

TECHNICAL UNIVERSITY OF CRETE  
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

# Physics-based Drone Racing Simulator in Virtual Reality



Michalis Oikonomakis

Thesis Committee

Professor Katerina Mania (ECE)

Professor Michail Lagoudakis (ECE)

Professor Panagiotis Partsinevelos (MRE)

Chania |Crete , March 2023



## Abstract

In this thesis, a virtual reality drone racing simulator with the use of avionics is presented, designed to help humans train to fly quadcopters at high speeds. With the increase of the usage of UAVs in different fields nowadays, comes the demand in having experienced pilots that can fly the aircrafts both at high speed but also avoid obstacles. In this simulator, real physics for the movement of the quadcopters is implemented. A quadcopter has four propellers in each of the four corners. For each propeller, speed and direction of rotation are controlled in order to handle the yaw, pitch, roll and thrust. Lift and drag forces were implemented in order for the movement of the quadcopter to be realistic . Wind zones were implemented using the normal distribution algorithm to make them feel more realistic. With the usage of Virtual Reality, the user is called to pass through checkpoints in three stages while also avoid obstacles in the way. This makes the training danger free in real time scenarios, since the user can restart the simulation whenever it hits an obstacle. Each stage provides a different training experience, the first one simulates an urban and suburban environment, the second one a coastal environment with more wind zones and the third one mostly blocks and two main colors to help ease the training. A challenging opponent in the form of AI was made using reinforcement learning algorithms. It is trained using a neural network and takes the same inputs as the player drone and provides outputs on how to handle its yaw,roll, pitch and thrust. The application contains three stages and the possibility of first or third person view while the user is moving solo or versus AI opponents. Moreover, easy and hard mode is included that adjusts how well the opponent AI is trained. The evaluation of the system focuses on noting the completion time of the stage and the number of restarts. It was evaluated if the existence of the AI opponents had any impact on the training process of the user.

## Acknowledgements

Initially, I would like to thank my supervisor associate professor Katerina Mania for her advice and support throughout the whole development of the thesis and the trust she gave me into undertaking this challenge on combining Virtual Reality, drones and machine learning.

Next, this work could have not been possible if not for all the amazing people of the surreal team in the university which were always there for me providing company and help whenever I needed. Especially, I wanted to thank Polychronakis Andreas, Fotis Giariskanis, Yannis Kritikos and Minas Katsiokalis which assisted my project from the very beginning with providing valuable advise and feedback.

I wanted to thank associate professor Panagiotis Partsinevelos that helped me a lot at the start with explaining me how drones work and for the ideas and feedback throughout the whole process of the thesis. I would also like to give my thanks to Angelos which work on Partsinevelos' lab and assisted me a lot with explanations of the physics and the testing of the movement.

I would like to thank BEST Chania for teaching me that by doing volunteer work and challenging yourself and getting out of your comfort zone is the best way to develop yourself in many aspects.

Finally, I want to thank my friends, my father Manolis, my Mother Evdokia and my sister Katerina for believing in me and supporting me throughout my time in the university and especially during the months that I worked on the thesis which were really pressuring.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Brief Description . . . . .	1
1.2	Structure of the Thesis . . . . .	5
<b>2</b>	<b>Research Overview</b>	<b>7</b>
2.1	Introduction . . . . .	7
2.2	Background on Drones . . . . .	8
2.2.1	Definition . . . . .	8
2.2.2	History of Drones . . . . .	8
2.2.3	Types of drones and their uses . . . . .	11
2.2.4	How drones fly . . . . .	15
2.2.5	Drone Simulators . . . . .	19
2.3	Virtual Reality . . . . .	20
2.3.1	Brief History of VR . . . . .	20
2.3.2	Immersion & Interaction . . . . .	22
2.3.3	Head Mounted Displays . . . . .	23
2.4	Types of Machine Learning . . . . .	26
2.4.1	Neural Network . . . . .	29
2.5	Drone Racing . . . . .	31
2.5.1	Overview of the sport of drone racing . . . . .	31
2.5.2	Challenges and obstacles in drone racing . . . . .	32
2.5.3	The role of VR and reinforcement learning in enhancing the drone racing experience . . . . .	32

## CONTENTS

---

<b>3</b>	<b>Technological Background and Definitions</b>	<b>35</b>
3.1	Introduction . . . . .	35
3.2	Unity . . . . .	36
3.3	Unity Structure and Architecture . . . . .	36
3.3.1	Open XR . . . . .	36
3.3.2	Unity New Input System . . . . .	36
3.3.3	Cinemachine . . . . .	37
3.4	Drag Equation . . . . .	38
3.5	Normal Distribution . . . . .	39
3.6	Anaconda . . . . .	40
3.7	PyTorch . . . . .	40
3.8	ML Agents . . . . .	41
<b>4</b>	<b>Use Case</b>	<b>45</b>
4.1	Introduction . . . . .	45
4.2	Main Menu Scene . . . . .	45
4.2.1	Canvas Main Menu . . . . .	46
4.2.2	Canvas Modes . . . . .	47
4.2.3	Canvas Scene Selection . . . . .	49
4.3	Pause Menu . . . . .	50
4.4	Ending Panel . . . . .	51
<b>5</b>	<b>Implementation</b>	<b>53</b>
5.1	Introduction . . . . .	53
5.2	Oculus Integration . . . . .	54
5.3	Player Input / Unity New Input System . . . . .	54
5.4	Drone Movement . . . . .	55
5.4.1	Mode A . . . . .	59
5.4.2	Mode B . . . . .	59
5.5	Physics . . . . .	60
5.5.1	State Detection . . . . .	60
5.5.2	Velocity Manager . . . . .	61
5.5.3	Drag . . . . .	63
5.5.4	Random Wind . . . . .	64

5.6	3D Environments . . . . .	66
5.6.1	Tutorial Scene . . . . .	66
5.6.2	Stage 1 . . . . .	68
5.6.3	Stage 2 . . . . .	69
5.6.4	Stage 3 . . . . .	70
5.7	Trigger Events . . . . .	71
5.7.1	CheckPointPass . . . . .	71
5.7.2	RestartOnCollision . . . . .	72
5.8	Game Manager Scripts . . . . .	72
5.8.1	Sound Manager . . . . .	72
5.8.2	Canvas Manager . . . . .	72
5.8.3	Panel Manager . . . . .	73
5.9	Setting Up Python Environment . . . . .	74
5.9.1	Anaconda . . . . .	74
5.9.2	Pytorch . . . . .	74
5.9.3	ML Agents Toolkit . . . . .	75
5.10	Training Scene . . . . .	79
5.10.1	Drone Area . . . . .	79
5.10.2	Drone Agent . . . . .	81
5.10.3	Observing Training . . . . .	90
5.10.4	Training Results . . . . .	91
5.11	3rd Party Assets . . . . .	91
<b>6</b>	<b>Evaluation &amp; Testing</b>	<b>93</b>
6.1	Introduction . . . . .	93
6.2	Evaluation Method & Result Analysis . . . . .	93
<b>7</b>	<b>Conclusion &amp; Future Work</b>	<b>97</b>
7.1	Conclusion . . . . .	97
7.2	Future Work . . . . .	98

## CONTENTS

---

# List of Figures

1.1	Photo from Stage 1 playing as first person . . . . .	3
1.2	Photo from Stage 2 playing as third person . . . . .	4
2.1	Tesla’s first radio-controlled boat 1898 . . . . .	9
2.2	Tesla’s presentation in Madison Square Garden in 1898 . . . . .	10
2.3	The Prime Minister, Mr Winston Churchill, with Captain The Right Honourable David Margesson, Secretary of State for War, watching preparations being made in an unspecified UK location for the launch of a De Havilland Queen Bee seaplane L5984 from its ramp. . . . .	11
2.4	Quadcopter . . . . .	12
2.5	Fixed-wing drone . . . . .	13
2.6	Single-rotor drone . . . . .	14
2.7	Drone Movement . . . . .	15
2.8	Controls of Quadcopter . . . . .	16
2.9	Forces Acting on a Drone in Vertical Position . . . . .	17
2.10	Forces Acting on a Drone in Inclined Position . . . . .	18
2.11	History of VR timeline . . . . .	20
2.12	Oculus Rift S . . . . .	23
2.13	Oculus Quest . . . . .	24
2.14	HTC Vive Pro . . . . .	25
2.15	Supervised Learning . . . . .	26
2.16	Unsupervised Learning . . . . .	27
2.17	Reinforcement Learning . . . . .	28
2.18	Types of Machine Learning . . . . .	29
2.19	Neural Network . . . . .	29

## LIST OF FIGURES

---

2.20	Neural Network Path . . . . .	30
2.21	Drone Racing League . . . . .	31
3.1	Cinemachine Smooth Path . . . . .	37
3.2	Drag Equation . . . . .	38
3.3	Normal Distribution Observations . . . . .	39
3.4	Normal Distribution Formula . . . . .	39
3.5	ML Agents Learning Environment . . . . .	41
3.6	Two phases for agents training and gameplay . . . . .	43
4.1	Canvas Main Menu . . . . .	46
4.2	Canvas Main Menu Use Case . . . . .	47
4.3	Canvas Modes . . . . .	48
4.4	Canvas Modes Use Case . . . . .	48
4.5	Canvas Scene Selection . . . . .	49
4.6	Scene Selection Use Case . . . . .	49
4.7	Pause Menu . . . . .	50
4.8	Pause Menu Use Case . . . . .	50
4.9	Ending Panel . . . . .	51
5.1	Input Action Functions . . . . .	55
5.2	Drone Movement Overview . . . . .	55
5.3	MovementUpDown . . . . .	56
5.4	MovementForward . . . . .	57
5.5	Rotation . . . . .	57
5.6	Swerve . . . . .	58
5.7	ClampingSpeedValuesModeA . . . . .	59
5.8	State Detection . . . . .	60
5.9	Velocity Manager Initialization . . . . .	61
5.10	Velocity Manager . . . . .	62
5.11	Drag Calculator . . . . .	63
5.12	Sampling Script . . . . .	64
5.13	Random Wind Diagram . . . . .	65
5.14	Tutorial Menu . . . . .	66

## LIST OF FIGURES

---

5.15	Tutorial Scene . . . . .	67
5.16	Stage 1 . . . . .	68
5.17	Stage 2.1 . . . . .	69
5.18	Stage 2.2 . . . . .	69
5.19	Stage 3 . . . . .	70
5.20	Checkpoint . . . . .	71
5.21	Application Managers Diagram . . . . .	73
5.22	Pytorch Installation . . . . .	74
5.23	yaml file . . . . .	78
5.24	Create Checkpoints . . . . .	79
5.25	Reset Agent Position . . . . .	80
5.26	Drone Agent Script Initialization . . . . .	81
5.27	On Action Received 1 . . . . .	82
5.28	On Action Received 2 . . . . .	84
5.29	Collect Observations Script . . . . .	85
5.30	VectorToNextCheckpoint Script . . . . .	85
5.31	On Episode Begin Script . . . . .	86
5.32	Got Checkpoint Script . . . . .	86
5.33	OnTriggerEnter and OnCollisionEnter Scripts . . . . .	87
5.34	Drone Sensors Setup . . . . .	88
5.35	Behavioural Parameters and Decision Requester . . . . .	89
5.36	Reward per Step Graph . . . . .	90
5.37	Training Results . . . . .	91
6.1	Trial 1 Restart Results . . . . .	94
6.2	Trial 2 Restart Results . . . . .	95

## LIST OF FIGURES

---

# Chapter 1

## Introduction

### 1.1 Brief Description

Unmanned Aerial Vehicles (UAVs), commonly known as drones, are aircrafts that are not driven by a human pilot but are controlled remotely by a human operator. In recent years, the use of UAVs is rapidly increasing in different fields. Companies, academics, researchers but also individual people are using these UAVs in many fields such as journalism, photography, videography, military, firefighting, search and rescue and many more. This increased the demand in having experienced pilots that can fly the aircrafts both at high speed but also avoid obstacles. This created the need of training of the human operators before flying the drones. A fun and more user-friendly approach to achieve this nowadays is the creation of drone racing competitions. Many of them take place all over the world with huge price pools and audiences. The goal is to complete a complex race course as quickly as possible and ahead of the other pilots. Nevertheless, all the racing environments face the risk of the drones to be damaged and this created the need for implementing a system where the users can train on these high speed and dangerous environments without damaging the drones.

Virtual Reality has become a highly developing field and with the prices of the headsets constantly going down, more and more people can afford one. Using the VR headsets the user can access virtual worlds and interact with them. Virtual reality helps in exploring places without actually being there. VR is also interactive and it can be used in the training of the users. More specifically in drone racing simulators, it can help reduce the resources that are needed in an actual race and also eliminate the dangers of flying

## 1. INTRODUCTION

---

a drone at these high risk environments as stated above. Despite the large amount of drone simulators and drone racing applications out there, there isn't yet an application created that can simulate a real VR drone racing experience. This makes the drone racing experience not so realistic and doesn't include the thrill of competing in a real FPV environment where the user can see the environment as if being inside the drone. Another important issue the drone operators face is getting the drones damaged or even destroyed during a flight, as stated previously, and these probabilities increase when the user is required to fly the drones at high speed such as an actual drone racing scenario.

In this thesis, we created a physics based VR drone racing simulator. This simulator provides the user with a very realistic drone racing experience since we used physics for the movement as well as wind zones. The user is called to pass through different checkpoints in different stages while also avoid obstacles in the way. This makes the training danger free, since the user can restart whenever it hits an obstacle while in real time scenarios the drone would have been damaged.

More specifically, we used the Unity game engine that offers the programming environment for the development of 3D computer graphics applications and can help create realistic physics. It also provides a lot of tools for the training of the AI that we used which we are gonna mention below. All the associated scripts were written in C# language which the engine uses.

After the integration of the Oculus Quest headset, we moved on to create a realistic movement for the drones. We created two drone modes that correspond to the drone modes of an actual drone that is stored in the laboratory. The first drone mode tries to hold its position when the drone is not moving while on the second one the user controls the pitch and, depending on that, moves the drone faster or slower and it stops by the drag force. We implemented the controls where with the right joystick the user can move the drone up or down and rotate it left or right by adjusting its yaw. On the other had, with the left joystick the user can move the drone forward or backward or make it bend left or right by adjusting its pitch and roll.

When the movement was complete, we focused on creating physics. The drag was implemented using the global drag equation that helped the movement feel more realistic rather than the standard built in drag and angular drag the unity editor provides. For the lift, a script was structured that helps the drone remain into a specific height while also maintaining its rotation. This was created because in the application forces might affect

## 1.1 Brief Description

---

the drone such as wind and a system was needed that helps the drone hold its position and rotation whenever the user doesn't provide input in the drone's position mode. In order to make the experience more realistic, random wind zones were created with the help of the Gaussian distribution which is a symmetric distribution that describes the variation of a random variable around its mean and many natural phenomena tend to follow this distribution.



Figure 1.1: Photo from Stage 1 playing as first person

In a race there are opponents, so we wanted to make the best opponents possible for the user to compete with. We trained AI opponents using reinforcement learning with the same inputs as the user in order for them to pass the stage fast. To implement that, we utilized the Unity ML Agents Package which assists in training AI using reinforcement learning. In order to run the package, connecting python environment with unity was required as well as defining the training parameters. The agent script that contains the actions and the reward the agent will get was defined and sensors were placed in the drone in order for it to collect observations from the environment so it can take decisions. The area and the checkpoints were created and the training started. In this thesis four different concepts for the training were tested that will be described furthermore in the chapter five. After many hours of training, the perfect opponent was made and we made

## 1. INTRODUCTION

---

it compete with the user. The application provides different features for the training such as playing first or third person, solo or versus the AI and easy or hard mode which adjusts how well the agents are trained.

In the end we evaluated how well the users did both on time spent passing the race while also the number of restarts they had before passing it. A comparison as well was made between playing solo or versus AI and if the competitive nature of the human has any effect into the process of training.

The main contributions of this thesis is an innovated mobile Virtual Reality training simulator for drone racing, a user study evaluation of our system that prove the improvement in performance of drone operators with and without using Artificial Intelligence and training with AI achieved higher performance rates compared to not using it.



Figure 1.2: Photo from Stage 2 playing as third person

## 1.2 Structure of the Thesis

In the following chapters as well as in this one, the whole thesis is presented in full detail.

Chapter 1 presents a small description of Drones, Drone Racing, Virtual Reality, a brief information of the implementation and the purpose and requirements of the application.

Chapter 2 describes the research that was conducted. The history of drones, their types and how they fly is presented and afterwards the role of drone simulators in providing effective training for drone pilots and operators is analyzed. Then, the history and VR is provided and an explanation why VR is immersive and interactive. The three main Head mounted Displays (HMDs) are shown and an explanation why we chose to go with the Oculus Quest headset. In the area of machine learning (ML), a discussion of the principles of neural networks and their application in drone racing is conducted. An overview of the sport of drone racing, the challenges and obstacles faced by pilots, and the potential for VR and reinforcement learning to enhance the drone racing experience will be provided.

Chapter 3 explains the technological background used in the thesis. A closer look into unity game engine and some of its features is provided as well as some of the physics and math equations that we used. Key information about how the training of the drone agents was conducted is presented by analyzing the Anaconda environment, the PyTorch framework and the ML Agents Unity Toolkit.

Chapter 4 goes through the user's experience through the application and the different capabilities that he has to interact with its UIs. All the menus are described and use case diagrams will also be provided for every UI.

## 1. INTRODUCTION

---

Chapter 5 explains the technical implementation of this project based on the tools and the information that have been presented earlier. The Integration in Oculus Quest, the movement of the drone and the physics development are presented. The whole process of training of the drone agents using the ML agents toolkit is described and some observations from the training process are collected and the final neural network is created.

Chapter 6 provides an evaluation of the training and how well the users performed both when playing against and without the AI trained opponents and if it had any impact on the training process.

Chapter 7 presents the conclusions of our physics based VR drone racing simulator. It sum ups the work done as well as if the results mirrored the original goal of the thesis. The chapter ends with suggestions and notes for future work.

# Chapter 2

## Research Overview

### 2.1 Introduction

At the beginning this thesis, research on various topics was conducted. This chapter will provide a thorough analysis of several key topics related to drone training and racing. Specifically, a dive into the history of drones will be conducted and an analysis of the various types of drones, as well as their uses and how they fly. Additionally, the role of drone simulators will be explored in providing effective training for drone pilots and operators. Turning our attention to virtual reality (VR), the brief history of this technology will be examined, as well as the concepts of immersiveness and interaction. Furthermore, the different types of head-mounted displays (HMDs) will be analyzed and their ability to provide realistic and immersive VR experiences. In the area of machine learning (ML), a discussion of the principles of neural networks and their application in drone racing will be conducted. Finally, an overview of the sport of drone racing, the challenges and obstacles faced by pilots, and the potential for VR and reinforcement learning to enhance the drone racing experience will be provided.

## 2.2 Background on Drones

### 2.2.1 Definition

Unmanned Aerial Vehicles (UAVs), commonly known as drones, are aircrafts that are not driven by a human pilot but are controlled remotely by a human operator. [1] They are essentially flying robots that are controlled remotely or can fly autonomously by using embedded code in its system that work in conjunction with different kind of sensors and a global positioning system (GPS).

### 2.2.2 History of Drones

”The world moves slowly, and new truths are difficult to see.”

Nikola Tesla’s way of responding to the crowd’s stunned disbelief upon viewing his scientific wizardry at New York’s Madison Square Garden in 1898.

**Nikola Tesla, 1898**

One of Tesla’s lesser-known but significant inventions along with the AC motor and wireless power was his development of the first radio-controlled boat in 1898, which served as a predecessor to modern-day remote-controlled drones. This technological feat showcases Tesla’s remarkable ingenuity and showcases the difficulties of inventing something ahead of its time. Tesla’s radio-controlled boat was a significant invention that marked a major milestone in the development of remote-controlled devices. His idea stemmed from his research into electromagnetic waves, which led him to design his own wave generator, the Tesla coil, capable of wirelessly illuminating lamps. By realizing that he could control devices at a distance, Tesla put an aim to create an automaton and began building a prototype in the form of a boat. This was a strategic choice given the ongoing naval armaments race of the 1890s. To achieve radio-controlled operation, Tesla transmitted a continuous radio wave to the boat which was detected by a ”delicate mechanism.” The transmitter box allowed him to interrupt the signal being sent to the boat by rotating a lever that touched four contacts. These interruptions caused a disk inside the boat to rotate and engage different contacts, thus activating the motors for the rudder or propeller. Remarkably, Tesla achieved this without using modern electronic components such as vacuum tubes, transistors or integrated circuits. [2]

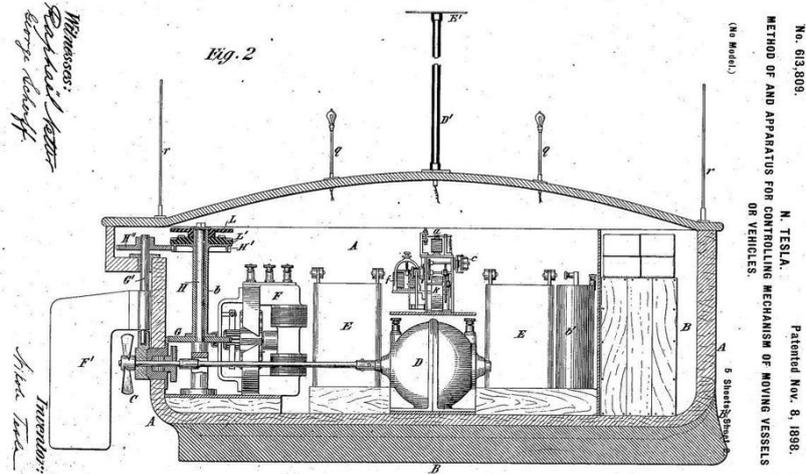


Figure 2.1: Tesla's first radio-controlled boat 1898

During his presentation in Madison Square Garden in 1898 by using a small, radio-transmitting control box, he was able to maneuver a tiny ship in a pool of water and even make its running lights open on and off, all without any visible connection between the boat and controller. It should be noted that during this time, only a handful of individuals were knowledgeable about the existence of radio waves. [3]

The development of this technology continued and many new inventions were created especially during the World Wars that followed. During the first World War, the first pilot-less air crafts were developed in Britain and the USA. Britain's Aerial Target, a small radio-controlled aircraft, was first tested in March 1917 while the American aerial torpedo known as the Kettering Bug first took flight in October 1918, although neither of those were used during the war. During the inter-war period, the development and testing of UAVs continued. In 1935 many radio controlled aircrafts were produced in Britain to be used as targets for training exercises. The name "drone" started to be used at that time and it was inspired by the name of one of those models called the DH.82B Queen Bee.

## 2. RESEARCH OVERVIEW

---

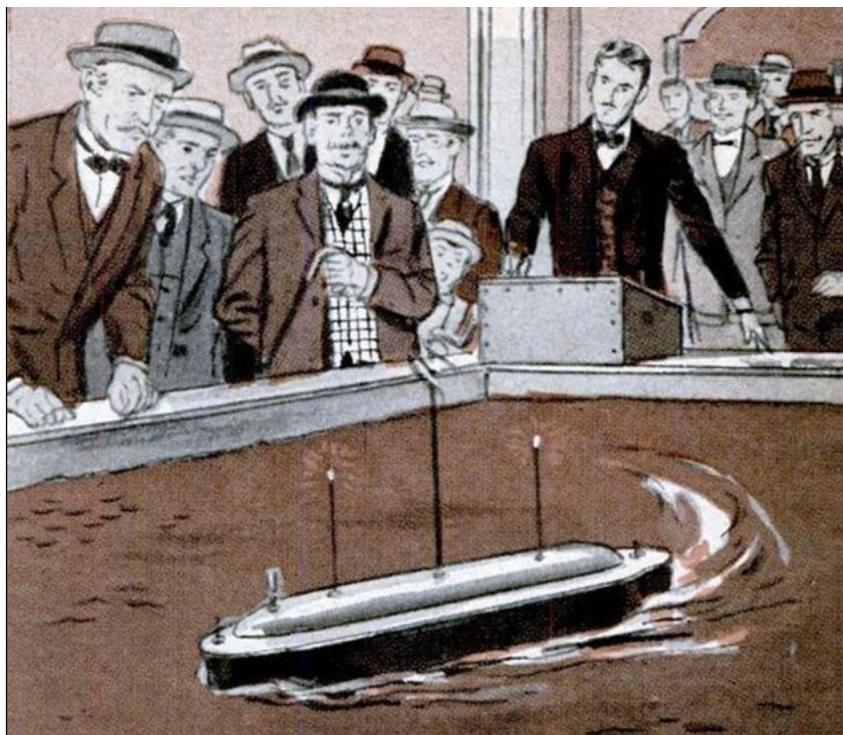


Figure 2.2: Tesla's presentation in Madison Square Garden in 1898

The first widespread deployment and use of drones was during the Vietnam War as dedicated reconnaissance UAVs. They also served as decoys in combat, launched missiles against fixed targets and dropped leaflets for psychological operations according to Imperial War Museum, London. [4]

In recent years, the use of UAVs is rapidly increasing in different fields. Companies, academics, researchers but also individual people are using these UAVs in many fields such as journalism, photography, videography, military, firefighting, search and rescue and many more [5]. This increased the demand in having experienced pilots that can fly the aircrafts both at high speed but also avoid obstacles. As a result, created the need of training of the human operators before flying the drones.



Figure 2.3: The Prime Minister, Mr Winston Churchill, with Captain The Right Honourable David Margesson, Secretary of State for War, watching preparations being made in an unspecified UK location for the launch of a De Havilland Queen Bee seaplane L5984 from its ramp.

### 2.2.3 Types of drones and their uses

Drones are of different types and sizes and are used for a variety of purposes. The most known ones are the following.

Multi-rotor drones, particularly quadcopters, are a popular choice among drone enthusiasts and professionals alike due to their enhanced control and maneuverability. With the ability to fly closer to structures and buildings, they are ideal for aerial photography and surveillance purposes. The versatility of multi-rotor drones allows them to be utilized in a variety of applications, such as visual inspections, thermal reports, photography, videography, and 3D scans.

However, multi-rotor drones also have their limitations. Their limited endurance, speed, and flight time, which is restricted to around 20-30 minutes with a lightweight camera payload, makes them unsuitable for large-scale aerial mapping and long-endurance monitoring. This is largely due to the current battery technology and the reliance on

## 2. RESEARCH OVERVIEW

---

electric motors, as using a gas engine is not practical for these types of drones. Additionally, multi-rotor drones are not particularly efficient, as they require a significant amount of energy just to remain airborne.

Despite these disadvantages, multi-rotor drones offer a cost-effective solution for aerial observations. Their ability to carry multiple payloads per flight increases their operational efficiency and reduces the time required for inspections. Until a new power source is developed, multi-rotor drones will continue to be a popular choice for those looking for an affordable and practical "eye in the sky."

In conclusion, multi-rotor drones, especially quadcopters, are a popular and useful tool for aerial observations and inspections, offering versatility and enhanced control. While they have limitations, their affordability and efficiency make them a suitable option for various applications.



Figure 2.4: Quadcopter

## 2.2 Background on Drones

---

Fixed-wing drones, with their one rigid wing designed to function like an airplane, are known for their energy efficiency and extended flight times. Unlike multi-rotor drones that require energy to maintain vertical lift, fixed-wing drones only need energy to move forward. This makes them suitable for covering longer distances, mapping larger areas, and loitering for extended periods of time for monitoring purposes. The average flight time for a fixed-wing drone is a few hours, but with a greater energy density of fuel from a gas engine, many can stay aloft for 16 hours or more. Additionally, fixed-wing drones can fly at higher altitudes, carry more weight, and are more forgiving in the air than other drone types.

However, fixed-wing drones can also be expensive, and training is usually required to operate them. Flying a fixed-wing drone for the first time can be challenging, as they move much quicker than multi-rotor drones, and require confidence in control abilities for a successful flight and landing. Furthermore, a launcher is usually needed to get a fixed-wing drone into the air.

Fixed-wing drones have numerous technical uses, including aerial mapping, drone surveying in various fields such as forestry, environmental surveys, pipeline surveys, and coastal surveys, agriculture, inspection, construction, and security. Despite its challenges, the extended flight time and energy efficiency of fixed-wing drones make them a valuable tool for various industries.



Figure 2.5: Fixed-wing drone

## 2. RESEARCH OVERVIEW

---

Single-rotor drone types, also known as helicopter-style drones, have a unique design that sets them apart from other drone types. They have one rotor and a tail rotor, which is similar to a traditional helicopter structure. The advantages of using a single-rotor drone include its increased efficiency compared to multi-rotor drones, especially if the drone is gas-powered, and its ability to hover with a heavy payload, making it ideal for aerial laser scanning and drone surveying. The longer blades of the single-rotor helicopter make it more efficient in flight. However, single-rotor drones are complex and expensive, and they also require a significant amount of maintenance and care due to their mechanical complexity. Furthermore, the vibration from a single-rotor drone makes it less stable and more dangerous in the event of a bad landing, especially with the long, heavy blades. Despite these disadvantages, a single-rotor drone is still a valuable tool for carrying heavy payloads and performing aerial laser scanning and drone surveying tasks. [6]



Figure 2.6: Single-rotor drone

### 2.2.4 How drones fly

In this thesis we focused on the physics of quadcopters. A quadcopter has four propellers in each of the four corners. For each propeller, speed and direction of rotation are independently controlled for the balance, rotation and movement of the drone. All four rotors are placed at an equal distance from each other and one pair of rotors rotates in a clockwise direction while the other pair rotates in an anti-clockwise direction. To move up or hover, all rotors should run at high speed. Also, by changing the speed of rotors, the drone can be moved forward, backward, and side-to-side. The movement of the drone can be categorized into four types based on the motion of the four propellers: throttle, pitch, roll and yaw.

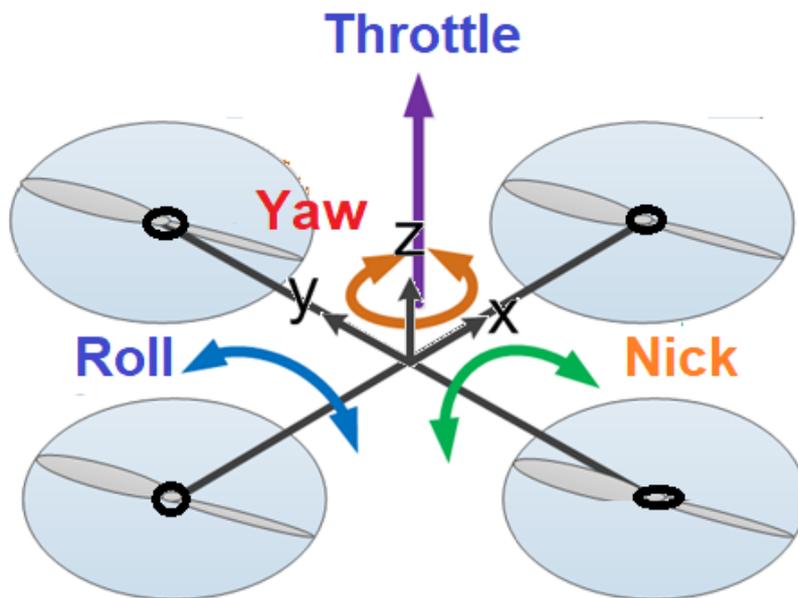


Figure 2.7: Drone Movement

## 2. RESEARCH OVERVIEW

**Throttle:** Up and down movement of the drone is called throttle. If all four propellers run at normal speed, then the drone will move down while if they run at a higher speed, then the drone will move up.

**Pitch:** Movement of a drone about a lateral axis (forwards or backwards) is called pitch. If two back propellers run at high speed, then the drone will move forwards while if the two front propellers run at high speed, then the drone will move backwards.

**Roll:** Movement of a drone about the longitudinal axis is called roll. If two right propellers run at high speed, then the drone will move left while if the two left propellers run at high speed, then the drone will move right.

**Yaw:** The rotation of the drone about the vertical axis (left or right) is called Yaw. If two propellers of a right diagonal run at high speed, then the drone will rotate anti-clockwise while if two propellers of a left diagonal run at high speed, then the drone will rotate clockwise.

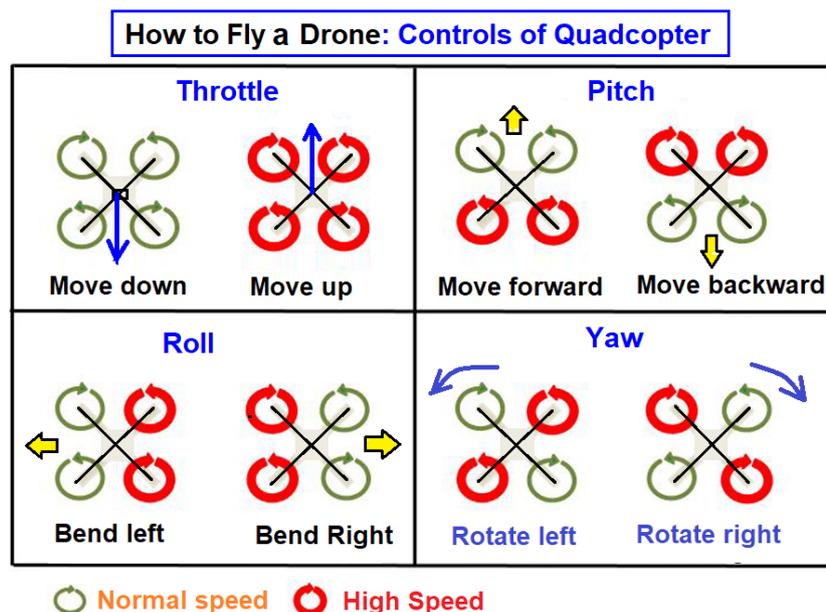


Figure 2.8: Controls of Quadcopter

Now let's talk about physics and see what forces are acting on a Drone while it is flying which decide its movement.

**Weight:** The drone has mass, so like any other object the body mass force acts in the direction of gravity. The higher the weight of the drone, the more power is required to lift and move the drone. The weight is calculated as mass of the drone x gravitational acceleration.

**Lift:** The vertical force acting on the drone is called lift and is due to vertical pressure differences across the drone. Hence, the speed, size, and shape of the propeller blade decide the amount of lift force. Lift is essential to lift the body against the gravity. In order for the drone to fly upwards, the combined vertical thrust of all four propellers minus the combined drag (which will be mentioned below), that is equal to the lift of the drone needs to be greater than its weight. If it is equal the drone hovers in the air and if its lower then it starts falling down.

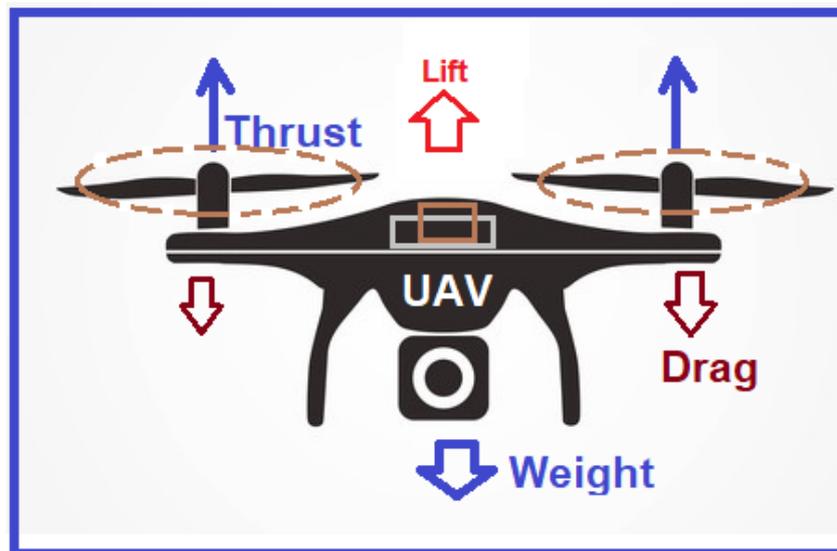


Figure 2.9: Forces Acting on a Drone in Vertical Position

## 2. RESEARCH OVERVIEW

---

**Thrust:** The force acting on the drone in the direction of motion is called thrust. However, for drone dynamics, it is normal to the rotor plane. When the drone hovers or moves up and down, the thrust is purely vertical. When the thrust is inclined the drone tilts forward or backward which is essential to move the drone in the desired direction at equal speed. To get the desired motion, two propellers have been given high speed as stated previously in the motion types.

**Drag:** The force acting on the drone in the opposite direction of its motion due to air resistance is called drag. This is because of the pressure difference and viscosity of air. In order to adjust the drag, the aerodynamic shape of the drone needs to be selected. More information about how drag works will be analyzed in chapter 3.4 Drag Equation.

[7]

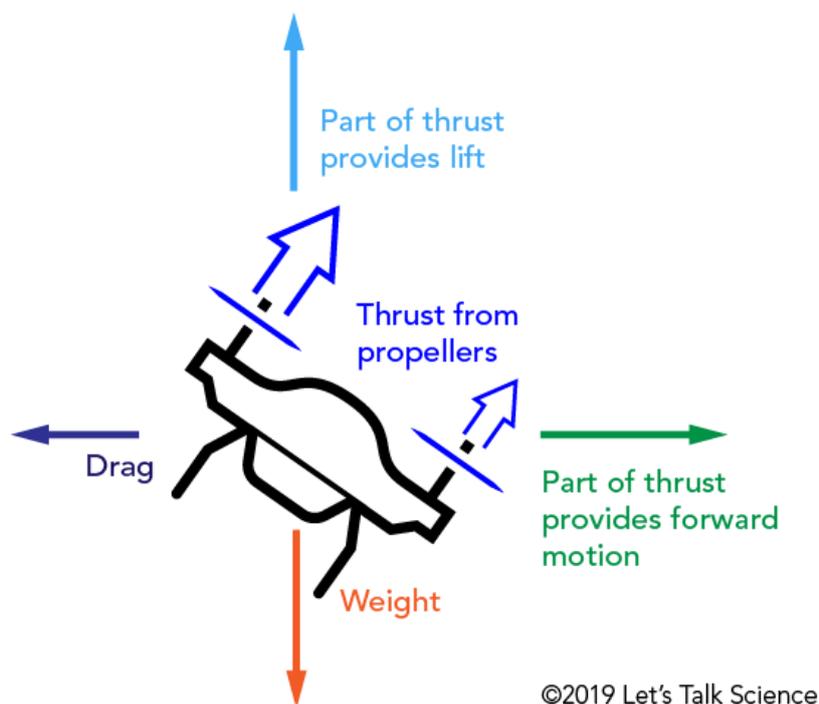


Figure 2.10: Forces Acting on a Drone in Inclined Position

### 2.2.5 Drone Simulators

With recent advances in hardware and software technology, drones are increasingly present in research activities. Many research papers nowadays focus on how drones fly and how we can optimize it.

Some focus on making simulations with different areas of interest. Two types of drone simulators are currently available on the market: racing simulators and commercial simulators. Drone racing simulators such as Drone Racing Simulator (DRL) [9] and VelociDrone FPV Racing Simulator [10] are designated for first-person-view (FPV) drone race training. In these simulators, the user is trained on various courses with gates and obstacles in order to simulate competitive drone racing sessions, which refine users' drone handling skills and improve their maneuvering speed in challenging flight environments. Although they simulate the race well and help the user train, they don't make the experience realistic like being inside the actual drone similar to the goggles that the users wear in actual competitions to watch the environment from the drone. On the other hand, commercial drone simulators are designed to train users for commercial applications. Many of them on the market offer a variety of virtual environments for different drone applications. As an example DroneSim, a VR-based flight training simulator for drone-mediated building inspections [11], an AR/VR-Hybrid Interaction System for Historical Town Tour Scenes Incorporating Mobile Internet [12] that implements an online scenic tour interaction system using AR and VR technologies combining 720° panoramic images to create a realistic experience just like a field trip, Simlat UAS Simulation [13] that provides scenarios including search and rescue, mining, maritime exploration, power line inspection, wind turbine inspection, railroad inspection, pipeline inspection, infrastructure security, among others. These VR simulators feel more realistic but don't help the user train in high speed environments.

## 2. RESEARCH OVERVIEW

### 2.3 Virtual Reality

#### 2.3.1 Brief History of VR

Virtual Reality has become a highly developing field and with the prices of the headsets constantly going down, more and more people can afford one. Using the VR headsets the user can access virtual worlds and interact with them.

Lets go back and see how it all started. The following timeline shows some of the biggest milestones in the history of Virtual Reality.

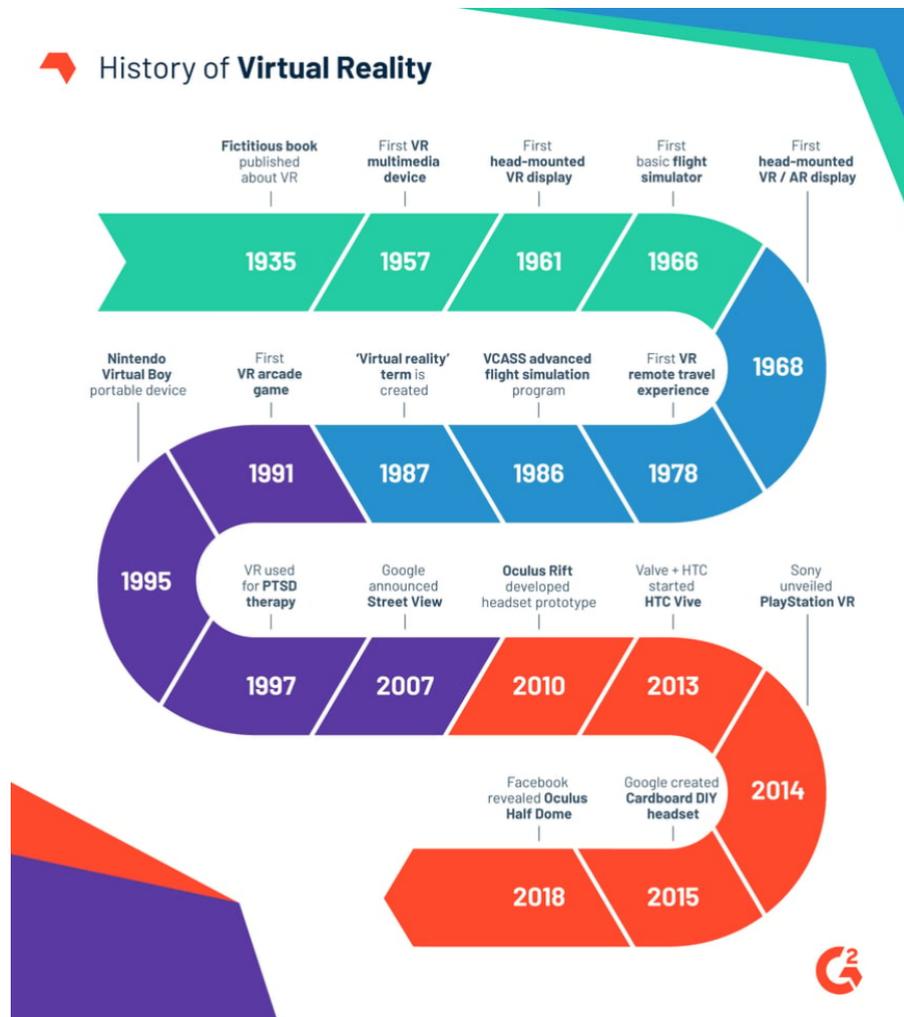


Figure 2.11: History of VR timeline

**1957:** Morton Heilig, a cinematographer, invented Sensorama, a theatre cabinet multimedia device that offered viewers an interactive experience. The device stimulated the users' senses with a viewing screen for sight, oscillating fans for touch, devices that emitted smells, and audio speakers for sound.

**1961:** Comeau and Bryan, two Philco Corporation engineers, created the first head-mounted display (HMD) called the Headsight. The display had two video screens, one for each eye and a magnetic tracking device. It was the first motion-tracking device that was ever created. It was essentially used to move a remote camera that allows a user to look around an environment without physically being there.

**1966:** Thomas Furness, a military engineer, developed the first flight simulator for the Air Force. This provoked a lot of interest in VR technologies and its training capabilities.

**1968:** Ivan Sutherland, a Harvard professor and computer scientist, invented the first VR / AR head-mounted display and named it 'The Sword of Damocles'. This HMD was connected to a computer, did not rely on cameras, and showed users primitive computer graphics.

**1978:** The Aspen Movie Map, developed by MIT, used photographs taken from a car in Aspen, Colorado to give viewers what they called a "Surrogate Travel" experience. It was an interactive first-person view of the city, a predecessor of VR version of Google Street View.

**1986:** Furness, that was mentioned earlier, worked on his Air Force simulation project through the 80s and in the end developed the Visually Coupled Airborne Systems Simulator (VCASS). The system gave pilots a virtual view that streamlined the barrage of information they get every moment. VCASS led to the invention of The Super Cockpit program that helps pilots make better decisions, faster, with technology like computer-generated 3-D maps, infrared, and radar imagery.

**1987:** John Lanier, computer scientist, researcher, and artist, coined the term 'virtual reality'. Through his company VPL research John developed a range of virtual reality gear including the Dataglove (along with Tom Zimmerman) and the EyePhone head mounted display making VPL the first company to sell VR goggles.

**1991-1995:** Series of games and arcade machines were created, bringing VR to the general public.

**2007:** Google, with Immersive Media, announced Street View. The technology launched using images of five mapped cities. The panoramic images were captured from a patented

## 2. RESEARCH OVERVIEW

---

camera mounted on a moving car to show users roads, inside buildings, and more.

**2010:** Palmer Luckey designed a prototype for what would become later on the Oculus Rift virtual reality headset. Facebook bought Oculus VR for around 3 billion USD in 2014 resulting in a lot of VR HMDs to be created and sold. Microsoft, Sony, and Samsung followed up with releasing their own Virtual Reality HMDs while Google introduced Cardboard, a do-it-yourself stereoscopic viewer where a user places their phone inside a literal piece of Cardboard to wear on your head. [15]

### 2.3.2 Immersion & Interaction

Immersion is a description of a technology, and describes the extent to which the computer displays are capable of delivering an inclusive, extensive, surrounding and vivid illusion of reality to the senses of a human participant. [16] Virtual reality helps in exploring places without actually being there. This makes the life of people much easier and more entertaining. It helps someone live a whole experience in a digital environment without the need of actually being there in real life. It creates a perceptual experience that fully engages the user through the use of visual, auditory, and other sensory inputs, resulting in a sense of being completely immersed in a simulated environment. This level of immersion can transform a person's awareness of their physical self, creating a state of consciousness that feels as though they are truly present in an artificial world.

VR is also interactive. The users can move inside the world and interact with objects using the hand-held controllers. This equipment allows the user to see the imaginary environment based on the topic, interact with and analyze it [8]. Since it is interactive, it can be used in the training of the users. More specifically in drone racing simulators, it can help reduce the resources that are needed in an actual race and also eliminate the dangers of flying a drone at these high risk environments.

### 2.3.3 Head Mounted Displays

A Head Mounted Display (HMD) is a device that can be worn over one or both eyes and is used in various fields such as gaming, aviation, engineering, medicine, and virtual reality. The HMD is a crucial component of virtual reality headsets. Commercially available HMDs, such as the Oculus Rift S, Oculus Quest and HTC Vive Pro, are now affordable options in the market and commonly used headsets.



Figure 2.12: Oculus Rift S

The Oculus Rift S headset has a Fast-switch LCD 2560×1440 (1280×1440 per eye) and 80 Hz refresh rate. It's field of view (FOV) is 115 degrees. Sensors include accelerometers, gyroscopes and magnetometers. The system also enables full 360-degree positional tracking. The touch controllers, that are included, are a pair of tracked controllers that provide intuitive hand presence in VR and provide the feeling that the virtual hands are actually your own. The Oculus Rift S is a tethered VR headset designed to work with a PC.

## 2. RESEARCH OVERVIEW

---



Figure 2.13: Oculus Quest

The Oculus Quest uses a Qualcomm Snapdragon 835 system-on-chip (SoC) with 4 GB of RAM. The software uses three out of the four 2.3 GHz CPU cores of the chip, while the remaining core and its four lower-power cores are reserved for motion tracking and other background functions. It runs an Android-based operating system, with modifications to enhance performance in VR applications. A diamond Pentile OLED display is used for each eye, with an individual resolution of  $1440 \times 1600$  and a refresh rate of 72 Hz. To enable Oculus Insight tracking, the design of the second generation Oculus Touch controllers was adjusted in the Oculus Quest. The tracking rings were relocated from the back of the controllers to the top, allowing them to be detected by the headset's cameras. Lastly, Oculus Quest is a standalone device that can run games and software wirelessly under an Android-based operating system.



Figure 2.14: HTC Vive Pro

The HTC vive pro has an embedded eye tracker and a frequency for gaze data output at 120Hz for both eyes(binocular) and a trackable FOV of 110 degrees. It has a Dual AMOLED 3.5” diagonal, a resolution of 1440 x 1600 pixels per eye (2880 x 1600 pixels combined) and a refresh rate is at 90 Hz. Precise, 360-degree controller and headset tracking, realistic graphics, directional audio and HD haptic feedback makes up for a very realistic movement and actions in the virtual world.

In this thesis we decided to go with the Oculus Quest headset for the following reasons. It is a wireless and standalone device, meaning it does not require any external sensors or a PC to operate. This provides greater freedom of movement and flexibility for the user, which is particularly beneficial for a high-speed environment such as a drone simulator. The standalone nature of the Oculus Quest makes it more portable and easy to set up compared to the Oculus Rift S and HTC Vive Pro, which require a PC and external sensors. Lastly, the Oculus Quest is more affordable than the Oculus Rift S and HTC Vive Pro which could be an important factor for students or researchers on a budget but also for the general public that want to learn to fly drones.

## 2. RESEARCH OVERVIEW

---

### 2.4 Types of Machine Learning

The study of Machine Learning is focused on developing techniques that enable systems to "learn" from data and use this knowledge to enhance their performance in accomplishing various tasks. This field is generally considered a subset of artificial intelligence. Machine learning involves creating models by analyzing sample data, also called training data, to enable them to make predictions or decisions autonomously, without requiring explicit programming. Machine learning involves showing a large volume of data to a machine so that it can learn and make predictions, find patterns, or classify data. The three main machine learning types are supervised, unsupervised, and reinforcement learning.

**Supervised Learning:** Supervised machine learning involves providing historical input and output data to machine learning algorithms, with intermediate processing steps between each input/output pair that allow the algorithm to adjust the model to produce outputs that closely match the desired results. This approach is referred to as "supervised" because the algorithm is provided with labeled data to aid in the learning process. The labeled data represents the desired outcomes, while the remaining information is used as input features. For instance, if you wanted to understand the relationship between borrower information and loan defaults, you might give the machine 500 cases of borrowers who defaulted and 500 who did not. This labeled data "supervises" the machine's learning to identify the patterns you are interested in.

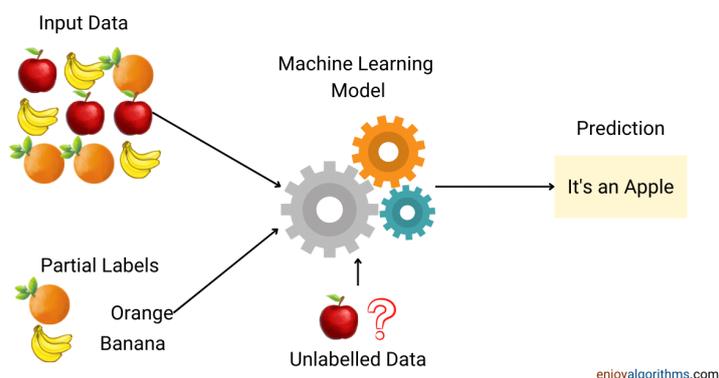


Figure 2.15: Supervised Learning

**Unsupervised Learning:** Unsupervised learning is a type of machine learning that does not rely on labeled training sets or data. Instead, the machine is tasked with identifying noticeable patterns in the data. This approach is particularly useful when you need to detect patterns and use data to make decisions. To illustrate this, consider the above example of identifying loan defaulters using unsupervised learning. Instead of providing the machine with labeled data, you would feed it borrower information and let it search for patterns among the borrowers so in the end it groups them into several clusters. This type of machine learning is widely used to develop predictive models.

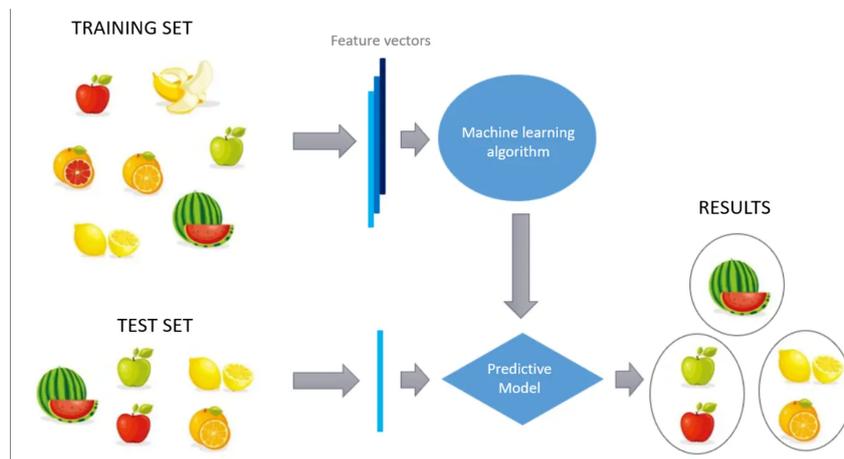


Figure 2.16: Unsupervised Learning

**Reinforcement Learning:** Reinforcement Learning involves an agent exploring an unknown environment to achieve a goal. It is the field of machine learning about learning the optimal behavior in an environment to obtain maximum reward. This optimal behavior is learned through the agent's interactions with the environment and observations of how it responds, similar to children exploring the world around them and learning the actions that help them achieve a goal.

Without having a supervisor, the learner must independently discover the sequence of actions that maximize the reward. The process of discovery can be compared to a trial-and-error search. The quality of actions is measured by not just the immediate reward they return, but also the delayed reward they might collect. As it can learn the

## 2. RESEARCH OVERVIEW

---

actions that result in eventual success in an unseen environment without the help of a supervisor, reinforcement learning is a very powerful algorithm.

In this thesis we had to train drone agents to learn to pass through some checkpoints fast while also avoiding obstacles. Reinforcement learning is a suitable approach for this training process because the drone agents can learn from their own experiences without requiring labeled training data. It is particularly useful for complex and dynamic environments, where unsupervised learning may not be sufficient. Reinforcement learning is ideal for scenarios that require real-time decision-making and can learn optimal policies that maximize a reward signal. Additionally, the interactive learning between the drone and the environment enables the drone to adjust its behavior based on feedback, which is essential in dynamic environments. Therefore, reinforcement learning is a suitable approach for training drones to navigate through checkpoints.



Figure 2.17: Reinforcement Learning

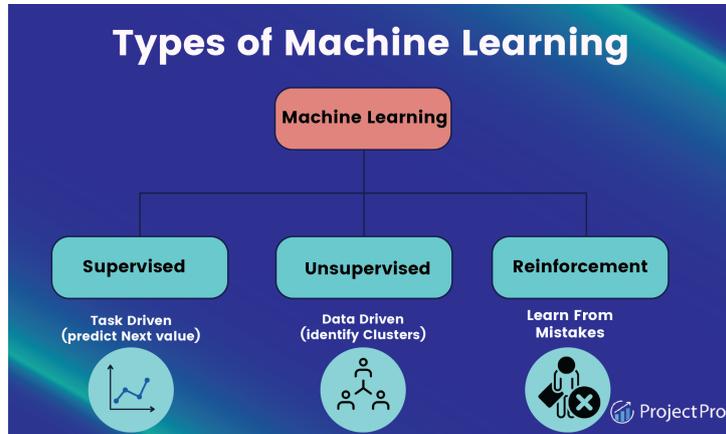


Figure 2.18: Types of Machine Learning

### 2.4.1 Neural Network

Neural networks are algorithms designed to identify the underlying relationships within a set of data, using a process that mimics the way the human brain functions. These systems consist of a network of neurons, which can be either organic or artificial in nature.

A neural network typically consists of an input layer, one or more hidden layers, and an output layer. The input layer includes one or more feature variables, also known as input variables. The hidden layer contains one or more hidden nodes or units, while the output layer consists of one or more output units.

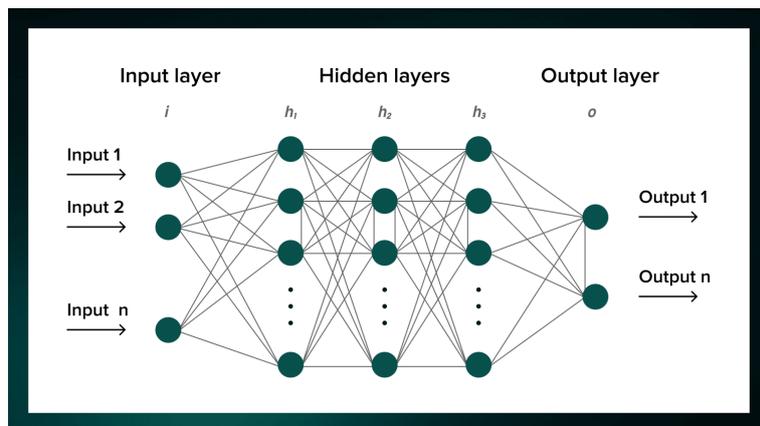


Figure 2.19: Neural Network

## 2. RESEARCH OVERVIEW

---

At its core, a neural network is essentially a network of equations. Each node in the network comprises of two functions: a linear function and an activation function. The linear function can be visualized as a line, while the activation function acts like a switch, producing a number between 0 and 1.

The input features are first processed by the linear function of each node, resulting in a value,  $z$ . This value is then passed through the activation function, which determines whether the switch is turned on or off. In this way, each node determines which nodes in the following layer will be activated, and this process continues until an output is produced. [14]

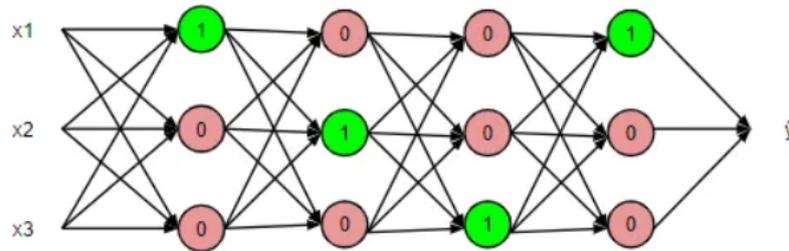


Figure 2.20: Neural Network Path

## 2.5 Drone Racing

### 2.5.1 Overview of the sport of drone racing

FPV drone racing is a thrilling sport that involves using remote-controlled drones equipped with cameras. Participants wear head-mounted displays that show the live camera feed from the drones, allowing them to control the drones as if they were flying inside them. The objective of the race is to complete a predefined course in the shortest time possible, much like air racing with full-sized aircraft.

Many of them take place all over the world with huge price pools and audiences. One of the most known ones is the Drone Racing League, also known as DRL, which is a First Person View (FPV) drone racing competition where pilots control drones equipped with cameras while wearing goggles that stream the live video feed from the drones so they feel like they're flying from inside the drone [9].

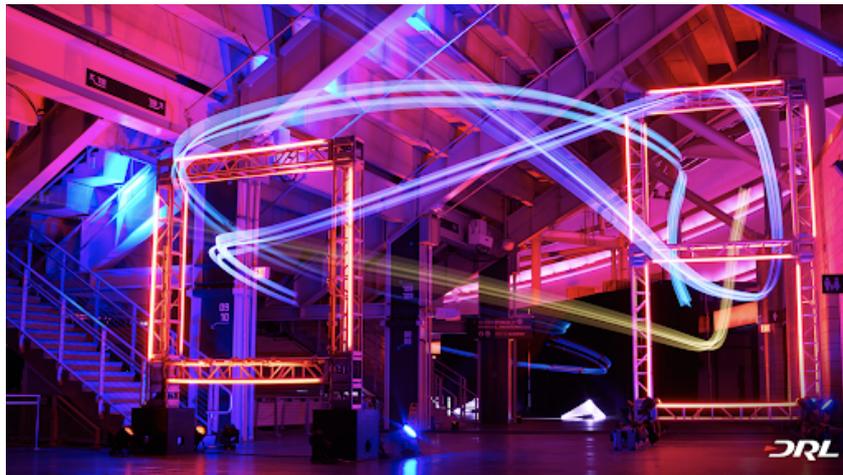


Figure 2.21: Drone Racing League

## 2. RESEARCH OVERVIEW

---

### 2.5.2 Challenges and obstacles in drone racing

Despite the high popularity of the sport, all racing environments face the risk of the drones to be damaged. Weather conditions can also have a significant impact on drone racing. High winds, rain, and snow can make it difficult for pilots to control their drones, and can even cause crashes. This created the need for implementing a system where the users can train on these high speed and dangerous environments without damaging the drones.

### 2.5.3 The role of VR and reinforcement learning in enhancing the drone racing experience

The demand in having experienced pilots that can fly the aircrafts both at high speed but also avoid obstacles creates the need of training of the human operators using Head Mounted Displays (HMD) such as Virtual Reality Headsets so they can become more competent when flying the drones. The usage of Virtual Reality headsets that are interactive can help assist the operators in this kind of training. Despite the large amount of drone simulators and drone racing applications out there, there isn't yet an application created that can simulate a real VR drone racing experience. This makes the drone racing experience not so realistic and doesn't include the thrill of competing in a real FPV environment where the user can see the environment as if being inside the drone.

Another important issue the drone operators face is getting the drones damaged or even destroyed during a flight, as stated previously, and these probabilities increase when the user is required to fly the drones at high speed such as an actual drone racing scenario. Some research have found ways to analyze these dangerous conditions and help guide the drone but, during the training, an application is needed that eliminates all of these dangers and still keeps the experience real.

Regarding autonomous drone flight, although there have been a lot of research to find good path planing algorithms, there isn't yet a scenario where the autonomous trained drone is faced in a race with a human in a virtual environment and how well the human can go versus the trained drone. In this thesis we use reinforcement learning to train AI drones, with the same inputs as the users to pass the different stages fast. A "perfect" opponent is made and it is put into a test versus the human operator. Through the use

of feedback from human testers, we analyze how playing alone or against an AI affects the training experience and if it results in any improvement in performance.

## 2. RESEARCH OVERVIEW

---

# Chapter 3

## Technological Background and Definitions

### 3.1 Introduction

In this chapter of the thesis, the technological background used in the thesis will be explained. A closer look into unity game engine will be conducted as well as some of its key features used in this thesis such as openXR, Unity New Input System and Cinemachine. Furthermore, we take a closer look on the physics part with the drag equation and normal distribution be explained. Lastly, key information about how the training of the drone agents was conducted is presented by analyzing the Anaconda environment, the PyTorch framework and the ML Agents Unity Toolkit.

## 3. TECHNOLOGICAL BACKGROUND AND DEFINITIONS

---

### 3.2 Unity

Unity is a popular cross-platform game engine that enables developers to create 2D, 3D, virtual reality, and augmented reality games and interactive experiences. It provides a wide range of tools and features to help developers create games, including an integrated development environment (IDE), physics engine, animation system, scripting tools, and a vast library of assets and resources.

### 3.3 Unity Structure and Architecture

#### 3.3.1 Open XR

Unity OpenXR is a set of tools and APIs provided by Unity that enables developers to create applications that can run on multiple virtual and augmented reality platforms such as Oculus Quest, HTC Vive, Microsoft HoloLens, and more. This makes it easier for developers to create applications that can reach a broader audience, without the need to rewrite the application for each specific device or platform.

Unity OpenXR also provides various tools and features to help developers create immersive experiences, including spatial mapping, hand tracking, and haptic feedback. Overall, Unity OpenXR is a powerful tool for developers who want to create VR and AR experiences and provides flexibility and ease of development.

#### 3.3.2 Unity New Input System

Unity's New Input System is a modular input system that allows developers to manage input devices and process user input in a flexible and efficient way. It is designed to replace Unity's legacy input system and provides several new features and improvements. It allows developers to easily define input actions, such as button presses, joystick movements and map them to different devices and platforms. This makes it easier to manage input across multiple devices and platforms, such as keyboards, mice, game-pads, touchscreens, and more.

The New Input System also supports advanced features such as input remapping, input binding UI, input interactions, and events, making it easier for developers to create complex input systems for their games and applications.

## 3.3 Unity Structure and Architecture

In this thesis we used the new input system to implement the player controls on both the joysticks of the Oculus Quest and the keyboard.

### 3.3.3 Cinemachine

Cinemachine is a Unity package that provides a suite of camera tools and behaviors for controlling virtual cameras in real-time. It allows game developers to create dynamic and cinematic camera shots for their games and virtual reality experiences. Cinemachine provides a variety of camera features, including advanced camera paths, procedural noise, virtual cinematography tools, and more, that can help game developers create visually interesting and immersive experiences.

In this thesis one of these features was used that is called Cinemachine Smooth Path. It is a component that defines a world-space path, consisting of an array of waypoints. Each waypoint has position and roll settings. Cinemachine uses Bezier interpolation to calculate positions between the waypoints to get a smooth and continuous path. The path passes through all waypoints. Unlike Cinemachine Path, first and second order continuity is guaranteed, which means that not only the positions but also the angular velocities of objects animated along the path will be smooth and continuous. [17] This feature was used to create the path that the checkpoints the drone has to pass will spawn and define their position and orientation.

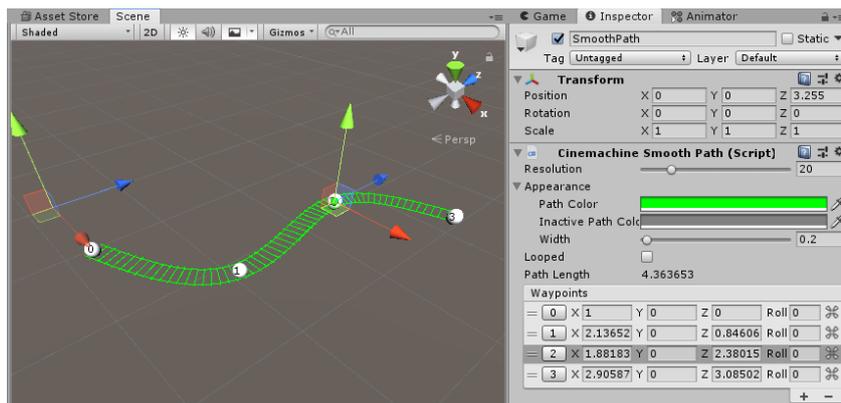


Figure 3.1: Cinemachine Smooth Path

### 3. TECHNOLOGICAL BACKGROUND AND DEFINITIONS

---

#### 3.4 Drag Equation

Drag is a mechanical force that is generated by the interaction and contact of a solid body with a fluid (liquid or gas). In aerodynamics, drag refers to forces that oppose the relative motion of an object through the air. Drag always is opposed to the motion of the object and, in an aircraft, is overcome by thrust.

Drag depends on the density of the air, the square of the velocity, the air's viscosity and compressibility, the size and shape of the body, and the body's inclination to the flow. In general, the dependence on body shape, inclination, air viscosity, and compressibility is very complex.

One way to deal with complex dependencies is to distinguish the dependence by a single variable. For drag, this variable is called the drag coefficient, designated "Cd." This allows all the effects, simple and complex, to be collected into a single equation. The drag equation states that drag D is equal to the drag coefficient Cd times the density  $\rho$  times half of the velocity V squared times the reference area A. [18]

**The Drag Equation**

$$D = C_d \frac{\rho V^2 A}{2}$$

Drag = coefficient x density x velocity squared x reference area  
two

Coefficient **Cd** contains all the complex dependencies  
and is usually determined experimentally.

Choice of reference area **A** affects the value of **Cd**.

Figure 3.2: Drag Equation

## 3.5 Normal Distribution

Normal distribution, also known as the Gaussian distribution, is a probability distribution that is symmetric around the mean, showing that data near the mean are more frequent in occurrence than data far from the mean. For all the normal distributions, 68.2% of the observations will happen +/- one standard deviation of the mean, 95.4% will appear within +/- two standard deviations and 99.7% within three as shown in the follow diagram.

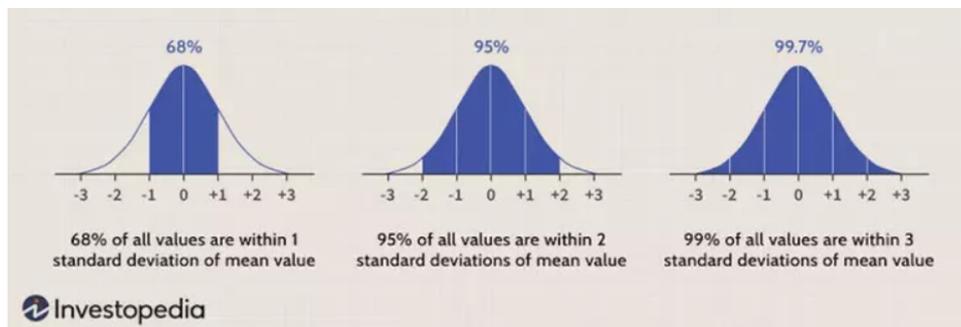


Figure 3.3: Normal Distribution Observations

The standard normal distribution has two parameters: the mean( $\mu$ ) and the standard deviation( $\sigma$ ) and it follows the following formula.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Figure 3.4: Normal Distribution Formula

### 3. TECHNOLOGICAL BACKGROUND AND DEFINITIONS

---

In this thesis normal distribution is used to implement the strength and duration of the pulses of the wind zones. The normal distribution, also known as the Gaussian distribution, is a probability distribution that is commonly used to model natural phenomena. It is a symmetric distribution that describes the variation of a random variable around its mean, as stated above, and many natural phenomena tend to follow this distribution. In the case of wind strength and period, the normal distribution can help to create more realistic and varied wind patterns. Wind speed and duration can vary widely over time, and the normal distribution can capture this variability by generating wind speeds and periods around a mean value with a certain degree of randomness.

By using the normal distribution to generate wind patterns a realistic and dynamic gameplay is created that keeps players engaged. Additionally, the balancing of the application is easier and ensures that it is fair and enjoyable for all players.

#### 3.6 Anaconda

Anaconda is a distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment. The distribution includes data-science packages suitable for different operating systems such as Windows, Linux, and macOS. In the thesis we used it to run Python in Windows for the training of the drone agents.

#### 3.7 PyTorch

PyTorch is a machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing, originally developed by Meta AI and now part of the Linux Foundation umbrella. It is free and open-source software released under the modified BSD license. In the thesis it was required in order to run the ML Agents Package (see below) that was used to train the drone agents.

## 3.8 ML Agents

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that allows developers to use games and simulations as environments for training intelligent agents. This toolkit provides PyTorch-based implementations of state-of-the-art algorithms, enabling developers to train intelligent agents for 2D, 3D, and VR/AR games with ease. Additionally, the toolkit offers a Python API that researchers can use to train agents using reinforcement learning, imitation learning, neuroevolution, or any other methods.

These trained agents can be used to control non-player character behavior in various settings such as multi-agent and adversarial environments, as well as for automated testing of game builds and evaluating different game design decisions pre-release. The ML-Agents Toolkit benefits both game developers and AI researchers by providing a central platform for evaluating AI advancements on Unity's diverse environments and making these advancements accessible to the broader research and game development communities.

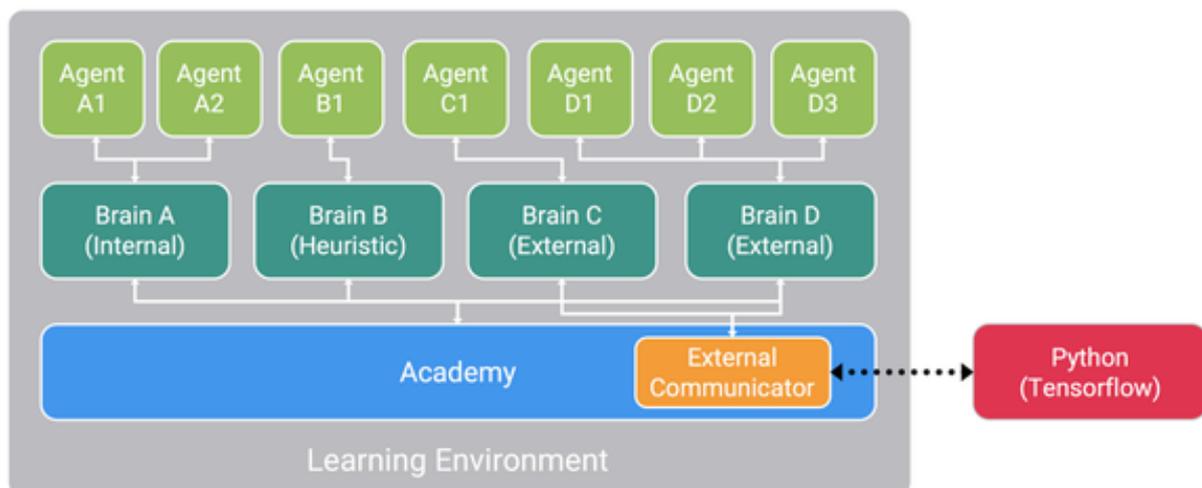


Figure 3.5: ML Agents Learning Environment

### 3. TECHNOLOGICAL BACKGROUND AND DEFINITIONS

---

The Python API that was mentioned above is crucial for defining agents, setting up environments, and running training and evaluation loops. Additionally, the ML-Agents Toolkit is built on top of the popular deep learning framework PyTorch, which is written in Python.

The three main kinds of objects within any Learning Environment are:

**Agent** - Each Agent has a unique set of states and observations, take unique actions based on the environment, and receive rewards for events within the environment. The agent's actions are decided by the brain it is linked to.

**Brain** - Each Brain has a defined state and action space and is responsible for determining the actions of its associated agents. The software currently provides four modes for setting up Brains:

- External mode: Action decisions are made through TensorFlow or another machine learning library by communicating with the Python API over an open socket.
- Internal mode: Action decisions are made using a trained model that is embedded into the project.
- Player mode: Action decisions are made based on player input.
- Heuristic mode: Action decisions are made based on hand-coded behavior.

**Academy** - In a scene, the Academy object holds all the Brains present within the environment as its children. Each environment has a single Academy, which defines the environment's scope in terms of:

- Engine Configuration: It determines the speed and rendering quality of the game engine in both training and inference modes.
- Frameskip: It specifies the number of engine steps that should be skipped before each agent makes a new decision.
- Global Episode Length: It determines the duration of the episode. Once it is reached, all agents are marked as done. [19]

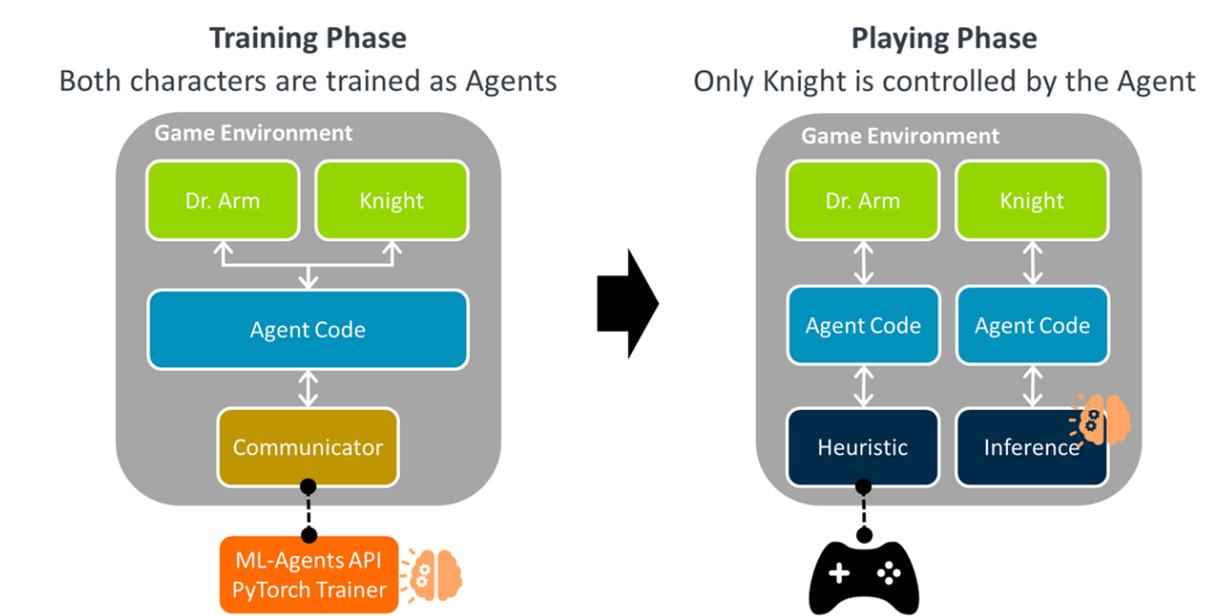


Figure 3.6: Two phases for agents training and gameplay

### 3. TECHNOLOGICAL BACKGROUND AND DEFINITIONS

---

# Chapter 4

## Use Case

### 4.1 Introduction

In this chapter we will go through the user's experience through the application and the different capabilities that he has to interact with its UIs. Use case diagrams will also be provided for every UI. The Main Menu will be explained, its canvases and the different parameters the user has to select before starting the game. Then the Pause Menu will be described as well as the ending screen of the game.

### 4.2 Main Menu Scene

The main menu is the entry scene the application. From there the user is able to select different parameters that will be used in the game as well as which stage he wants to be in.

## 4. USE CASE

---

### 4.2.1 Canvas Main Menu

The Main Menu canvas is the first canvas the user sees in the application. It contains a picture with drones and 4 buttons the player can select, the start, tutorial, credits and exit buttons. When the player clicks the start button, the function canvas switch is called and the canvas changes to Canvas Modes. If it clicks the tutorial button, then the tutorial scene opens. If it clicks the credits button then the canvas switch activates the canvas credits and if the exit button is pressed the application closes.

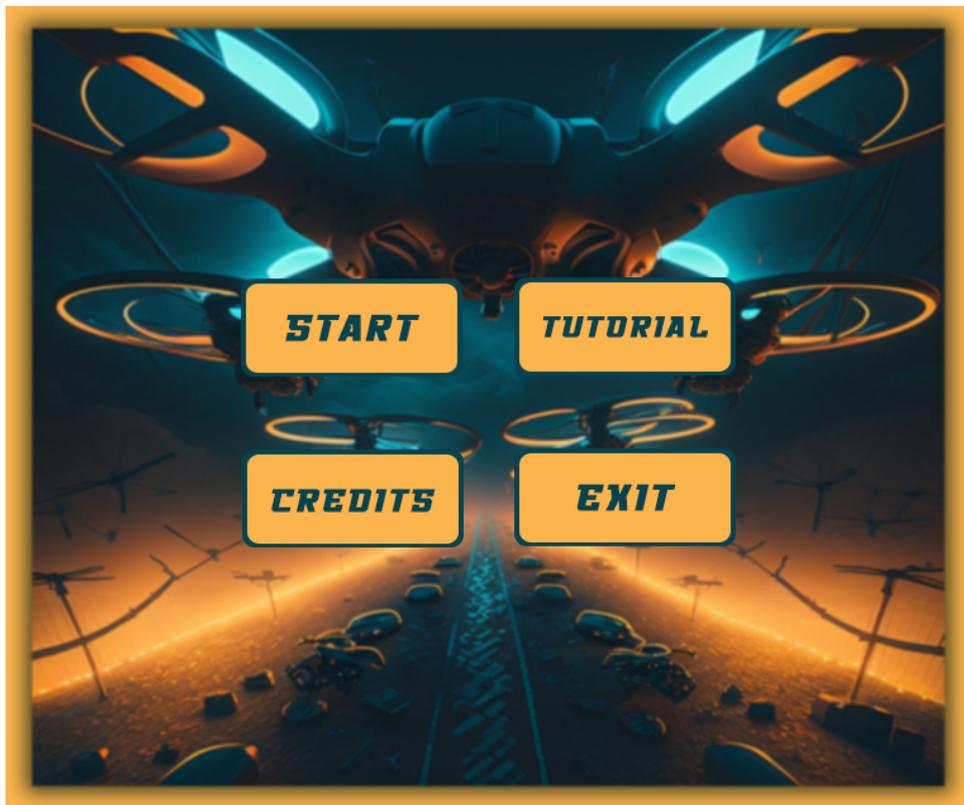


Figure 4.1: Canvas Main Menu

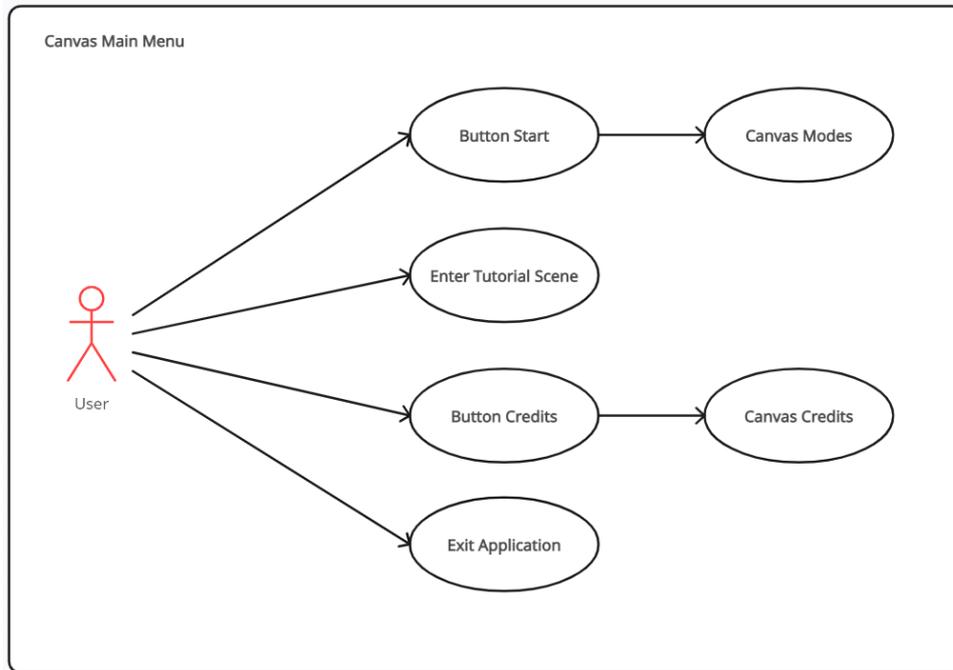


Figure 4.2: Canvas Main Menu Use Case

### 4.2.2 Canvas Modes

In the Canvas Modes that the user is redirected when clicks the Start button he can choose some important parameters in the application. He can select to play first or third person, so the camera is adjusted inside the game, solo or versus the trained drone agents and easy or hard mode that selects a different neural network brain for the drone agents the user competes. In easy mode, the agents pass the stage a bit more slow, while in hard more the drones almost always beat the user.

## 4. USE CASE

---

Canvas Modes:

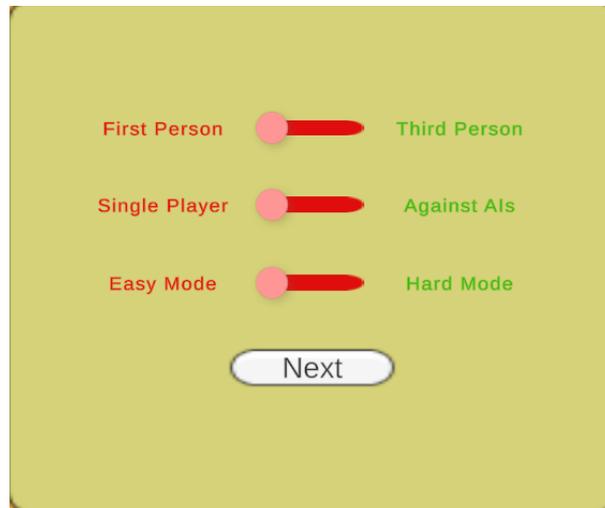


Figure 4.3: Canvas Modes

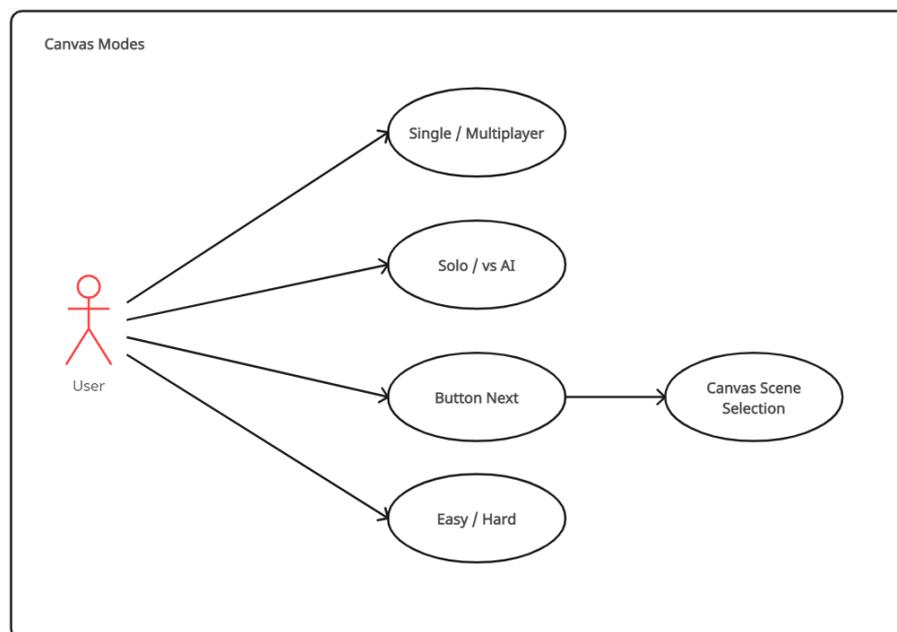


Figure 4.4: Canvas Modes Use Case

### 4.2.3 Canvas Scene Selection

In Canvas Scene Selection the user is able to pick which game scene wants to play in. There are three possible stages to pick.



Figure 4.5: Canvas Scene Selection

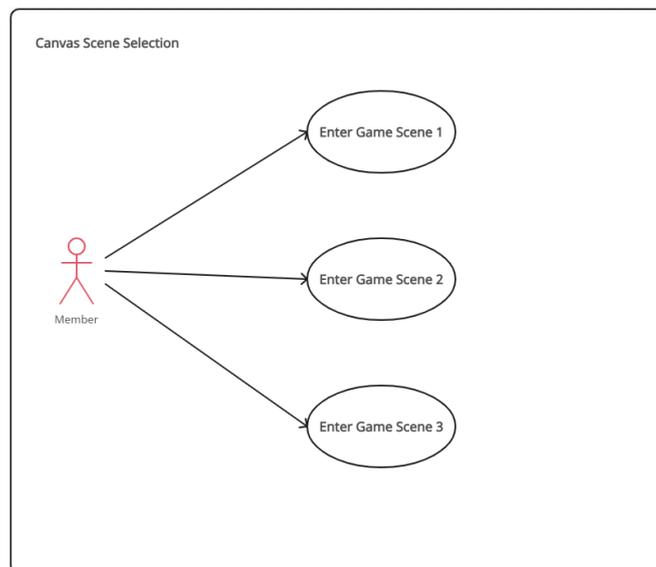


Figure 4.6: Scene Selection Use Case

## 4. USE CASE

---

### 4.3 Pause Menu

The pause Menu Opens when the users press the Left Hand Controller's primary button if they are in VR or the "P" button of the keyboard if they are using the PC and keyboard.

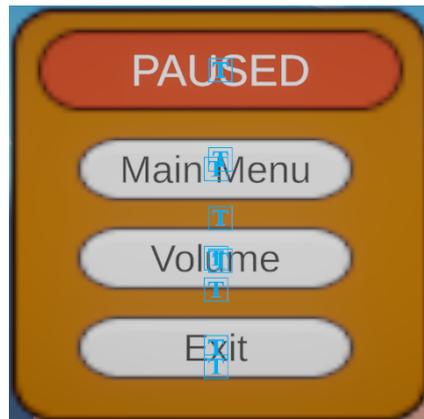


Figure 4.7: Pause Menu

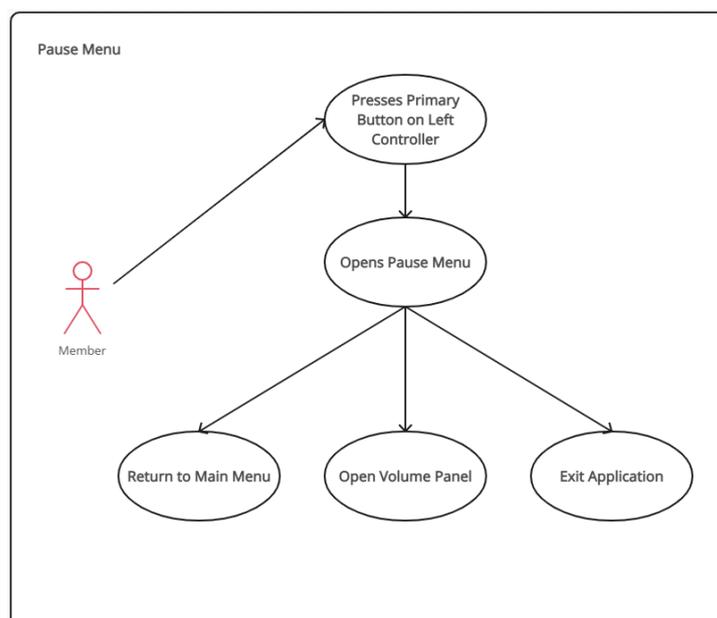


Figure 4.8: Pause Menu Use Case

## 4.4 Ending Panel

Lastly, the ending panel of our game appears in front of the player inside the application whenever the last checkpoint of a stage is passed and the player has passed all the other checkpoints beforehand. It shows how much time it took the user to pass the stage and has a restart button to restart the current stage and an exit button to close the application.



Figure 4.9: Ending Panel

#### 4. USE CASE

---

# Chapter 5

## Implementation

### 5.1 Introduction

In this chapter, the technical implementation of this project will be explained, based on the tools and the information that have been presented earlier. Most of the scripts will be analyzed with detail as well screenshots of some scripts will be provided.

The Integration in Oculus Quest will be explained as well as some key Unity Features such as Unity's new input system. Then we will move on to describe how the movement of the drone was implemented as well as how we implemented physics in our application. Details about the reasons we chose these 3 stages will be explained. The trigger event scripts will be analyzed as well the key manager scripts.

We conclude this chapter by explaining the whole process of training of the drone agents using the ML agents toolkit. We take a look on how to set the python environment and the configuration file and then we define the scripts for the drone area and the drone agents. Finally we make some observations from the training and see how we can extract the final neural network files after the training is over.

### 5.2 Oculus Integration

At the start of the thesis, in order to create VR projects that run in Oculus Quest, many steps were followed. The android Build Support was downloaded that contains the Android SDK & NDK Tools as well as the OpenJDK which are required in order to build a project in Oculus Quest. On the build Settings, the Max Texture Size was put as Max 2048 because the maximum texture that Oculus Quest is able to render is a 2k image. The XR Plug-in Management was setup along with Oculus XR and XR Interaction Toolkit that was needed to interact with the controllers inside the VR environment.

### 5.3 Player Input / Unity New Input System

In this thesis the new Unity Input System was used so that the player can handle the controls both in the VR headset as well as the keyboard and that made testing way easier. The system allows for the creating of action maps that the user can define specific actions (for example move action) and assign specific controls as bindings where when the specific control is triggered then an event is occurred which the user can later on code through the editor.

In this thesis we defined 4 of those input actions. One for each of the joysticks on the Oculus hand controllers, one for the left controller primary button and one for the right controller primary button. We also linked keyboard controls to these input actions for testing purposes.

At the Start function the Action Map is initialized and the input actions that we defined earlier were linked to the actions of the input map. When an action is performed a function is called that keeps the Vector data of the joystick in a variable, or enables a boolean variable when the 2 buttons are pressed. These variables then can be used in the scripts depending on what we want to achieve. In this thesis we assigned them into a moveVector that is responsible for the forward motion of the drone as well as making it tilt, a turnVector that helps the drone go up, down and rotate, a button that changes the mode and a button that opens the pause menu as can be seen in the following figure.

```

2 references
public void OnMovementChanged(InputAction.CallbackContext context){
    Vector2 direction = context.ReadValue<Vector2>();
    moveVector = new Vector3(direction.x, 0 ,direction.y);
}
2 references
public void OnTurnChanged(InputAction.CallbackContext context1){
    Vector2 direction1 = context1.ReadValue<Vector2>();
    turnVector = new Vector3(direction1.x, 0 ,direction1.y);
}
1 reference
public void OnButtonChanged(InputAction.CallbackContext context2){
    modeSwitcher = true;
}

```

Figure 5.1: Input Action Functions

## 5.4 Drone Movement

A big part of this thesis research was to implement a system that represents a realistic movement of drone controls. The goal was to make flying the drone in the simulator feel like you are flying a real drone. In the script some functions were implemented to achieve this goal. These functions are the MovementUpDown(), MovementForward(), Rotation(), ClampingSpeedValuesModeA() and Swerve() that are able to identify the force and the rotation that the drone has every frame depending on the inputs the user gives it using the Input Actions that we talked previously.

```

private void FixedUpdate(){
    MovementUpDown();
    MovementForward();
    Rotation();
    //ClampingSpeedValues();
    if(SwitchMode.instance.currentDroneMode == SwitchMode.DroneMode.ModeA){ //Mode A
        ClampingSpeedValuesModeA();
        //Debug.Log("I am here");
    }
    Swerve();

    drone.AddRelativeForce(Vector3.up*upForce);
    drone.rotation = Quaternion.Euler(
        new Vector3(tiltAmountForward, currentYRotation, tiltAmountSideways)
    );
}

```

Figure 5.2: Drone Movement Overview

## 5. IMPLEMENTATION

---

**MovementUpDown():** In this function the vertical Movement of the drone was implemented. If the user is moving forward or bending (will be mentioned later) then the user can use the joystick of the right controller to move the drone upwards or downwards if it moves it on the vertical axis or give a small upwards force if the drone is rotating or bending or just moving forward.

```
9 references
public float upForce;
1 reference
void MovementUpDown(){

    if(Mathf.Abs(Input.GetAxis("Vertical")) > .2f || Mathf.Abs(Input.GetAxis("Horizontal")) > .2f || Mathf.Abs(moveVector.z) > .2f || Mathf.Abs(moveVector.x) > .2f) {
        if (Input.GetKey(KeyCode.I) || Input.GetKey(KeyCode.K) || (Mathf.Abs(turnVector.z) > .2f)){
            drone.velocity = drone.velocity;
        }
        if (!Input.GetKey(KeyCode.I) && !Input.GetKey(KeyCode.K) && !Input.GetKey(KeyCode.J) && !Input.GetKey(KeyCode.L) && Mathf.Abs(turnVector.z) < .2f && Mathf.Abs(turnVector.x) < .2f) {
            drone.velocity = new Vector3(drone.velocity.x, Mathf.Lerp(drone.velocity.y, 0, Time.deltaTime * 5), drone.velocity.z);
            upForce = 281;
        }
        if (!Input.GetKey(KeyCode.I) && !Input.GetKey(KeyCode.K) && Mathf.Abs(turnVector.z) < .2f && !Input.GetKey(KeyCode.J) || Input.GetKey(KeyCode.L) || turnVector.x > 0){
            drone.velocity = new Vector3(drone.velocity.x, Mathf.Lerp(drone.velocity.y, 0, Time.deltaTime * 5), drone.velocity.z);
            upForce = 110;
        }
        if (Input.GetKey(KeyCode.J) || Input.GetKey(KeyCode.L) || Mathf.Abs(turnVector.x) > .2f) {
            upForce = 410;
        }
    }
    if ((Mathf.Abs(Input.GetAxis("Vertical")) < .2f && Mathf.Abs(Input.GetAxis("Horizontal")) > .2f) || (Mathf.Abs(moveVector.z) < .2f && Mathf.Abs(moveVector.x) > .2f)){
        upForce = 135;
    }

    if (Input.GetKey(KeyCode.I) || turnVector.z > .2f){
        //upForce = 450;
        drone.velocity = drone.velocity;
        upForce = 650;
        //Debug.Log("Im going up!");
        if (Mathf.Abs(Input.GetAxis("Horizontal")) > .2f || Mathf.Abs(moveVector.x) > .2f) {
            //upForce = 500;
            upForce = 700;
        }
    } else if (Input.GetKey(KeyCode.K) || turnVector.z < -.2f) {
        //upForce = -200;
        upForce = -400;
    } else if (!Input.GetKey(KeyCode.K) && !Input.GetKey(KeyCode.I) && Mathf.Abs(Input.GetAxis("Vertical")) < .2f && Mathf.Abs(Input.GetAxis("Horizontal")) < .2f && Mathf.Abs(moveVector.z) < .2f
    ) {
        //upForce = 98.1f;
        upForce = 0f;
        //Debug.Log("im here");
    }
    else
    {

```

Figure 5.3: MovementUpDown

**MovementForward():** Depending on the mode of the Drone, when the player uses the left control joystick it can move the drone forward or backward. There is also a tilt implemented to the drone because, as we saw in the physics chapter, in order to move the drone forward the two back proppelers need to move with higher speed than the two front ones resulting in the drone tilting and providing forward motion.

```

2 references
private float movementForwardSpeed = 750.0f;
2 references
private float movementForwardSpeedModeB= 70.0f;
11 references
private float tiltAmountForward = 0;
5 references
private float tiltVelocityForward;
1 reference
void MovementForward(){
    if((Input.GetAxis("Vertical") > 0 || Input.GetAxis("Vertical") < 0) && SwitchMode.instance.currentDroneMode == SwitchMode.DroneMode.ModeA) {
        drone.AddRelativeForce(Vector3.forward * Input.GetAxis("Vertical") * movementForwardSpeed);
        tiltAmountForward = Mathf.SmoothDamp(tiltAmountForward, 20 * Input.GetAxis("Vertical"), ref tiltVelocityForward, 0.1f);
    }
    else if((Input.GetAxis("Vertical") > 0 || Input.GetAxis("Vertical") < 0) && SwitchMode.instance.currentDroneMode == SwitchMode.DroneMode.ModeB) {
        drone.AddRelativeForce(Vector3.forward * Input.GetAxis("Vertical") * movementForwardSpeedModeB);
        tiltAmountForward = Mathf.SmoothDamp(tiltAmountForward, 20 * Input.GetAxis("Vertical"), ref tiltVelocityForward, 0.1f);
    }
    else if((moveVector.z > 0 || moveVector.z < 0) && SwitchMode.instance.currentDroneMode == SwitchMode.DroneMode.ModeA) { //testing input system
        drone.AddRelativeForce(Vector3.forward * moveVector.z * movementForwardSpeed); // make it a bit slower in VR
        tiltAmountForward = Mathf.SmoothDamp(tiltAmountForward, 20 * moveVector.z, ref tiltVelocityForward, 0.1f);
    }
    else if((moveVector.z > 0 || moveVector.z < 0) && SwitchMode.instance.currentDroneMode == SwitchMode.DroneMode.ModeB) { //testing input system
        drone.AddRelativeForce(Vector3.forward * moveVector.z * movementForwardSpeedModeB); // make it a bit slower in VR
        tiltAmountForward = Mathf.SmoothDamp(tiltAmountForward, 20 * moveVector.z, ref tiltVelocityForward, 0.1f);
    }
    else {
        tiltAmountForward = Mathf.SmoothDamp(tiltAmountForward, 0, ref tiltVelocityForward, .1f);
    }
}
}

```

Figure 5.4: MovementForward

**Rotation():** In this function, when the user uses the right controller’s joystick and moves it horizontally, the yaw of the drone is adjusted making it to rotate left or right by a given rotation amount. In order to not confuse the system of going up or rotating when the user moves the joystick diagonally, we defined that if the axis is below 0.85 out of maximum 1 then we consider that the user wants to rotate instead of going up.

```

private float rotateAmountByKeys = 2f;
//private float rotateAmountByKeys = 2.5f;
1 reference
private float rotationYVelocity;
1 reference
void Rotation(){
    if(Input.GetKey(KeyCode.J) || (turnVector.x < -.2f && Mathf.Abs(turnVector.z) < .85f)){
        wantedYRotation -= rotateAmountByKeys;
    }
    if(Input.GetKey(KeyCode.L) || (turnVector.x > .2f && Mathf.Abs(turnVector.z) < .85f)){
        wantedYRotation += rotateAmountByKeys;
    }
    currentYRotation = Mathf.SmoothDamp(currentYRotation, wantedYRotation, ref rotationYVelocity, .25f);
}
}

```

Figure 5.5: Rotation

## 5. IMPLEMENTATION

---

**Swerve():** In this function, when the user uses the left controller's joystick and moves it horizontally, the roll of the drone is adjusted making it to bend left or right and also starts moving left or right by a given amount.

```
2 references
private float sideMovementAmount = 300.0f;
2 references
private float sideMovementAmountModeB = 70.0f;
11 references
private float tiltAmountSideways;
5 references
private float tiltAmountVelocity;
1 reference
void Swerve() {
    if(Mathf.Abs(Input.GetAxis("Horizontal")) > .2f && SwitchMode.instance.currentDroneMode == SwitchMode.DroneMode.ModeA){
        drone.AddRelativeForce(Vector3.right*Input.GetAxis("Horizontal")*sideMovementAmount);
        tiltAmountSideways = Mathf.SmoothDamp(tiltAmountSideways, -20 * Input.GetAxis("Horizontal"), ref tiltAmountVelocity, .1f);
    }
    else if(Mathf.Abs(Input.GetAxis("Horizontal")) > .2f && SwitchMode.instance.currentDroneMode == SwitchMode.DroneMode.ModeB){
        drone.AddRelativeForce(Vector3.right*Input.GetAxis("Horizontal")*sideMovementAmountModeB);
        tiltAmountSideways = Mathf.SmoothDamp(tiltAmountSideways, -20 * Input.GetAxis("Horizontal"), ref tiltAmountVelocity, .1f);
    }
    else if(Mathf.Abs(moveVector.x) > .2f && SwitchMode.instance.currentDroneMode == SwitchMode.DroneMode.ModeA){
        drone.AddRelativeForce(Vector3.right*moveVector.x*sideMovementAmount);
        tiltAmountSideways = Mathf.SmoothDamp(tiltAmountSideways, -20 * moveVector.x, ref tiltAmountVelocity, .1f);
    }
    else if(Mathf.Abs(moveVector.x) > .2f && SwitchMode.instance.currentDroneMode == SwitchMode.DroneMode.ModeB){
        drone.AddRelativeForce(Vector3.right*moveVector.x*sideMovementAmountModeB);
        tiltAmountSideways = Mathf.SmoothDamp(tiltAmountSideways, -20 * moveVector.x, ref tiltAmountVelocity, .1f);
    }
    else {
        tiltAmountSideways = Mathf.SmoothDamp(tiltAmountSideways, 0, ref tiltAmountVelocity, .1f);
    }
}
```

Figure 5.6: Swerve

There were 2 modes of drone controls implemented that are gonna be explained next. The switch of the mode happens by pressing the primary button of the right controller.

### 5.4.1 Mode A

In this mode, the drone holds its position in the air whenever the user doesn't use any of the inputs, more details on how it does this will be explained in the Physics section. It also moves with a standard speed that is previously defined and whenever the user stops using the controls the drone automatically stops in the air and try to remain in its current position.

**ClampingSpeedValuesModeA():** In this mode another script was implemented to make the speed of the drone clamp at a desired speed in a short passage of time to feel more realistic.

```
void ClampingSpeedValuesModeA()
{
    if(Mathf.Abs(Input.GetAxis("Vertical")) > .2f && Mathf.Abs(Input.GetAxis("Horizontal")) > .2f) {
        drone.velocity = Vector3.ClampMagnitude(drone.velocity, Mathf.Lerp(drone.velocity.magnitude, 10.0f, Time.deltaTime * 5f));
        //drone.velocity = Vector3.zero;
    }
    if(Mathf.Abs(moveVector.z) > .2f && Mathf.Abs(moveVector.x) > .2f) {
        drone.velocity = Vector3.ClampMagnitude(drone.velocity, Mathf.Lerp(drone.velocity.magnitude, 23.0f, Time.deltaTime * 5f));
        //drone.velocity = Vector3.zero;
    }
    if(Mathf.Abs(Input.GetAxis("Vertical")) > .2f && Mathf.Abs(Input.GetAxis("Horizontal")) < .2f){
        drone.velocity = Vector3.ClampMagnitude(drone.velocity, Mathf.Lerp(drone.velocity.magnitude, 10.0f, Time.deltaTime * 5f));
        //drone.velocity = Vector3.zero;
    }
    if(Mathf.Abs(moveVector.z) > .2f && Mathf.Abs(moveVector.x) < .2f){
        drone.velocity = Vector3.ClampMagnitude(drone.velocity, Mathf.Lerp(drone.velocity.magnitude, 23.0f, Time.deltaTime * 5f));
        //drone.velocity = Vector3.zero;
    }
    if(Mathf.Abs(Input.GetAxis("Vertical")) < .2f && Mathf.Abs(Input.GetAxis("Horizontal")) > .2f){
        drone.velocity = Vector3.ClampMagnitude(drone.velocity, Mathf.Lerp(drone.velocity.magnitude, 10.0f, Time.deltaTime * 5f));
        //drone.velocity = Vector3.zero;
    }
    if(Mathf.Abs(moveVector.z) < .2f && Mathf.Abs(moveVector.x) > .2f){
        drone.velocity = Vector3.ClampMagnitude(drone.velocity, Mathf.Lerp(drone.velocity.magnitude, 23.0f, Time.deltaTime * 5f));
        //drone.velocity = Vector3.zero;
    }
    if(Mathf.Abs(Input.GetAxis("Vertical")) < .2f && Mathf.Abs(Input.GetAxis("Horizontal")) < .2f && Mathf.Abs(moveVector.z) < .2f &&
    Mathf.Abs(moveVector.x) < .2f && !Input.GetKey(KeyCode.I) && !Input.GetKey(KeyCode.K) && Mathf.Abs(turnVector.z) < .2f) {
        //drone.velocity = Vector3.SmoothDamp(drone.velocity, Vector3.zero, ref velocityToSmoothDampingToZero, .95f);
        drone.velocity = Vector3.zero;
        stoppedYPos = drone.position.y;
    }
}
```

Figure 5.7: ClampingSpeedValuesModeA

### 5.4.2 Mode B

In this mode the movement happens depending on how much pitch has the drone in each directions. The bigger the angle the faster the drone moves. The drone stops by it's drag force that is applied to it which is gonna be explained furthermore in the physics section.

## 5. IMPLEMENTATION

---

### 5.5 Physics

In this chapter many elements are introduced that made the application physics-based. The physics equations that were used are presented in the Chapter 3 sub sections. They present different physics laws and principles.

#### 5.5.1 State Detection

A State Detection script was needed in order to determine the different parameters of the drone movement that we presented in Chapter 2.2.4 which are required for the flight of the drone. More specifically the Pitch, the Roll and the Yaw of the drone each frame were recorded in radians. The altitude of the drone is also defined as well as the Velocity and Angular Velocity Vectors in local space. All these parameters are then used in different scripts. There is also a reset function that initializes again the values as 0.

```
2 references
public void GetState() {

    Vector3 worldDown = vm.transform.InverseTransformDirection (Vector3.down);
    float Pitch = worldDown.z; // Small angle approximation
    float Roll = -worldDown.x; // Small angle approximation
    float Yaw = vm.transform.eulerAngles.y;

    Angles = new Vector3 (Pitch, Yaw, Roll);

    Altitude = vm.transform.position.y;

    VelocityVector = vm.transform.GetComponent<Rigidbody> ().velocity;
    VelocityVector = vm.transform.InverseTransformDirection (VelocityVector);

    AngularVelocityVector = vm.transform.GetComponent<Rigidbody> ().angularVelocity;
    AngularVelocityVector = vm.transform.InverseTransformDirection (AngularVelocityVector);

}

1 reference
public void Reset() {
    flag = true;
    VelocityVector = Vector3.zero;
    AngularVelocityVector = Vector3.zero;
    Angles = Vector3.zero;
    Altitude = 0.0f;

    enabled = true;
}
```

Figure 5.8: State Detection

## 5.5.2 Velocity Manager

This script was essential for the Mode A of the drone that requires it to stay in a specific position when it is not moving. In this thesis other forces might affect the drone such as wind, that is going to be explained later on, and a system was needed that helps the drone hold its position and rotation physically. The script Velocity Manager was influenced by a research done at university of Berkeley which were working on the physics of drones. At the start the script first adds force to the drone equal to its weight = gravity x mass that is the required lift the drone need to have to counteract its weight force.

```
void Start () {
    state.GetState ();
    Rigidbody rb = GetComponent<Rigidbody> ();
    Vector3 desiredForce = new Vector3 (0.0f, gravity * state.Mass, 0.0f);
    rb.AddForce (desiredForce, ForceMode.Acceleration);
}
```

Figure 5.9: Velocity Manager Initialization

Then, when the application is running, each time the drone stops (no inputs are given) on Mode A, a desired height is calculated at the position the drone stopped. If any force is applied to the drone during that period and makes it move from its current position then the script acts in order to assist the drone to regain its desired position. Firstly, the height error is calculated as well as the velocity error which is defined as the desired velocity minus the current state velocity. Then the desired acceleration is calculated that is needed to define the desired Thrust we need to apply to the drone as well some desired angles that are needed in order to defined the Torque that we need to give the drone. By that, we managed to implement a system and help the drone regain its position and lift in the air without having to worry about which other forces are applied to it at that moment.

## 5. IMPLEMENTATION

---

```
void FixedUpdate () {
    state.GetState ();
    //set the desired height as much as the drone y pos when it stopped
    desired_height = DroneMovement._Instance.stoppedYPos;

    Vector3 desiredTheta;
    Vector3 desiredOmega;
    (field) float VelocityManager.desired_height

    float heightError = state.Altitude - desired_height + 3.27f;

    Vector3 desiredVelocity = new Vector3 (desired_vy, -1.0f * heightError / time_constant_z_velocity, desired_vx);
    Vector3 velocityError = state.VelocityVector - desiredVelocity;

    Vector3 desiredAcceleration = velocityError * -1.0f / time_constant_acceleration;

    desiredTheta = new Vector3 (desiredAcceleration.z / gravity, 0.0f, -desiredAcceleration.x / gravity);
    if (desiredTheta.x > max_pitch) {
        desiredTheta.x = max_pitch;
    } else if (desiredTheta.x < -1.0f * max_pitch) {
        desiredTheta.x = -1.0f * max_pitch;
    }
    if (desiredTheta.z > max_roll) {
        desiredTheta.z = max_roll;
    } else if (desiredTheta.z < -1.0f * max_roll) {
        desiredTheta.z = -1.0f * max_roll;
    }

    Vector3 thetaError = state.Angles - desiredTheta;

    desiredOmega = thetaError * -1.0f / time_constant_omega_xy_rate;
    desiredOmega.y = desired_yaw;

    Vector3 omegaError = state.AngularVelocityVector - desiredOmega;

    Vector3 desiredAlpha = Vector3.Scale(omegaError, new Vector3(-1.0f/time_constant_alpha_xy_rate, -1.0f/time_constant_alpha_z_rate, -1.0f/time_constant_alpha_xy_rate));
    desiredAlpha = Vector3.Min (desiredAlpha, Vector3.one * max_alpha);
    desiredAlpha = Vector3.Max (desiredAlpha, Vector3.one * max_alpha * -1.0f);

    float desiredThrust = (gravity + desiredAcceleration.y) / (Mathf.Cos (state.Angles.z) * Mathf.Cos (state.Angles.x));
    desiredThrust = Mathf.Min (desiredThrust, 2.0f * gravity);
    desiredThrust = Mathf.Max (desiredThrust, 0.0f);

    Vector3 desiredTorque = Vector3.Scale (desiredAlpha, state.Inertia);
    Vector3 desiredForce = new Vector3 (0.0f, desiredThrust * state.Mass, 0.0f);

    Rigidbody rb = GetComponent<Rigidbody>();

    rb.AddRelativeTorque (desiredTorque, ForceMode.Acceleration);
    rb.AddRelativeForce (desiredForce, ForceMode.Acceleration);
}
```

Figure 5.10: Velocity Manager

### 5.5.3 Drag

As it was mentioned in Chapter 2, drag is one of the most important forces that act in every object with mass in the opposite direction of its motion due to air resistance. In the unity Editor there is drag implemented but this is not physics-based and is put manually. In our application we wanted the drag to be based on the physics and the drag equation that was presented in chapter 3.4. There was an approximation made for the aerodynamic Coefficient that was used, and is based on the shape of the propellers. The drag force is then applied to the drone and it is what stops it from moving in Mode B but is also applied in mode A as a resistance force that is opposed to the motion.

```
1 reference
public Vector3 CalculateDrag(){
    Vector3 relativePosition = transform.position - rb.worldCenterOfMass;
    Vector3 worldAirVelocity = -rb.velocity - Vector3.Cross(rb.angularVelocity,relativePosition);
    //Vector3 worldAirVelocity = -rb.velocity;
    Vector3 airVelocity = transform.InverseTransformDirection(worldAirVelocity);
    //airVelocity = new Vector3(airVelocity.x, airVelocity.y);

    Vector3 dragDirection = transform.TransformDirection(airVelocity.normalized);
    //Vector3 dragDirection = Vector3.one;
    float area = 1 * 1; // span * chord
    float dynamicPressure = 0.5f * airDensity * airVelocity.sqrMagnitude;
    float aerodynamicCoefficients = 0.06f;
    Vector3 DragForce = dragDirection * aerodynamicCoefficients * dynamicPressure * area;
    return DragForce;
}
```

Figure 5.11: Drag Calculator

## 5. IMPLEMENTATION

---

### 5.5.4 Random Wind

Another very important aspect to make flying of drones realistic is wind and how it's force can effect the drone's movement. In this thesis we implemented a random wind script that creates wind zones in our game. Each wind zone applies a wind force to the drone with different parameters. First the pulse period is decided and then its duration and strength. Then the direction is decided, the speed of its change. In the end a ray is produced that adds instant force to the drone using its mass (ForceMode.Impulse).

As we mentioned in Chapter 3.5 in order to make the wind feel more realistic we implemented Gaussian distribution in it using the sampling method. In statistics and probability theory, the Gaussian or normal distribution is one of the most commonly used distributions to model many real-world phenomena. When we sample from a Gaussian distribution, we are generating a set of observations that follow a normal distribution with a particular mean and standard deviation. Specifically, each sample point is drawn from the Gaussian distribution according to its probability density function, which is a mathematical expression that describes the likelihood of observing a particular value. It is a symmetric distribution that describes the variation of a random variable around its mean and the above parameters that we mentioned earlier can vary widely over time so the normal distribution can simulate it by generating these wind parameters around a mean value with a certain degree of randomness.

```
public float Sample(float mean, float var)
{
    float n = NextGaussianDouble();

    return n * Mathf.Sqrt(var) + mean;
}

5 references
public float SamplePositive(float mean, float var) {
    return Mathf.Abs(Sample(mean, var));
}

public float NextGaussianDouble()
{
    float u, v, S;

    do
    {
        u = 2.0f * (float) r.NextDouble() - 1.0f;
        v = 2.0f * (float) r.NextDouble() - 1.0f;
        S = u * u + v * v;
    }
    while (S >= 1.0f);

    float fac = Mathf.Sqrt(-2.0f * Mathf.Log(S) / S);
    return u * fac;
}
```

Figure 5.12: Sampling Script

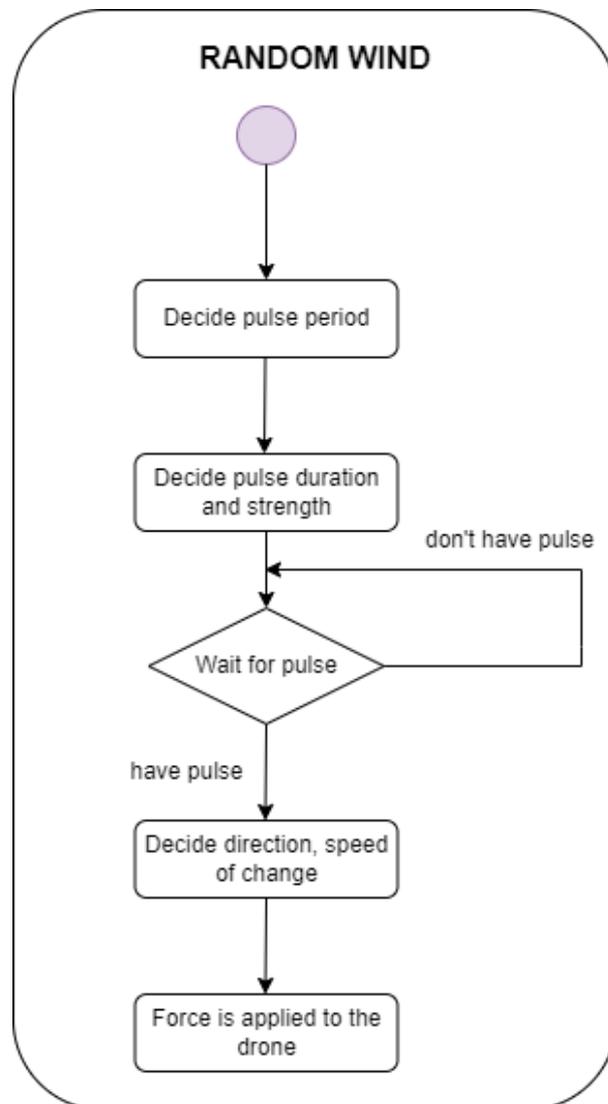


Figure 5.13: Random Wind Diagram

## 5. IMPLEMENTATION

---

### 5.6 3D Environments

In the thesis many different virtual environments were implemented that simulate two different real time scenarios and a more simple scenario that helps the players train easier.

#### 5.6.1 Tutorial Scene

A tutorial scene was essential to explain the controls to the user. The base scene is that of Stage 2. A menu shows up when the player enters the tutorial scene that follows it wherever it looks. The menu shows the controls to the user in a small rectangular graphic and has a proceed button. The rays of the joysticks automatically close whenever the user presses the button and the player is now dove in the stage. The task the player needs to complete is pass the 2 checkpoints that are in front of him. By doing that, the player can understand the controls that were shown to him and have a first glance of the gameplay. After the player passes the second checkpoint, a menu appears again that redirects him to the main menu scene.



Figure 5.14: Tutorial Menu

During the tutorial the user is playing in a third person view that helps him understand better the controls by watching the drone's movement and reactions visually.



Figure 5.15: Tutorial Scene

## 5. IMPLEMENTATION

---

### 5.6.2 Stage 1

This scene simulates many different environments. It has an urban environment with a beach, some sub-urban fields, mountains and a forest. In the mountainous area wind zones are implemented that apply wind force to the drone. With this stage the user can get a glimpse of all the small environments it can fly the drone. The player has to control and maneuver the drone through checkpoints that are created in the stage with the help of Cinemachine Smooth Path. Fog is also implemented in the environment to make it more realistic and block the vision of the player on faraway objects.



Figure 5.16: Stage 1

### 5.6.3 Stage 2

This scene simulates a sea environment with a lot of small and medium islands. In this scene we wanted to use the windzones more and see how they effect the race. The checkpoints are created by the same way of stage 1.



Figure 5.17: Stage 2.1



Figure 5.18: Stage 2.2

## 5. IMPLEMENTATION

---

### 5.6.4 Stage 3

This stage is made to help the user train more effectively. It is consisted with simple gameobjects, rectangles, cubes and capsules and it only uses 2 colors so that the player doesn't feel distracted when training. It is perfect for the user who just want to maneuver around and try to pass the stage. This stage was created from the feedback of a user that wanted something more simple and not so destructing to help strengthen the training process.

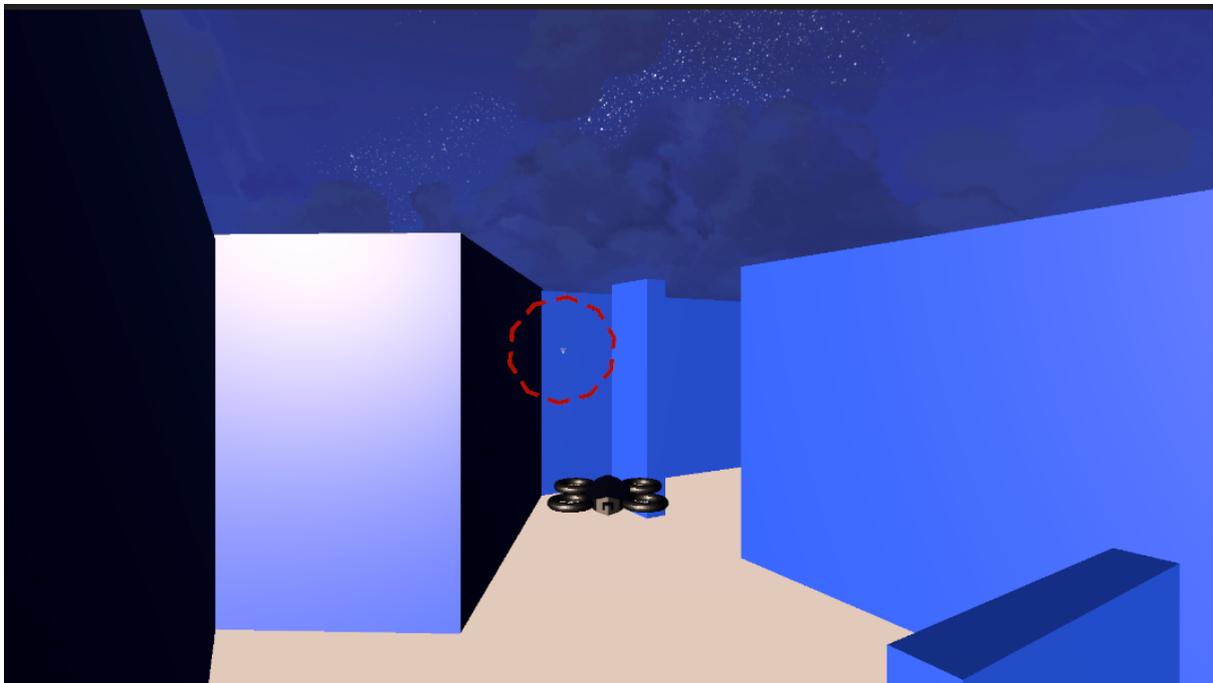


Figure 5.19: Stage 3

## 5.7 Trigger Events

Trigger events were very crucial in the thesis in order for the system to understand whenever the drone passes a checkpoint or hits into an object and has to restart.

### 5.7.1 CheckPointPass

A checkpoint has an invisible collider inside it that whenever the drone hits it the `OnTriggerEnter()` function is activated. There are 2 types of checkpoints that are defined with an enumeration, the regular checkpoint and the finish line checkpoint. Also, another enumerator defines if the checkpoints are passed or not passed. When the drone agents hit a checkpoint an audio effect is played, some particles pop up and the material of the checkpoint is changed to a different color and the enumerator of the checkpoint is switched from not passed to passed indicating that the player has passed that checkpoint. There is also a counter that keeps how many checkpoints are passed.

If the player passes the last checkpoint, we check if he has passed all the previous in order to prevent players going directly to the finish. The material changes to a different color than before indicating the passage of the last checkpoint, a particle system is played and the ending menu opens after one second. There is also a counter that keeps the time the player spent completing the race and it is presented to him in the ending menu.



Figure 5.20: Checkpoint

## 5. IMPLEMENTATION

---

### 5.7.2 RestartOnCollision

Another important feature in our application was to restart the race whenever the user controlled drone hit a terrain element. It is very important to have this feature because in real time conditions whenever the user trains to fly the drone it risk the danger of it to be damaged, with this risk increasing in high speed scenarios.

In the script that was implemented we check if the drone collides with elements with the tag "terrain" or "terrain elements" that were assigned to their respectable prefabs. Whenever it collides, the scene restarts and the user has to try to pass the stage again from the beginning.

## 5.8 Game Manager Scripts

In the application some manager scripts were implemented using the singleton design pattern. The singleton class provides a global access point for other scripts to get the instance of the class.

### 5.8.1 Sound Manager

The first Manager we had to implement was Sound Manager. It is responsible for all the sound effects and audio in the application. It contains an enumeration with the different sound types. The audio source and the audio clips are inserted by the inspector. There are two functions in the script, one to play audio and one to stop audio. The `audioToPlay` function gets a sound type enumerator and plays the audio source that is assigned to it. The stop audio function was used for the wind zones because the audio that played was in loop and we wanted whenever the player exits the windzone the audio to stop playing.

### 5.8.2 Canvas Manager

Another very important script in terms of User Interface and especially in the Main Menu was the Canvas Selector. It uses the singleton logic as well and it stores the canvases into a List. The script has a `SwitchCanvas` function which takes a canvas type enumerator and makes a canvas switch to a new desired canvas when the user calls the function, usually by pressing a button.

### 5.8.3 Panel Manager

A similar procedure is followed in the Panel Manager Script. Each Panel has an enumerator in it and when the choose panel function is called the panel switches to the desired panel that has the specified enumerator. This manager was crucial for the implementation of the pause/settings menu in the application where different panels needed to open in a specific canvas.

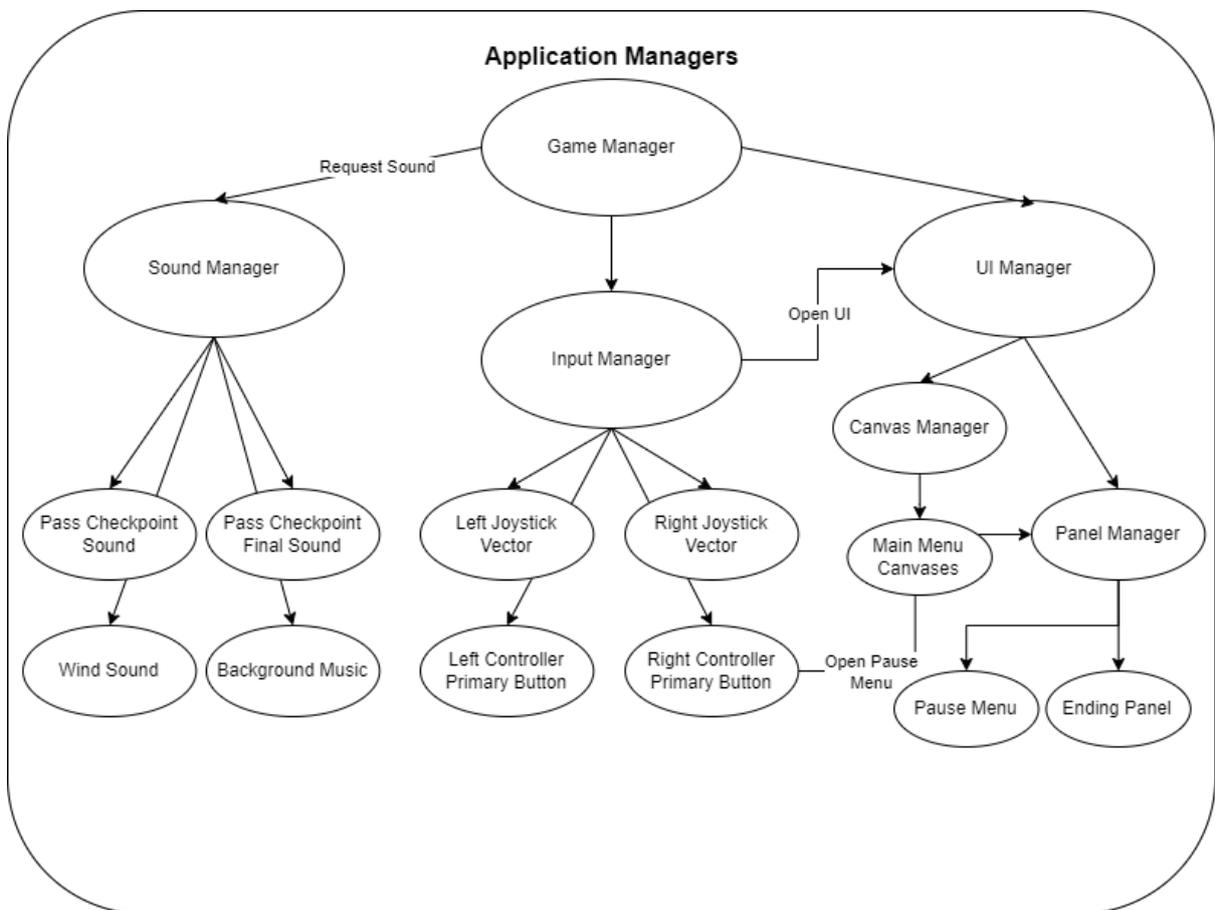


Figure 5.21: Application Managers Diagram

## 5. IMPLEMENTATION

---

### 5.9 Setting Up Python Environment

In order to train our agents a python environment needed to be set up as it was mentioned in Chapter 3.

#### 5.9.1 Anaconda

The Anaconda environment was set up in order to run python in windows. The anaconda environment can be found in this website. [20] A new python environment was created using the anaconda prompt and the command "conda create -n myenv python=3.9". The python version needs to be compatible with the pytorch package that will be explained later.

#### 5.9.2 Pytorch

PyTorch is a machine learning framework based on the Torch library that is required to run the ML Agents Package, as described in chapter 3.6. It has some specifications in order to run that can be found in the following graph as well as its installation info. The installation has to happen in the folder that the new environment was created. The

PyTorch Build	Stable (2.0.0)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 11.7	CUDA 11.8	ROCm-5.4.2	CPU
Run this Command:	<pre>pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu117</pre>			

Figure 5.22: Pytorch Installation

installation instructions can be found in this link. [21]

### 5.9.3 ML Agents Toolkit

The ML Agents package that was required for the training was downloaded through the unity package manager. The package documentation and installation requirements can be found in this link. [22]

A training configuration file (.yaml) is also needed in order to run the training process and define the parameters of the training. The first one is the trainer type which was set as ppo (Proximal Policy Optimization) that is a policy gradient method for reinforcement learning. Then some **hyperparameters** were set.

- **batch size:** It is the number of experiences in each iteration of gradient descent. This should always be multiple times smaller than buffer size that will be mentioned later. In this project we used continuous actions so this value should be large (on the order of 1000s). In our case we set it at 2048.
- **buffer size:** In PPO it is the number of experiences to collect before updating the policy model. Corresponds to how many experiences should be collected before we do any learning or updating of the model. We set it to 20480.
- **beta:** It is the strength of the entropy regularization, which makes the policy "more random." This ensures that agents properly explore the action space during training. Increasing this will ensure more random actions are taken. We left it at the typical range of 1.0e-2.
- **epsilon:** It influences how rapidly the policy can evolve during training. Corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating. Setting this value small will result in more stable updates, but will also slow the training process. We set this to default 0.2.
- **numepoch:** It is the number of passes to make through the experience buffer when performing gradient descent optimization. The larger the batch size, the larger it is acceptable to make this. Decreasing this will ensure more stable updates, at the cost of slower learning. We left this at the default range of 3.

## 5. IMPLEMENTATION

---

- **lambd:** The Regularization parameter (lambda) is used when calculating the Generalized Advantage Estimate (GAE). This can be thought of as how much the agent relies on its current value estimate when calculating an updated value estimate. Low values correspond to relying more on the current value estimate, and high values correspond to relying more on the actual rewards received in the environment. The lambd was set at 0.95 which is inside the typical range of 0.9 - 0.95.
- **learning rate:** Initial learning rate for gradient descent. Corresponds to the strength of each gradient descent update step. It was set at 3.0e-4.
- **learning rate schedule:** It determines how learning rate changes over time. For PPO a decaying learning rate is recommended until max steps so learning converges more stably. Linear decays the learning rate linearly so we chose that.

Next we had to determine some **network settings**.

- **hidden units:** It defines the number of units in the hidden layers of the neural network. Correspond to how many units are in each fully connected layer of the neural network. The default is 128 but we chose 256 because in our case the action is a very complex interaction between the observation variables.
- **normalize:** Whether normalization is applied to the vector observation inputs. This normalization is based on the running average and variance of the vector observation. It was set to false as a default.
- **num layers:** It is the number of hidden layers in the neural network. Corresponds to how many hidden layers are present after the observation input, or after the CNN encoding of the visual observation. The typical range is 1-3 but despite fewer layers are likely to train faster and more efficiently in our case we chose 2 layers instead of 1 because our training was more complex. In the thesis we tested both 1 and 2 and found out that while 2 takes a bit more time it provides more accurate results.
- **vis encode type:** The encoder type for encoding visual observations. It was set to simple which uses a simple encoder that consists of two convolutional layers.

## 5.9 Setting Up Python Environment

---

Next some **reward signals** were defined. More specifically, extrinsic rewards had to be enabled to ensure that our training run incorporates our environment-based reward signals.

- **strength:** Factor by which to multiply the reward given by the environment. Typical ranges will vary depending on the reward signal. It was set to 1.
- **gamma:** The discount factor for future rewards coming from the environment. This can be thought of as how far into the future the agent should care about possible rewards. In situations when the agent should be acting in the present in order to prepare for rewards in the distant future, this value should be large. In cases when rewards are more immediate, it can be smaller. Must be strictly smaller than 1 and in our case we put it at 0.99.

Lastly, some other settings needed to be defined.

- **max steps:** Total number of steps (i.e., observation collected and action taken) that must be taken in the environment (or across all environments if using multiple in parallel) before ending the training process. In our case that we had 4 agents with the same behavior name within our environment, all steps taken by those agents contributed to the same max steps count. It was set at  $5.0e7$ .
- **time horizon:** It defines how many steps of experience are needed to collect per-agent before adding it to the experience buffer. When this limit is reached before the end of an episode, a value estimate is used to predict the overall expected reward from the agent's current state. It was set at 128.
- **summary freq:** It is the number of experiences that needs to be collected before generating and displaying training statistics. We set this to 10.000 in order to observe the progress faster, instead of the default 50.000

There are also other settings that were never used so they were put into their default values. The full list can be found here. [\[23\]](#)

## 5. IMPLEMENTATION

---

The image of our final config file can be found below. In order to run the ML agents training you should first activate the pytorch python environment that was set earlier, then download the ml agents package with `pip install mlagents` then `cd` into the folder you want the results to be produced and then run `mlagents-learn` then the path of your `.yaml` file and then `-run-id` then a name of a folder that will be created to store the results (write a string) and also `-quality-level=0` to save processing power. Then click enter and you should see the unity logo. Afterwards click the play in the unity inspector to begin the training.

```
nity things > TrainingThesis > config > ! trainer_config.yaml
behaviors:
  AircraftLearning:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      beta: 1.0e-2
      buffer_size: 20480
      epsilon: 0.2
      num_epoch: 3
      lambda: 0.95
      learning_rate: 3.0e-4
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 256
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        strength: 1.0
        gamma: 0.99
    max_steps: 5.0e7
    time_horizon: 128
    summary_freq: 10000
```

Figure 5.23: yaml file

## 5.10 Training Scene

For the training, a new scene was implemented with just a plane, some obstacles and the drone area path in order to minimize the processing power and still train the agents effectively. The result of the training that is the brain or a .onnx file can then be later used in our other scenes with the same results. Two major scripts were implemented, one for the drone area and one for the drone agent.

### 5.10.1 Drone Area

The drone area script was responsible of creating the checkpoints but also holds some important parameters that are needed in the drone Agent script. During the awake function all the agents that are inside the area are put into a list. During the start function the checkpoints are created and put into a list. A checkpoint is instantiated and then using the help of the cinemachine smooth path points and functions its local position and rotation is defined.

```
private void Createcheckpoints()
{
    //Create checkpoints along the race path
    Debug.Assert(racePath != null, "Race Path was not set");
    checkpoints = new List<GameObject>();
    int numCheckpoints = (int)racePath.MaxUnit(CinemachinePathBase.PositionUnits.PathUnits);
    for (int i = 0; i < numCheckpoints; i++)
    {
        // Instantiate either a checkpoint or finish line checkpoint
        GameObject checkpoint;
        if (i == numCheckpoints - 1)
        {
            checkpoint = Instantiate<GameObject>(finishCheckpointPrefab);
            //set the parent, position and rotation
            checkpoint.transform.SetParent(racePath.transform);
            checkpoint.transform.localPosition = racePath.m_Waypoints[i].position;
            checkpoint.transform.rotation = racePath.EvaluateOrientationAtUnit(i, CinemachinePathBase.PositionUnits.PathUnits) * Quaternion.Euler(90, 0, 0);

            //Add the checkpoints to the list
            Checkpoints.Add(checkpoint);
        }
        else
        {
            checkpoint = Instantiate<GameObject>(checkpointPrefab);

            //set the parent, position and rotation
            checkpoint.transform.SetParent(racePath.transform);
            checkpoint.transform.localPosition = racePath.m_Waypoints[i].position;
            checkpoint.transform.rotation = racePath.EvaluateOrientationAtUnit(i, CinemachinePathBase.PositionUnits.PathUnits) * Quaternion.Euler(90, 0, 0);

            //Add the checkpoints to the list
            Checkpoints.Add(checkpoint);
        }
    }
}
```

Figure 5.24: Create Checkpoints

## 5. IMPLEMENTATION

---

A second script was implemented that resets the position of an agent using its current `NextCheckpointIndex`, unless `randomize` is true, then will pick a new random checkpoint. This is done by setting the start position of the agent to the previous checkpoint, then converting the position on the race path to a position in 3d space and getting the orientation at that position. A horizontal offset is implemented so that the agents are spread out and not fall on top of each other. In the end using the above we set the drone position and rotation.

```
public void ResetAgentPosition(DroneAgent agent, bool randomize = false){
    if(DroneAgents == null){
        FindDroneAgents();
    }
    if(Checkpoints == null){
        CreateCheckpoints();
    }
    if(randomize){
        // Pick a new next checkpoint at random
        agent.NextCheckpointIndex = Random.Range(0, Checkpoints.Count);
    }

    // Set start position to the previous checkpoint
    int previousCheckpointIndex = agent.NextCheckpointIndex - 1;
    if(previousCheckpointIndex == -1){
        previousCheckpointIndex = Checkpoints.Count - 1;
    }

    float startPosition = racePath.FromPathNativeUnits(previousCheckpointIndex, CinemachinePathBase.PositionUnits.PathUnits);
    // Convert the position on the race path to a position in 3d space
    Vector3 basePosition = racePath.EvaluatePosition(startPosition);

    // Get the orientation at the position on the race path
    Quaternion orientation = racePath.EvaluateOrientation(startPosition);

    // Calculate a horizontal offset so that the agents are spread out
    Vector3 positionOffset = Vector3.right * (DroneAgents.IndexOf(agent) - DroneAgents.Count / 2f) * Random.Range(9f, 10f);

    // Set the drone position and rotation
    agent.transform.position = basePosition + orientation * positionOffset;
    agent.transform.rotation = orientation;
}
```

Figure 5.25: Reset Agent Position

### 5.10.2 Drone Agent

The drone agent script is required in order for the training to start. First one of the most crucial parameters that needed to be set is the step timeout which is the number of steps to time out after training. The smaller this is the faster the agent need to act so it does not time out and start again. In the initialization we get the drone agent, the drone area that we defined previously and we set the max step to 5000 steps if training and infinite steps if racing. If the agent do more steps it then has to restart the process.

The function **OnActionReceived(ActionBuffers actions)** comes with the unity ML agents package and needs to be overrode. Here we set 4 continuous actions, one for each movement we want the agent to do. Like the joysticks one Vector 2 is for the movement up down, one for the rotation, one for the move forward and one for the bend. When the training starts the system starts giving random values to these Vector 2s with -1 the minimum value and +1 the maximum. These values then are put in the same functions as our drone movement script (MovementUpDown, MovementForward, Rotation, ClampingSpeedValues, Swerve). With this we manage to train an agent using the same inputs as the user but this time by taking "random" values. These values then are selected accordingly by the system so it can maximize the reward it is set to it. A function was defined that is called VectorToNextCheckpoint() that gets a vector to the next checkpoint the agent needs to fly through and returns a local-space vector.

```
public override void Initialize()
{
    //area = GetComponentInParent<DroneArea>();
    area = areaGameObject.GetComponent<DroneArea>();
    drone = GetComponent<Rigidbody>();
    trail = GetComponent<TrailRenderer>();

    // Override the max step set in the inspector
    // Max 5000 steps if training, infinite steps if racing
    MaxStep = area.trainingMode ? 5000 : 0;
}
```

Figure 5.26: Drone Agent Script Initialization

## 5. IMPLEMENTATION

---

```
public override void OnActionReceived(ActionBuffers actions)
{
    if (frozen) return;

    // Move vector
    moveVectorUpDownChange = actions.ContinuousActions[0]; // move up or none
    if (moveVectorUpDownChange == 2){
        moveVectorUpDownChange = -1f; // move down
    }
    moveVectorLeftRightChange = actions.ContinuousActions[1]; // move right or none
    if (moveVectorLeftRightChange == 2){
        moveVectorLeftRightChange = -1f; // move left
    }

    // Turn vector
    turnVectorUpDownChange = actions.ContinuousActions[2]; // turn up or none
    if (turnVectorUpDownChange == 2){
        turnVectorUpDownChange = -1f; // turn down
    }
    turnVectorLeftRightChange = actions.ContinuousActions[3]; // turn right or none
    if (turnVectorLeftRightChange == 2){
        turnVectorLeftRightChange = -1f; // turn left
    }

    MovementUpDown();
    MovementForward();
    Rotation();
    ClampingSpeedValues();
    Swerve();
    drone.AddRelativeForce(Vector3.up*upForce);
    drone.rotation = Quaternion.Euler(
        new Vector3(tiltAmountForward, currentYRotation, tiltAmountSideways));
}
```

Figure 5.27: On Action Received 1

After the movement functions the reward system had to be defined. Many different attempts were done in order to find the optimal reward system.

**Concept 1:** At start we tested that if this length is 0, meaning that the agent will reach the checkpoint, it gets a reward but after 20 hours of training the results were not good.

**Concept 2:** Afterwards, we tested playing around the parameter that gives the vector to next checkpoint. if the length of the vector was smaller than 50, meaning that the distance between the agents and the checkpoint was smaller than 50, then the agent will get reward 0.2, if it was smaller than 30, it will get 0.4, smaller than 20 it will get 0.6, smaller than 10 will get 0.8 reward and whenever the agent hits the checkpoint invisible collider it will get reward as 1. Then the `nextCheckpointIndex()` will be set to the next checkpoint so the agent has to pass the next one and the `nextStepTimeout` is set to `StepCount + stepTimeout`, so the steps the agents did to reach the checkpoint plus the step timeout that is set for the agent to reach the next checkpoint. This time after around 100 hours of testing with different rewards and distances the results were better. The drone manages to pass the checkpoints but after a small delay. The agents would reach near the checkpoint and then take one or two seconds doing random movement like they were thinking and then passing it.

**Concept 3:** After seeing the previous results, it was realized that the step timeout had to have more impact on the reward system. While in the previous concepts it was set to 300, this time it was set to 80. This means that if the agent doesn't pass the checkpoint in 80 Academy steps the episode ends and it has to try again while also getting a negative reward of -100. By this we encourage the agent to reach and pass the checkpoint fast. We also gave a small negative reward every step -0.02 to encourage it even more. A positive reward is given every step equal to  $2 / \text{distance to next checkpoint}$  so the closer the agent is to the checkpoint the bigger reward it gets every step. If the agent manages to pass a checkpoint a big reward is given that we set as 200. Then again the `nextCheckpointIndex()` is set to the next checkpoint so the agent has to pass the next one and the `nextStepTimeout` is set to `StepCount + stepTimeout`, so the steps the agents did to reach the checkpoint plus the step timeout that is set for the agent to reach the next checkpoint. By this concept we encourage the agent to go fast, to move towards

## 5. IMPLEMENTATION

---

the next checkpoint and also reward him a lot if it manages to pass a checkpoint. This concept worked as this time there were no delays of the agent before each checkpoint and it passed the stage fast and with no delays.

**Final Concept:** We could stay with the previous concept but we wanted to create the best agent possible, an agent that could not just pass the race but beat the user and make him compete with him. Could the user actually manage to beat this agent? We implemented this by reducing the step timeout to 70 (from 80), a number that is smaller than before but not too small so the agents don't even manage to reach the checkpoints in time. The results were amazing since this time the agents flew very fast and optimal and could beat the user. We used this neural network brain for the hard mode in our application and the brain of concept 3 for the easy mode.

```
if (area.trainingMode){
    // Small negative reward every step
    //AddReward(-1f / MaxStep);
    AddReward(-100f / MaxStep);

    // Make sure we haven't run out of time if training
    if(StepCount > nextStepTimeout)
    {
        //AddReward(-.5f);
        AddReward(-100f);
        EndEpisode();
    }

    Vector3 localCheckpointDir = VectorToNextCheckpoint();
    if (localCheckpointDir.magnitude < Academy.Instance.EnvironmentParameters.GetWithDefault("checkpoint_radius", 0f))
    {
        GotCheckpoint();
    }

    AddReward(2f/localCheckpointDir.magnitude);
}
```

Figure 5.28: On Action Received 2

With the above, the big function `OnActionReceived` was done and we had to focus on other functions that were needed in the agent's script. The function `CollectObservations(VectorSensor sensor)` is needed for the agent to collect observations from the environment using some sensors and make decisions based on these observations. This is a fundamental step to implement reinforcement learning as we saw in chapter 2.4. We collect 3 kinds of observations, one for the velocity of the agent, one to where is the distance of the next checkpoint and one for the orientation of the next checkpoint. Each of these is a Vector 3 that contain 3 values that each one represents an observation so the total number of observations is 9. We need this number to pass it in the inspector afterwards.

```
// Collects observations used by agent to make decisions
0 references
public override void CollectObservations(VectorSensor sensor)
{
    // Observe drone velocity (1 Vector3 = 3 values)
    sensor.AddObservation(transform.InverseTransformDirection(drone.velocity));

    // Where is the next checkpoint? (1 Vector3 = 3 values)
    sensor.AddObservation(VectorToNextCheckpoint());

    // Orientation of the next checkpoint (1 Vector3 = 3 values)
    Vector3 nextCheckpointForward = area.Checkpoints[NextCheckpointIndex].transform.forward;
    sensor.AddObservation(transform.InverseTransformDirection(nextCheckpointForward));

    // Total Observations = 3 + 3 + 3 = 9
}
```

Figure 5.29: Collect Observations Script

```
// Gets a vector to the next checkpoint the agent needs to fly through
// returns a local-space vector
2 references
private Vector3 VectorToNextCheckpoint(){
    Vector3 nextCheckpointDir = area.Checkpoints[NextCheckpointIndex].transform.position - transform.position;
    Vector3 localCheckpointDir = transform.InverseTransformDirection(nextCheckpointDir);
    return localCheckpointDir;
}
```

Figure 5.30: VectorToNextCheckpoint Script

## 5. IMPLEMENTATION

---

The `OnEpisodeBegin()` function is another function that needed to be overrode. It is called when a new episode begins, so at the start and every time a previous episode ends with the `EndEpisode()` that happens when the agent hits something or the step timeout reaches. The velocity and angular velocity of the agent is reset to 0 and the `ResetAgentPosition` function is called from the area script that makes the agent try again starting from a different random checkpoint. By doing this we speed up the training process and also help the agent not learn a single standard path but make its movement depending on wherever the next checkpoint is.

```
// Called when a new episode begins
0 references
public override void OnEpisodeBegin(){
    // Reset the velocity, position and orientation
    drone.velocity = Vector3.zero;
    drone.angularVelocity = Vector3.zero;
    area.ResetAgentPosition(agent: this, randomize: area.trainingMode);

    // Update the step timeout if training
    if(area.trainingMode) nextStepTimeout = StepCount + stepTimeout;
}
```

Figure 5.31: On Episode Begin Script

```
// Called when the agent flies through the correct checkpoint
2 references
private void GotCheckpoint(){

    //Next checkpoint reached, update
    NextCheckpointIndex = (NextCheckpointIndex + 1) % area.Checkpoints.Count;

    if(area.trainingMode)
    {
        //AddReward(.5f);
        AddReward(200f);
        nextStepTimeout = StepCount + stepTimeout;
    }
}
```

Figure 5.32: Got Checkpoint Script

The `OnTriggerEnter()` function was set that every time the agent passes through the invisible checkpoint collider it will get a reward. Lastly, the `OnCollisionEnter()` function was defined so that every time the agent hit a collider of an object that was not "agent" it will get a negative reward and the episode would end. This helped the agent learn to dodge structures during the training.

```
// React to entering a trigger
0 references
private void OnTriggerEnter(Collider other)
{
    if (other.transform.CompareTag("checkpoint") && other.gameObject == area.Checkpoints[NextCheckpointIndex]){
        GotCheckpoint();
    }
}

// React to collisions
0 references
private void OnCollisionEnter(Collision collision)
{
    if(!collision.transform.CompareTag("agent"))
    {
        // we hit something that wasn't another agent
        if(area.trainingMode)
        {
            AddReward(-1f);
            EndEpisode();
        }
    }
}
```

Figure 5.33: `OnTriggerEnter` and `OnCollisionEnter` Scripts

As it was mentioned above, for the `CollectObservations` script to work, some sensors had to be put on to the drone agent. We did that by adding them to the drone agent prefab at its front side and using the unity built in script `Ray Perception Sensor 3D`. In this script we defined the rays per direction as 4, Max rays degrees at 70 and ray length at 250. Also on detectable tags we put `untagged` and `checkpoint`. 3 of these sensors were set, one forward with x rotation at 0, one up with x rotation -15 and one down with x rotation at 15 degrees so the agent can scan a larger space that the checkpoints could be in.

## 5. IMPLEMENTATION

---

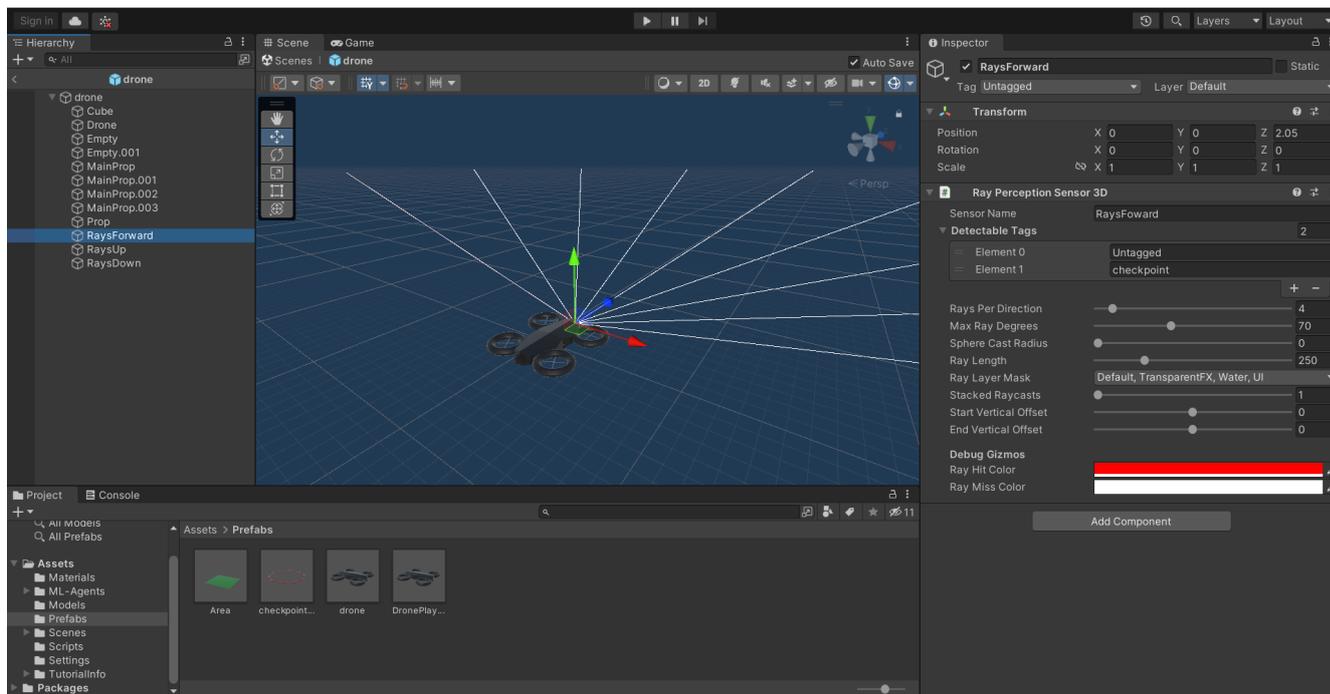


Figure 5.34: Drone Sensors Setup

In the end few more scripts had to be setup inside the editor. One is the Behavioural Parameters script that we pass the number of observations that was defined previously, how many continuous actions we have as well as the model. For the training, the model is empty but after we finish it and have the neural network, we can put the onnx file there as the agent's brain and it will start moving as the training results' suggested. The other script is the Decision Requester that we force the agent to take decisions at least every 5 academy steps.

Bellow is the above described Unity Built in Scripts in the editor.

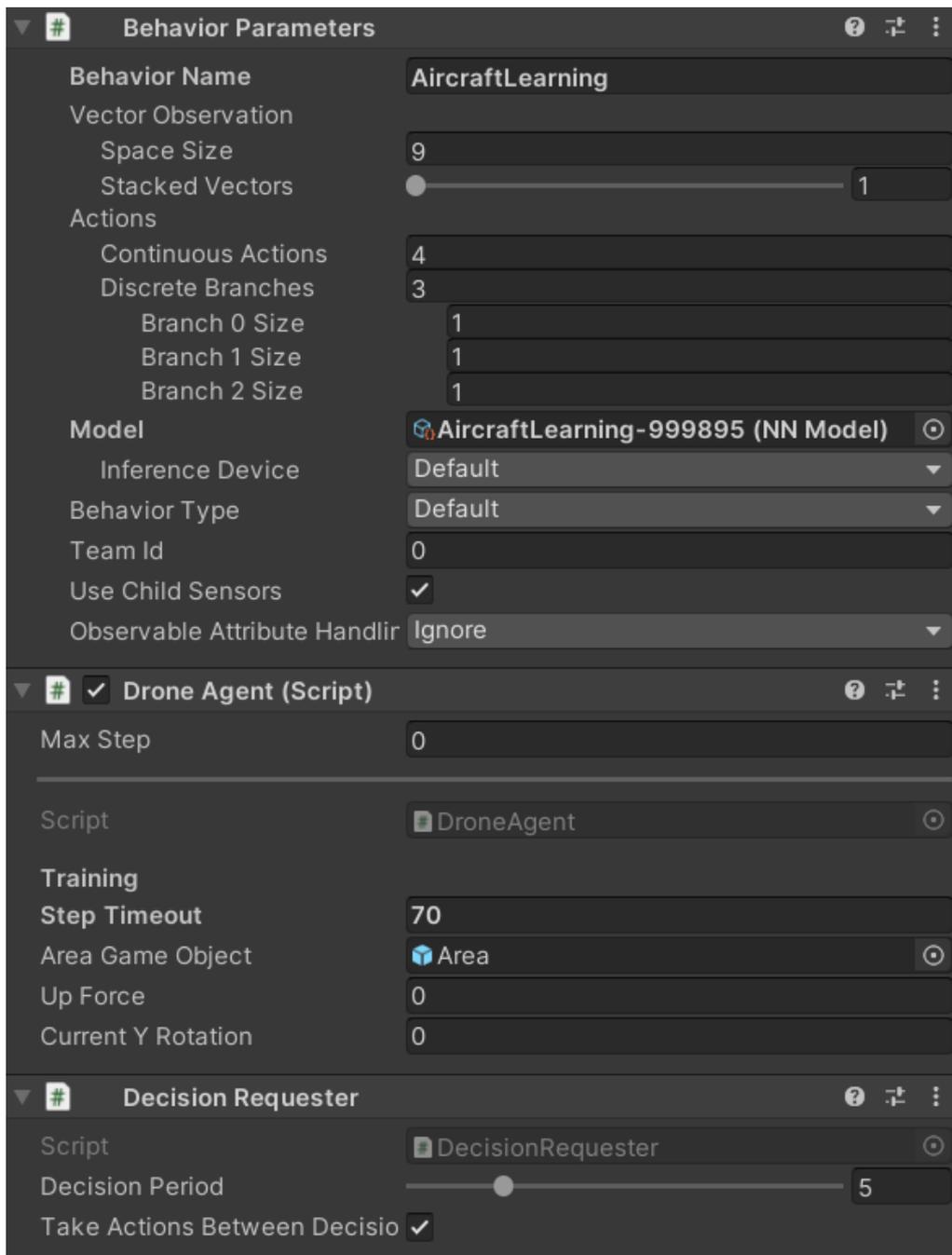


Figure 5.35: Behavioural Parameters and Decision Requester

## 5. IMPLEMENTATION

---

### 5.10.3 Observing Training

During the training we put 4 agents to train at the same time in the same stage and started observing if the reward was getting better every 10.000 steps. In graph bellow we could see that at the start it was very difficult for the agents to get reward but, once they found their way, the reward started increasing exponentially until a certain point. This happened because when we started the training the agents started flying to the sky but later on they learned to go closer to the checkpoint up to the point that they were passing the stage super fast, which then the reward started capping around the same number.

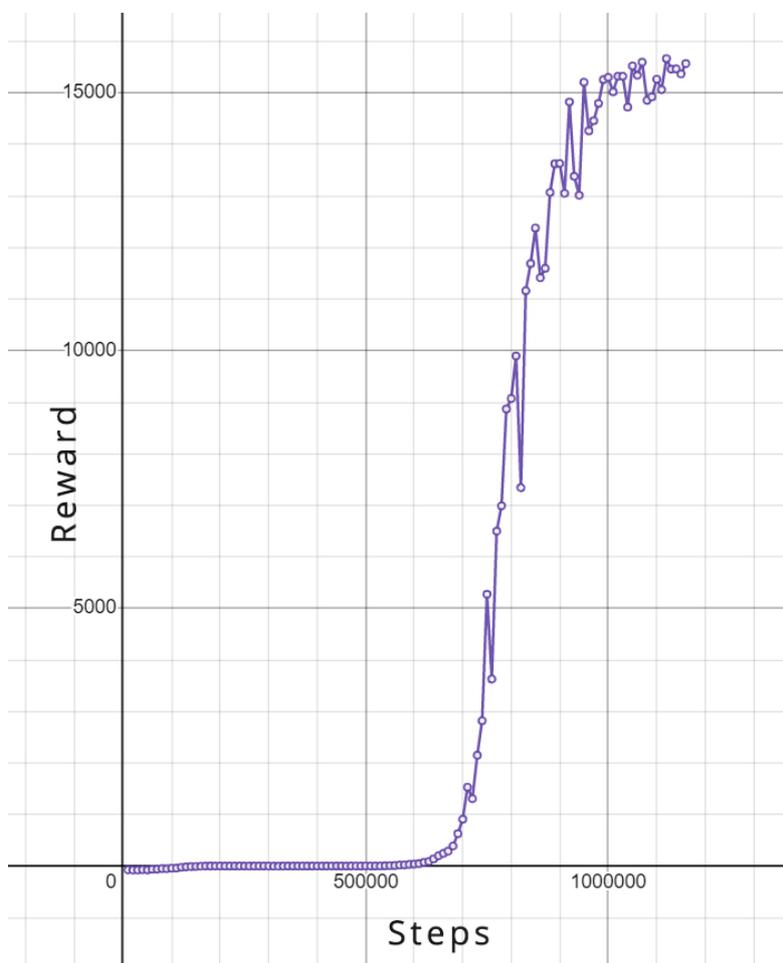


Figure 5.36: Reward per Step Graph

### 5.10.4 Training Results

At the end of the training, the neural network file is created that can be found in the destination folder that have been set when we run the ml agents command. This onnx file is the produced "brain" and can be then assigned into the behavioural parameters of the agent to make it run by himself following the assigned logic.

Name	Date modified	Type	Size
AircraftLearning-499989.onnx	1/30/2023 3:51 AM	ONNX File	389 KB
AircraftLearning-499989.pt	1/30/2023 3:51 AM	PT File	2,301 KB
AircraftLearning-999895.onnx	1/30/2023 4:36 AM	ONNX File	389 KB
AircraftLearning-999895.pt	1/30/2023 4:36 AM	PT File	2,301 KB
AircraftLearning-1026121.onnx	1/30/2023 4:39 AM	ONNX File	389 KB
AircraftLearning-1026121.pt	1/30/2023 4:39 AM	PT File	2,301 KB
checkpoint.pt	1/30/2023 4:39 AM	PT File	2,301 KB
events.out.tfevents.1675040324.DESKTOP...	1/30/2023 4:38 AM	0 File	874 KB

Figure 5.37: Training Results

## 5.11 3rd Party Assets

Most of the assets that were used in this thesis are taken by the polyeffect's Low Poly Ultimate Pack. The audio used as a loop is the Lupus Nocte - Hadouken (Royalty Free Music) while the the passing checkpoint sounds are taken from the casual music pack.

## 5. IMPLEMENTATION

---

# Chapter 6

## Evaluation & Testing

### 6.1 Introduction

After the application was done we wanted to evaluate the training and how well the users performed both when playing against and without the AI trained opponents and if it had any impact on the training process.

### 6.2 Evaluation Method & Result Analysis

The evaluation method that we used was to make different users compete in the first stage of our application that had a variety of elements as well as wind zone around the mountain area. The input that we wanted to get is how many restarts they did as well as how much time it took them to finish the stage, if they managed to reach till the end. We asked them to do it 2 or even 3 times to see if they improved in performance. Lastly, we put half the users to compete with and the other half without the AI to see if there were any differences in the training process and if the competitive nature of the humans have any effect in it.

We had twenty four people testing the application, twelve playing against AI and the other twelve with. The results were noticeable since the number of restarts dropped significantly between the first and the second trial. This can be shown in a great way in the following two graphs that represent the number of people and the restarts they had in the two trials.

## 6. EVALUATION & TESTING

---

On the first graph we can see that over half of the people took 4 or more restarts before finally managing to pass the stage. The players restarted either by falling to the ground or hitting a terrain element. Using the feedback that we got for the players, every time they restarted they were reaching further into the stage so there was an improvement.



Figure 6.1: Trial 1 Restart Results

In contrast, on the second graph we can see that all of the users half of the people took zero to three tries and most of them were on the one or two mark, that signifies the successful learning of the player through trial 1 and 2.

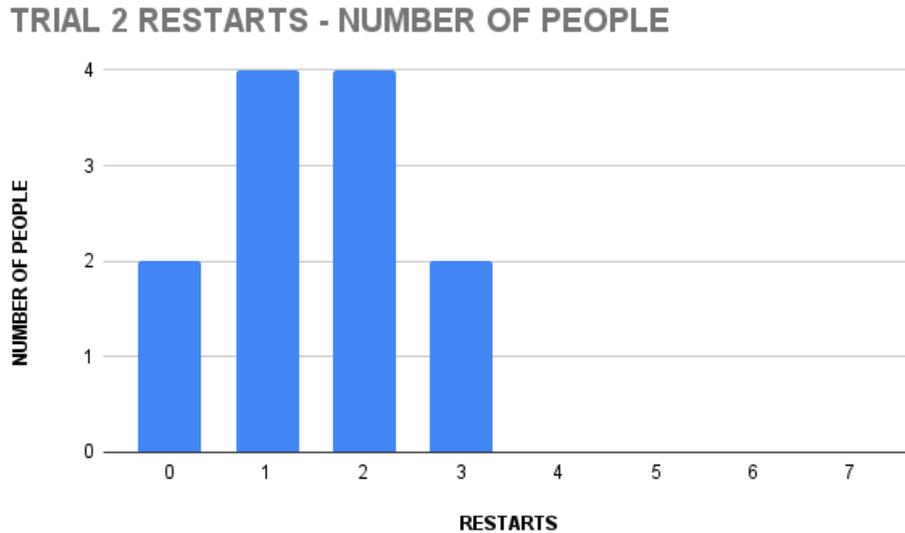


Figure 6.2: Trial 2 Restart Results

From the twenty four people that we tested, there is a significant number of nine people people that couldn't finish even the first trial due to motion sickness. Most of these people were very new to VR, to most of them it was the first time trying a Virtual Reality headset. Few of these cases tho were also experienced users in VR. The remaining 15 people managed to complete at least the first trial, meaning they passed all the checkpoints. From those people, three couldn't manage to complete the second trial due to motion sickness that they have developed after they started it. During the testing we asked some users to also do three trials but then we abandoned the idea because most of the people developed motion sickness.

Besides the restarts, the completion time of the users was collected. The average completion time for the first trial of the users that tested against an AI opponent was 94 seconds while for the second trial the average time was 81 seconds (13.8% drop). On the other hand, for the users that tested playing without an AI the average completion time for the first trial was 168 seconds while for the second was 112.6 seconds (33% drop). Analyzing the above results we can see that the required time to finish the stage dropped from trial one to trial two on both of the cases. This signifies the learning of the users on operating drones and flying them fast. Also, playing versus the AI helped the users

## 6. EVALUATION & TESTING

---

train better as we observed a 44% faster time on the first trial and 27.7% faster time on the second trial compared to playing without the AI opponent. This signifies that the competitive spirit of the users helped them pass the stage faster.

It is worthy to mention that the users that competed against the trained AI opponents had faster passing time than the players that were training solo. Although, these players had also a higher number of restarts for the first trial with average number of 3.55 compared to the 3 restarts of people that played without AI. For the second trial, we observed similar results to 1.71 average restarts of playing with AI compared to 1.2 average restarts of playing without. As we can see, the users that tested were faster to complete the stage but also took risks and hit the terrain or terrain elements more often than those playing without. This can be due to trying to compete with the AI opponents rather than just focusing on passing safely the stage.

# Chapter 7

## Conclusion & Future Work

### 7.1 Conclusion

In this thesis, we created a physics-based virtual reality drone racing simulator designed to help humans train to drive drones at high speeds. The simulator includes realistic physics for drone movement and wind zones, and features an artificial intelligence trained using reinforcement learning to compete with human players. With this simulator, we managed to provide a safe and immersive environment for humans to practice and improve their drone racing skills, while also providing a challenging opponent in the form of the AI. Through the use of VR technology, the simulator allows for a more realistic and engaging training experience. The user is trained to pass the stage fast while also avoid obstacles in the way that could cause serious harm to a drone in real life scenarios. Through the evaluation and the feedback from the users we saw that for every time the players tried the stage they improved their time taken as well as had lower number of crashes which makes the simulator a success. There were cases tho with new players to VR experiencing motion sickness and this needs to be further researched on how to reduce it.

This thesis also explores the impact of playing against an artificial intelligence (AI) opponent on the training and performance of human players. Through the use of feedback from human testers, we analyzed how playing alone versus against an AI affected the training experience and the resulting improvement in performance. Our findings suggest that playing against an AI opponent can provide a more challenging and engaging training experience, leading to better performance outcomes for human players. However, we also found that the benefits of playing against an AI may be influenced by individual

## 7. CONCLUSION & FUTURE WORK

---

differences in motivation and preferences, and that there may be value in providing both solo and AI-opponent training options to meet the needs of different players. Overall, our results highlight the potential of AI to enhance training and performance in the sport of drone racing, and suggest that the use of AI opponents in training programs may be a promising approach for improving human performance in this and other domains but needs to be researched further.

Overall, the physics-based VR drone racing simulator represents a significant advancement in the field of drone racing training and offers a promising tool for improving human performance in this exciting and rapidly-growing sport.

### 7.2 Future Work

While the VR Drone racing simulator developed in this thesis provides a realistic and physics-based training platform for users, there are several areas where further research could be conducted to improve its overall effectiveness and usability.

One limitation of the simulator is the potential for motion sickness among new users. While the simulator includes features such as a limited field of view to reduce the likelihood of motion sickness, it would be worthwhile to explore additional methods to minimize this issue. For example, a study could be conducted to determine the optimal frame rate and field of view for standalone devices, such as the Oculus Quest for reducing motion sickness in VR, and these findings could be applied to the simulator. This study has been conducted in PC VR Devices but not in standalone VR headsets such as Oculus Quest.

Another area for future work is the development of a multiplayer mode for the simulator. This would allow users to compete with each other in a virtual environment and promote a more engaging gameplay. Racing against other players in a multiplayer mode can provide a more realistic experience for users, as they will be competing against human opponents who are unpredictable and can make mistakes. This can help to test the realism of our physics model and improve it based on feedback from users. This could be achieved through the integration of a networking system, which would allow multiple users to connect to the simulator and compete with each other in real-time.

For the physics, while the drag was implemented using the drag equation, the lift was an approximation. Further research on the lift equation and how to apply it to drones inside the editor needs to be conducted. Also, if we want to dive deeper into the physics, classes can be made for the motors, the PID controller and calculate the rpm before applying any motion to the drone. Now in the thesis, we use the resultant force for the force provided by the propellers so the next step is to separate the forces for each propeller and provide force in each one separately. This will allow more accurate movement of the drones based on real life physics.

For the training perspective, more research needs to be conducted by changing the training parameters like layers, hidden units and batch size and the others that were mentioned in chapter five. Moreover, the reward function needs to be optimized more by finding the golden ration which the AI drone will fly on the best way possible. This will help find a more optimal training process both by time and processing power spent for the training as well as the results provided.

Finally, the simulator could be validated against real-world drone racing to assess its effectiveness as a training tool. This could be achieved by comparing the performance of users who have trained in the simulator to those who have trained using real flying drones. A study could be conducted to compare the performance of these two groups in a real-world racing environment, and the results could be used to determine whether the simulator is an effective training tool.

Overall, while the VR Drone racing simulator developed in this thesis represents a significant step forward in the field of drone racing training, there are still several areas where further research could be conducted to improve its overall effectiveness and usability. By addressing the limitations identified in this chapter, the simulator could become an even more powerful tool for training drone racing enthusiasts.

## 7. CONCLUSION & FUTURE WORK

---

# Bibliography

- [1] Wikipedia, Unmanned aerial vehicle. [https://en.wikipedia.org/wiki/Unmanned\\_aerial\\_vehicle](https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle). 8
- [2] Forbes, Nikola Tesla's Third Greatest Invention: The First Drone. Jul 11, 2018 <https://www.forbes.com/sites/berniecarlson/2018/07/11/nikola-teslas-third-greatest-invention-the-first-drone>. 8
- [3] Engadget, Tesla's toy boat: A drone before its time. January 19, 2014 <https://www.engadget.com/2014-01-19-nikola-teslas-remote-control-boat.html>. 9
- [4] Imperial War Museums, London, A Brief History of Drones. <https://www.iwm.org.uk/history/a-brief-history-of-drones>. 10
- [5] Ben Lutkevich, Alan R. Earls. (2021) drone (UAV). Published in [techtarget.com](https://www.techtarget.com), a network of technology-specific websites. <https://www.techtarget.com/iotagenda/definition/drone>. 10
- [6] AUAV, Drone Types. <https://www.auav.com.au/articles/drone-types>. 14
- [7] Cfd Flow Engineering, Working Principle and Components of Drone. <https://cfdflowengineering.com/working-principle-and-components-of-drone/>. 18
- [8] Prasanna.(2022) Virtual Reality Advantages And Disadvantages — What is Virtual Reality (VR)?, Benefits, Drawbacks, Pros and Cons. Published in [aplustopper.com](https://aplustopper.com), India's Number 1 Educational Portal for ICSE students. “virtual-reality-advantages-and-disadvantages” 22
- [9] The Drone Racing League (2022), available at: “The drone racing league” 19, 31

## BIBLIOGRAPHY

---

- [10] “VelociDrone FPV Racing Simulator” (2022), available at: [“VelociDrone FPV”](#) 19
- [11] Gilles Albeaino, Ricardo Eiris, Masoud Gheisari and Raja Raymond Issa. (2021) DroneSim: a VR-based flight training simulator for drone mediated building inspections. 19
- [12] Jian Guo, Ming Liu, Yi Guo, Ting Zhou. (2021) An AR/VR-Hybrid Interaction System for Historical Town Tour Scenes Incorporating Mobile Internet. In 2021 International Conference on Digital Society and Intelligent Systems (DSInS) 19
- [13] Simlat UAS Simulation” (2022), available at: “Simlat UAS Simulation” 19
- [14] A Beginner-Friendly Explanation of How Neural Networks Work, Jun 2, 2020, available at: “How Neural Networks Work” 30
- [15] g2.com, September 26, 2019, The Very Real History of Virtual Reality (+A Look Ahead) <https://www.g2.com/articles/history-of-virtual-reality>. 22
- [16] Slater Mel, Wilbur Syliva A Framework for Immersive Virtual Environments (FIVE): Speculations on the Role of Presence in Virtual Environments Presence: Teleoperators and Virtual Environments, 1997. 22
- [17] Unity Packages, Cinemachine Documentation <https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/CinemachineSmoothPath.html>. 37
- [18] NASA Glenn Research Center, 2021 <https://www.grc.nasa.gov/www/k-12/rocket/drageq.html>. 38
- [19] Unity Blog, Introducing: Unity Machine Learning Agents Toolkit, September 19, 2017 by Arthur Juliani. <https://blog.unity.com/technology/introducing-unity-machine-learning-agents>. 42
- [20] Anaconda environment <https://www.anaconda.com/>. 74
- [21] Pytorch environment <https://pytorch.org/>. 74
- [22] Unity ML Agents <https://github.com/Unity-Technologies/ml-agents>. 75

## BIBLIOGRAPHY

---

- [23] Unity ML Agents Configuration File Instructions <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md>. 77