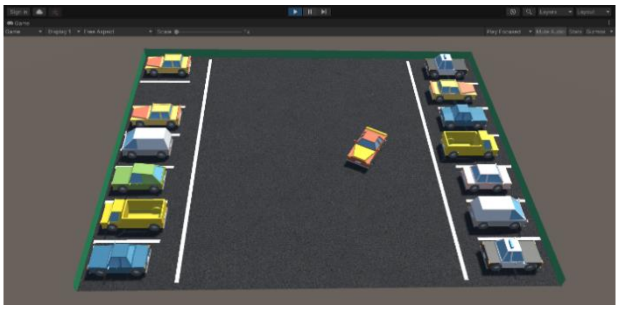
D- After making a 180 degrees turn the agent detects the parking spot infront of it and starts to move towards it.

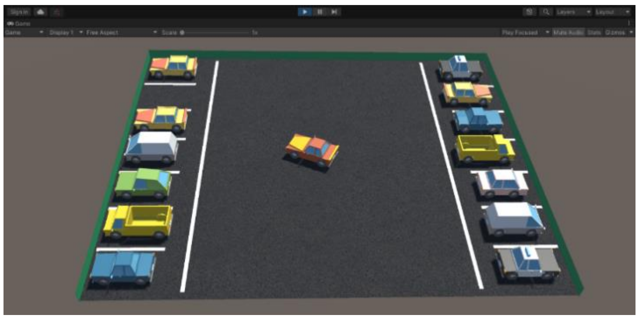
E- Finally, the agents makes some slight adjustements to its movement and enters the empty parking spot.

The testing is shown in **Figure III.7.**


(A)


(B)


(C)


(D)

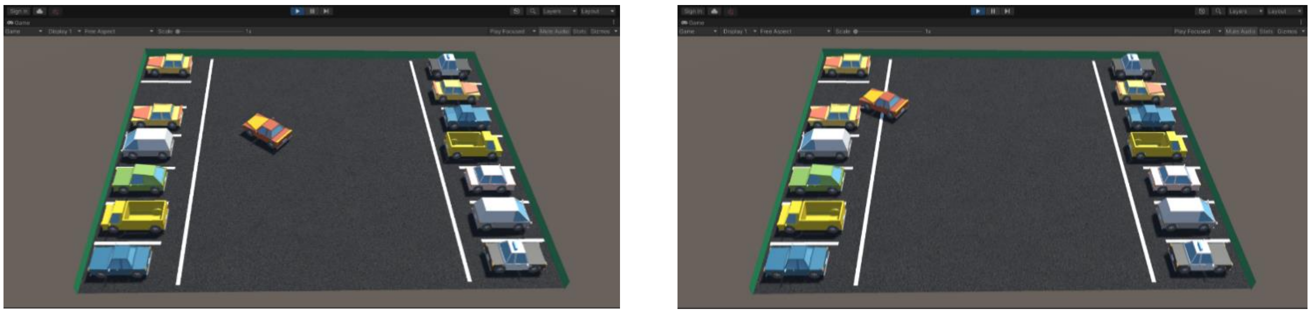

(E)

Figure III.7: Trained model test in training environment

III.5 Conclusion

This chapter included a discussion of the data that was collected throughout the training process in the form of graphs, followed by testing the final trained model.

We also talked about the major problems we came across while working on this project, and came to the conclusion that the training time issue might be resolved by utilizing powerful hardware, while the action optimization issue could be resolved by using a different reward configuration.

Despite the issues mentioned, the training's results were excellent, with a high success rate reaching 97%

General Conclusion

Our project's objective was to demonstrate the viability of utilizing reinforcement learning (RL) to create autonomous parking vehicles as a solution to parking issues.

Reinforcement learning (RL) is helpful in this circumstance because it can handle complex problems that people face without relying heavily on data or hard coding every aspect.

In order to accomplish this goal, we used the Unity video game engine to create an environment in which there is just one parking spot available in a parking lot that is packed with cars.

Then, we trained an agent to locate that open parking space and navigate to it using the Unity ML-Agents toolkit and the Proximal Policy Optimization (PPO) algorithm.

We face some difficulties during training due to hyperparameter tuning; after training we noticed that the policy developed a weird behavior of not using reverse but instead turning a full 180 degrees instead, this was to optimize the movement and save time.

Despite these difficulties the results were excellent, with a rate of success reaching 97%.

This project can be improved upon in multiple ways, first different algorithms such as Deep Q-Learning or Actor-Critic can be tried to find the most effective one for the specific parking scenario. Each algorithm has its strengths and weaknesses, and finding the right one is important.

Introducing various other obstacles like other cars or pedestrians can simulation more realistic. With different obstacles, the agent can respond better to various situations and improve its overall performance.

To ensure that the autonomous parking model can be successfully implemented in real-life situations, testing it in a real-world parking scenario is crucial. Testing the agent in a real-world parking environment can help identify any potential issues and improve its overall performance. In addition, creating a standard smart parking lot can make achieving autonomous parking easier. By integrating the autonomous parking model with a smart parking system, it can navigate the parking lot, direct the car to an available parking spot, and park the car safely and efficiently. The standard smart parking system can also help reduce congestion, optimize parking space usage, and improve parking safety. Testing the autonomous parking model within the standard smart parking system can help refine its performance and ensure that it performs reliably and safely in real-life parking situations.

We came to the conclusion from this project that video game engines are an excellent tool for creating simulations for many purposes, and reinforcement learning could be a solution to address various common issues due to its capacity to handle complex scenarios.

Bibliography

[1] Philip Boucher, **Book of** ‘Artificial Intelligence: How does it work, why does it matter, and what can we do about it?’, June 2020.

[2] Judith Hurwitz and Daniel Kirsch, **Book of** ‘Machine learning for dummies’, IBM Limited Edition, 2018.

[3] François Chollet, **Book of** ‘Deep learning with python’,2017.

[4] Ivan Nunes da Silva, Danilo Hernane Spatti, Rogerio Andrade Flauzino, Luisa Helena Bartocci Liboni, Silas Franco dos Reis Alves, **Book of** ‘Artificial Neural Networks: A practical course”, 2017

[5] J. Arockia Jeyanthi, Dr. S. Chidambaranathan, **Article of** ‘A Brief Study on Machine Learning Tools’, International Journal of Engineering Research & Technology (IJERT), 2020.

[6] Kevin Gurney, **Book of** ‘An introduction to neural networks’, 1997.

[7] www.github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md Accessed 3rd September 2022

[8] https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md Accessed 25th June 2022

[9] Richard S. Sutton and Andrew G. Barto, **Book of** ‘Reinforcement Learning: An Introduction’, Second edition, 2018.

[10] https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md Accessed 2nd July September 2022

[11] https://docs.unity3d.com/2021.2/Documentation/Manual/RigidbodiesOverview.html Accessed 12th August 2022

[12] https://docs.unity3d.com/2021.2/Documentation/Manual/CollidersOverview.html Accessed 12th August 2022

[13] https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.Agent.html Accessed 17th August 2022

[14] https://survey.stackoverflow.co/2022/ Accessed 5th July 2022

[15] https://code.visualstudio.com/docs/other/unity Accessed 18th September 2022

[16] Joseph Awoamim Yacim & Douw Gert Brand Boshoff, **Article of** ‘Impact of Artificial Neural Networks Training Algorithms on Accurate Prediction of Property Values’,Journal of Real Estate Research, 17 Jun 2020.

[17] https://medicalxpress.com/news/2018-07-neuron-axons-spindly-theyre-optimizing.html Accessed 24th June 2022

[18] https://www.ml-concepts.com/2022/04/04/everything-you-need-to-know-about-reinforcement-learning/?fbclid=IwAR05_j5e103zvaSexn_RBqntpjgsYq3kqJu0ZmeoJ8xBZ6A6ah8jGSXqRts Accessed 24th June 2022