SEMESTER TRAINING REPORT

On

"Droidrush: A Deep Imitation Learning & Reinforcement Learning Flight Simulation"

Submitted in partial fulfilment of requirements
for the award of the degree

Bachelor of Technology
In
Computer Science and Engineering
To
IKG Punjab Technical University, Jalandhar

SUBMITTED BY:
Name: Manan Sharma
Roll no.: 1902133
Semester: 8th
Batch: 2019-23

Under the guidance of
Mr. Rajeev Sharma
Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Chandigarh Engineering College-CGC, Landran
Mohali, Punjab - 140307

CERTIFICATE

This is to certify that **Mr. Manan Sharma** has partially completed / completed / not completed the Semester Training during the period from **January 2023** to **June 2023** in our Organization / Industry as a Partial Fulfilment of Degree of Bachelor of Technology in Computer Science & Engineering.

(Signature of Project Supervisor)

Date: 31-05-2023

ThinkNEXT Technologies Private Limited

ThinkNEXT™

Innovation at every step...

ISO 9001:2015 Certified Company

Scan and verify your Certificate

Certificate ID:675453

Ref.No. TNT/C-23/10479

Iconic Business Summit AWARDS WINNER 2021

Nation's Business Pride AWARDS WINNER 2021

National Gratitude AWARDS WINNER 2020

Asia's Quality & Entrepreneurship AWARDS WINNER 2019

Business Leaders AWARDS WINNER 2019

NATIONAL ICON AWARDS WINNER 2018

Certificate

Certificate of Excellence

Provisional

This Certificate do hereby recognizes that

Manan Sharma S/o Brijesh Sharma

has successfully completed Industrial Training Program from

02nd January 2023 to *26th May 2023*

in *Python* Grade *A*

ISO Certified

Member of Confederation of Indian Industry

ISTQB International Software Testing Qualifications Board

Affiliated

For ThinkNEXT Technologies Pvt. Ltd.

Authorised Signatory

Member Training Division

For ThinkNEXT Technologies Pvt. Ltd.

Director

Director

Corporate Office: S.C.F 113, Phase XI, Mohali (Punjab)

A Outstanding 100-90% | B Excellent 89-80% | C Very Good 79-70% | D Good 69-60% | E Satisfactory 59-50%

CANDIDATE DECLARATION

I hereby declare that the Project Report entitled ("**Droidrush: A Deep Imitation Learning & Reinforcement Learning Flight Simulation**") is an authentic record of my own work as requirements of 8th semester academic, during the period from **2nd January 2023** to **30th June 2023** for the award of degree of B. Tech. (Computer Science & Engineering, Chandigarh Engineering College- CGC, Landran, Mohali under the guidance of (Mr. Rajeev Sharma).

(Signature of Student)
(Manan Sharma)
(1902133)

Date: 31-05-2023

Certified that the above statement made by the student is correct to the best of our knowledge and belief.

Signatures

Examined by:

1. 2. 3. 4.

Head of Department
(Signature and Seal)

ACKNOWLEDGMENT

I take this opportunity to express my sincere gratitude to the Director- Principal **Dr. Rajdeep Singh** Chandigarh Engineering College, Landran for providing this opportunity to carry out the present work.

I am highly grateful to the **Dr. Sukhpreet Kaur** HOD CSE, Chandigarh Engineering College, Landran (Mohali). I would like to express my gratitude to other faculty members of Computer Science & Engineering department of CEC, Landran for providing academic inputs, guidance & Encouragement throughout the training period. The help rendered by **Mr. Rajeev Sharma;** Supervisor for Experimentation is greatly acknowledged. I would also like to acknowledge the help of my Project Partner, **Naman Gupta**, who helped me all along the building of this project. Finally, I express my indebtedness to all who have directly or indirectly contributed to the successful completion of my semester training.

ABOUT COMPANY

As a final semester student pursuing a degree in Computer Science & Engineering, I had the opportunity to gain valuable industry experience through a Six-month Semester Training program with **ThinkNext Technologies Pvt. Ltd.**

During my training, I received a certification in Python, which provided me with a strong foundation in programming and software development. However, my major project was focused on the use of Reinforcement Learning, Deep Reinforcement Learning and working on Autonomous and Intelligent Flight, which required me to work independent of the Institute with my project partner, to meet our End-Semester Project requirements.

ThinkNext Technologies Pvt. Ltd. is an ISO 9001:2008 Certified Software/Electronics Systems Development and CAD/CAM consultancy company that offers a range of services in areas such as website designing and development, college ERP, school ERP, university ERP, digital marketing, Android and iPhone app development, industrial automation, robotics, machine learning, Artificial embedded/PLC-SCADA products, industrial training, and many more. It is also certified by Ministry of Corporate Affairs. ThinkNEXT Technologies has a global client base, primarily from USA, Canada, Singapore, UK, Australia, New Zealand, and **India.**

Through my association with Chandigarh Engineering College, and ThinkNext Technologies Pvt. Ltd., I was able to gain practical knowledge of the latest technologies and tools used in the industry. I was also able to enhance my understanding of software development, project management, and team collaboration. Overall, my experience with ThinkNext Technologies Pvt. Ltd. has been invaluable in shaping my skills and knowledge as a software developer, and I am grateful for the opportunity to have been a part of this dynamic organization.

ABSTRACT

The field of Reinforcement Learning has seen tremendous growth in recent years, and this project aims to contribute to this field by creating an artificial environment where agents can learn to fly through a course with accuracy and near-perfect flight paths. The project utilizes Unity's ML-Agents Libraries to train competitive AI agents with different reward functions, incorporating both Imitation Learning and Reinforcement Learning to create a smooth and stable gameplay. The airplanes in this environment fly freely in space using "Raycast" vision and use reinforcement learning to train along the flight path.

An important aspect of the project that we have created is that it includes a Human Controlled agent, or the Aircraft Player, (going by the scripting definition we used) that will compete against agents which were trained using imitation learning and Deep Reinforcement Learning algorithms, thereby creating a multi-agent environment and simulates real-world scenarios, providing an engaging, and realistic hands-on experience to the players.

The objective of this project is to showcase the potential of Reinforcement Learning in the field of autonomous self-driving technology, which has shown tremendous promise with innovative research and developments. This project demonstrates the ability of AI agents to learn and adapt to different scenarios and showcases the potential for further advancements in this field, particularly in the field of Reinforcement Learning and Deep Learning.

TABLE OF CONTENTS

LIST OF FIGURES

CONTENT **PAGE NO.**

LIST OF TABLES

CONTENT **PAGE NO.**

CHAPTER 1: INTRODUCTION

Reinforcement Learning is an area of Machine Learning concerned with how intelligent agents ought to take actions in an environment to maximize the notion of cumulative reward. In each state of the environment, it acts based on the policy, and as a result, receives a reward and transitions to a new state. The goal of RL is to learn an optimal policy which maximizes the long-term cumulative rewards. To achieve this, there are several RL algorithms and methods, which use the received rewards as the main approach to approximate the best policy. Generally, these methods perform well. In some cases, though the teaching process is challenging. This can be especially true in an environment where the rewards are sparse (e.g., a game where we only receive a reward when the game is won or lost). To help with this issue, we can manually design rewards functions, which provide the agent with more frequent rewards. Also, in certain scenarios, there isn't any direct reward function (e.g., teaching a self-driving vehicle), thus, the manual approach is necessary. However, manually designing a reward function that satisfies the desired behaviour can be extremely complicated. A feasible solution to this problem is imitation learning (IL). In IL instead of trying to learn from the sparse rewards or manually specifying a reward function, an expert (typically a human) provides us with a set of demonstrations. The agent then tries to learn the optimal policy by following, imitating the expert's decisions. Imitation Learning is a training method where the computer imitates human behaviour. In IL, instead of the reward function, an expert, usually a human, provides the agent with a set of demonstrations. The agent then tries to learn the optimal policy by following and imitating the expert's decisions. Finally, the agent learns to map between observations and actions based on the demonstrations. I have created a virtual environment comprising of a desert habitat, along with checkpoints along the way, effectively forming a race trace, and interestingly enough, since there is much impetus on self-driving cars these days, my interest naturally diverged towards the sky, and decided that autonomous flight needs its due recognition. The 3D environment, which has been created using Blender, and worked upon in Unity and consists of various checkpoints, serving as rewards, and the agents, or the planes must fly through the checkpoints to progress to the next lap, and beat the other agents as well. After undergoing multiple training sequences, the agents' fly a near perfect path, which is an objective of the endeavour.

1.1 BACKGROUND

The history of reinforcement learning has two main threads, both long and rich, that were pursued independently before intertwining in modern reinforcement learning. One thread concern learning by trial and error that started in the psychology of animal learning. This thread runs through some of the earliest work in artificial intelligence and led to the revival of reinforcement learning in the early 1980s. The other thread concerns the problem of optimal control and its solution using value functions and dynamic programming. For the most part, this thread did not involve learning. Although the two threads have been largely independent, the exceptions revolve around a third, less distinct thread concerning temporal-difference methods such as used in the tic-tac-toe. The term "optimal control" came into use in the late 1950s to describe the problem of designing a controller to minimize a measure of a dynamical system's behaviour over time. One of the approaches to this problem was developed in the mid-1950s by Richard Bellman and others through extending a nineteenth century theory of Hamilton and Jacobi. This approach uses the concepts of a dynamical system's state and of a value function, or "optimal return function," to define a functional equation, now often called the Bellman Ford Equation. The notion of self-play has a long history in the practice of building artificial agents to solve and compete with humans in games. One of the earliest uses of this mechanism was Arthur Samuel's checker playing system, which was developed in the '50s and published in 1959. This system was a precursor to the seminal result in RL, Gerald Tesauro's TD-Gammon published in 1995. TD-Gammon used the temporal difference learning algorithm TD(λ) with self-play to train a backgammon agent that nearly rivalled human experts. In some cases, it was observed that TD-Gammon had a superior positional understanding to world-class players. Self-play has been instrumental in a number of contemporary landmark results in RL. Notably, it facilitated the learning of super-human Chess and Go agents, elite DOTA 2 agents, as well as complex strategies and counter strategies in games like wrestling and hide and seek. In results using self-play, the researchers often point out that the agents discover strategies which surprise human experts. Self-play in games imbues agents with a certain creativity, independent of that of the programmers. The agent is given just the rules of the game and told when it wins or loses. From these first principles, it is up to the agent to discover competent behaviour. In the words of the creator of TD-Gammon, this framework for learning is liberating "...in the sense that the program is not hindered by human biases or prejudices that may be erroneous or unreliable." This freedom has led agents to uncover brilliant strategies that have changed the way human experts view certain games.

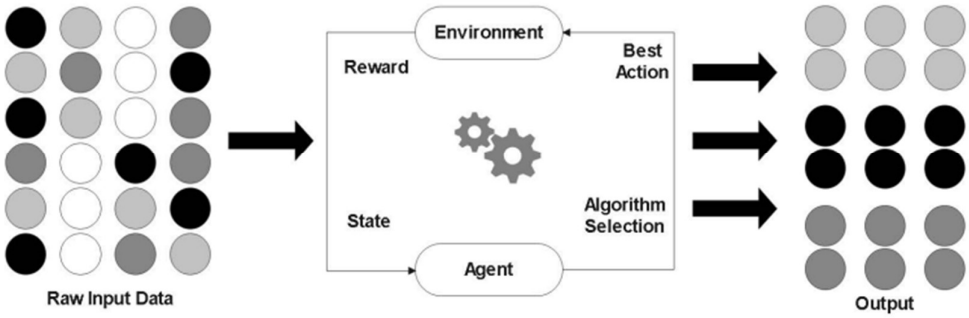

Figure 1: Background Framework for the Proposed Agent system

1.2 MOTIVATION

As a passionate problem solver, our motivation for pursuing this project was fuelled by a deep interest in the intersections of neuroscience, reinforcement learning, deep learning, and aviation. Inspired by the ground-breaking work of industry leaders such as Lex Fridman and Elon Musk, we sought to explore the potential for integrating self-driving technology with the complex and dynamic world of aviation.

One of our team members was particularly motivated by the intersection of self-driving technology and aviation, having previously created a VFX trailer for Top Gun: Maverick. This personal project sparked their interest in the aviation industry, and they began to explore the possibility of combining self-driving technology with airplanes. Speaking about his personal project, which inspired him to conceive self-flying aircrafts and using that foundational idea to continue with their own project, he writes – "In addition to my love for Artificial Intelligence and Automated driving technology, I was also interested in Aviation, having created a VFX trailer of Top Gun: Maverick in a personal venture and year and a half back, which has since then garnered over 14k views, I was naturally driven to try to create a game in which a manual pilot competes against the AI agents which have been fed the deep neural network which has used PPO and other RL models, and have the potential to fly a near perfect path – Just like the Su-57s flew – advanced fighter jets, while the Manually controlled aircrafts struggles. Let's just visualize them to be older F-14s, driven by an inexperienced fighter pilot.

As we delved deeper into the world of reinforcement learning, we realized the potential for its application in creating a multi-agent environment where manual pilots compete against neural networks trained using reinforcement learning models. The complexity of such an environment presented a fascinating and rewarding challenge, requiring us to consider a range of variables such as physics, rigid body laws, and complex behaviours of automated agents.

Our goal in this project was to develop the optimal policy and reinforcement learning algorithm to ensure that non-playable agents performed optimally and were nearly unbeatable. Overcoming the challenge of getting the agents to fly in a straight path, we were motivated to push ourselves further by designing the agents to follow checkpoints and fly optimally through them.

Through this project, we hope to contribute to the growing body of research on the integration of reinforcement learning and aviation, and ultimately inspire others to explore the possibilities of this emerging field.

1.3 OBJECTIVE

- The objective of this project is to create a 3D virtual game environment with diverse objects, comprising two different scenes (a desert and a snowy environment) and **varying difficulty levels.**
- The environment will be populated with intelligent agents trained using reinforcement learning algorithms, including imitation learning and Proximal Policy Optimization (PPO). These agents will fly planes in the environment with different levels of proficiency, providing a challenging gameplay experience for the user. One plane will be controlled by the user, with the goal of beating the other three intelligent agents while passing different checkpoints. The game will be won by the player who finishes two laps in the least time split.
- The project will utilize Unity's ML-Agents libraries to train the agents, using different reward functions to perform specific tasks. The agents will use "Raycast" vision to navigate the environment and will be trained to follow a near-perfect flight path using reinforcement learning algorithms. The project will contribute to the field of reinforcement learning and autonomous self-driving technology, providing insights into how machine learning agents can be trained to navigate complex environments and perform specific tasks. The project also aims to showcase the potential of reinforcement learning algorithms in creating engaging and challenging gameplay experiences for users.

1.4 FEASIBILITY ANALYSIS

In order to determine the feasibility of the project, several factors must be considered, such as the availability of training data, the hardware requirements, and the potential risks associated with the development of the game environment.

1.4.1 Availability of Training Data

The availability of training data is critical for the successful development of a neural network model. In this project, the training data will be generated by training four different environments simultaneously on an Nvidia GPU. By training the different agents on these four different environments or scenes, the processing time required to train the neural network model can be reduced by splitting the processes in parallel. The use of ML-agents will enable the agents to be trained in a reasonable manner.

1.4.2 Hardware Requirements

Hardware bottlenecks and resource restraints pose a moderate risk to the development of the game environment. These constraints could affect the performance of the neural network model, potentially leading to slower processing times or even failures. However, since the risk appetite for this project is high, these risks can be mitigated by allocating the necessary resources and optimizing the performance of the hardware.

1.4.3 Potential Risks

The development of any project involves potential risks, and it is important to identify these risks and take the necessary actions to mitigate their impact. Table 1 below outlines the potential risks associated with the development of the game environment, along with their probability, actions, and cost of action.

Table 1: Risks and Costs

Risk	Probability	Action	Cost of Action
The 3D environment of the game fails to render	Low	Have a backup in Blender, so it won't take much time to load up a brand-new copy	Low
The adversaries or opponents fail at flying through checkpoints	High	Train a better neural network model	High
The aircraft manual agent continues on after losing the path	Very High	Adjust the airplane back to the checkpoint so that it is not able to veer off path	Low
The game takes forever to load up in the Unity Hub	Low	Free up unwanted resource usage from background tasks and unwanted processes	0
The user is not able to control the main player agent unfeasibly	High	The whole thing will not function as per the requirements of the same	Dependence

CHAPTER 2: SOFTWARE REQUIREMENTS AND SPECIFICATIONS

2.1 ANALYSIS DOCUMENT

The final game is promised to deliver to the customer, a proper functioning game with a manually controllable three dimensional aircraft model, which will have proper reflections of the environment, and a properly designed race course, complete with desert environment, and rocks, and valleys, and hills through which the course has been made, placing checkpoints strategically, to test the flying of the neural network trained agents against the user. The checkpoints have been placed haphazardly to maximize the difficulty for the agents, thereby showcasing the prowess of the reinforcement learning algorithm and ML-Agents library, a corresponding timer will be keeping track of the lap timing, and the user would be expected to fly two odd laps, on the completion of which the game would end.

The end-product will be playable on any device, provided it is a windows device, we expect the user to be able to control the plane and fly against the competing agents, which the user can expect to fly almost perfectly.

The customer can check the final game for four working models of the RL Agents, a complete three-dimensional environment, proper rewards and checkpoints, complete with detailed environmental aspects. They can expect a proper working game by the end of the final product.

2.2 PURPOSE

The purpose of this project is to design and develop a three-dimensional racing game that utilizes reinforcement learning algorithms and ML-Agents library to create intelligent agents capable of navigating a complex race course. The project aims to demonstrate the effectiveness of these algorithms and showcase the potential of ML-Agents in developing advanced artificial intelligence agents for gaming applications.

The game is designed to provide an engaging user experience and challenge the user's piloting skills by pitting them against intelligent agents trained using the reinforcement learning algorithm. The game's purpose is to provide a platform for users to enjoy the thrill of racing while simultaneously demonstrating the potential of using advanced artificial intelligence techniques in gaming applications.

The final product aims to deliver a fully functional game that meets the expectations of the client while providing a comprehensive understanding of the capabilities of reinforcement learning

algorithms and ML-Agents library in game development. The game should be playable on any Windows device and deliver a high-quality user experience.

2.2.1 DEFINITIONS

- **Neural network model**: A type of artificial intelligence that is inspired by the structure and function of the human brain. It is designed to recognize patterns and make predictions based on input data.
- **ML-Agents**: A software package developed by Unity Technologies that allows users to train intelligent agents using machine learning algorithms.
- **Reinforcement learning**: A type of machine learning that is based on trial and error. An agent learns to make decisions by receiving rewards or punishments for certain actions.
- **Unity Hub**: A desktop application that allows users to manage multiple installations of Unity on their computer, as well as access a variety of Unity services.
- **Checkpoint**: A predefined location in a game that players must reach in order to progress to the next level or complete a task.
- **GPU**: Graphics Processing Unit, a specialized processor designed to perform complex calculations related to rendering images and video. It is commonly used in machine learning applications for its high processing power.
- **Reinforcement Learning in Adversarial Games**: In a traditional RL problem, an agent tries to learn a behaviour policy that maximizes some accumulated reward. The reward signal encodes an agent's task, such as navigating to a goal state or collecting items. The agent's behaviour is subject to the constraints of the environment. For example, gravity, the presence of obstacles, and the relative influence the agent's own actions have, such as applying force to move itself are all environmental constraints. These limit the viable agent behaviours and are the environmental forces the agent must learn to deal with to obtain a high reward. That is, the agent contends with the dynamics of the environment so that it may visit the most rewarding sequences of states.

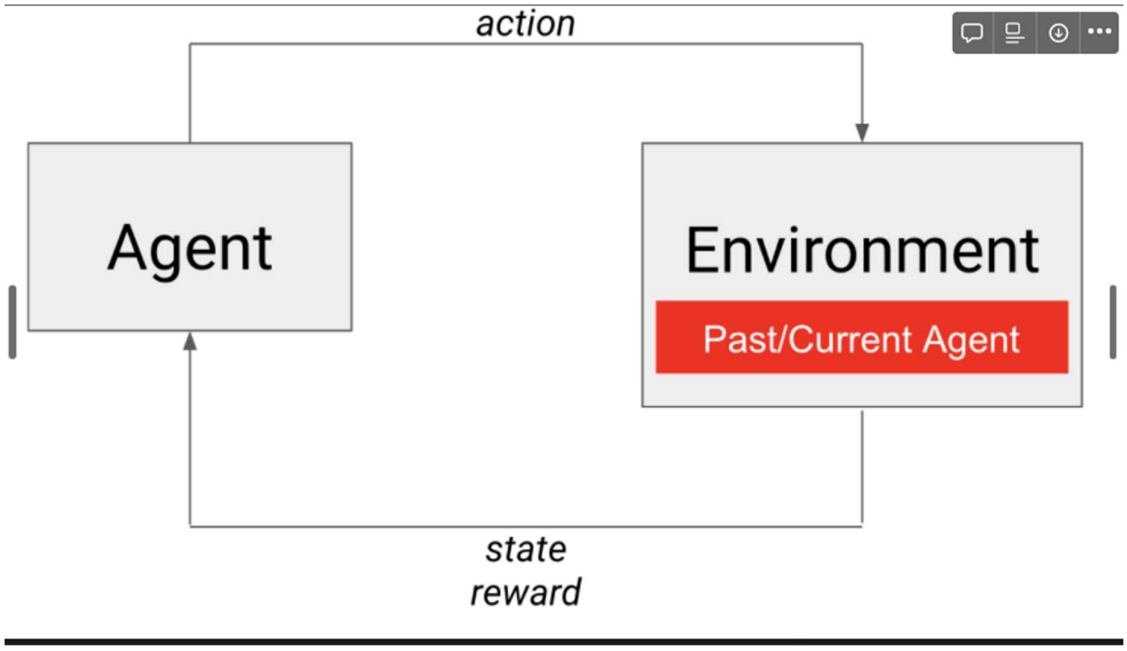

Figure 2: Reinforcement Learning in adversarial games

- **Deep Learning**: Deep learning is a subset of machine learning, which is essentially a neural network with three or more layers. These neural networks attempt to simulate the behaviour of the human brain—albeit far from matching its ability—allowing it to "learn" from large amounts of data. While a neural network with a single layer can still make approximate predictions, additional hidden layers can help to optimize and refine for accuracy.
- **Player Modelling and Human AI Interactions:** Social gaming, such as the Battle-Royale genre and Animal Crossing, has gained increasing popularity. Combined with heterogeneous data provided on social media and streaming platforms, understanding, and predicting players' behaviour patterns considering graph structures becomes increasingly important. The data provided by major AAA games will offer resources to imitating and modelling human behaviours (Sapienza et al. 2018; Zeng 2020) and facilitate understanding of human collaborations (Zeng, Sapienza, and Ferrara 2019). Gaming industry with exuberant data of in-game human collaborations makes suitable sand-box environments for conducting multi-agent interaction/collaboration research. For instance, multi-agent Hide-and-Seek (Baker et al. 2019), OpenAI Five (Berner et al. 2019), AlphaStar (Vinyals et al. 2019), Hanabi (Bard et al. 2020) and capture the flag (Jaderberg et al. 2019) are some initial attempts. With detailed human behaviour trajectory recorded as replays or demos, gaming environments provide data-intensive sources for human-computer interaction research. Recent

advancements of AI in games have evolved human computer interactions in gaming environments into human-NPC interactions. As suggested in paper (Risi and Preuss 2020), with the increasing popularity in human/AI interactions, we will see more research on human-like NPC and human-AI collaboration in the future.

- **GAIL (Generative Adversarial Imitation Learning):** GAIL, or, uses an adversarial approach to reward your Agent for behaving like a set of demonstrations. GAIL can be used with or without environment rewards and works well when there are a limited number of demonstrations. In this framework, a second neural network, the discriminator, is taught to distinguish whether an observation/action is from a demonstration or produced by the agent. This discriminator can then examine a new observation/action and provide it a reward based on how close it believes this new observation/action is to the provided demonstrations. At each training step, the agent tries to learn how to maximize this reward. Then, the discriminator is trained to better distinguish between demonstrations and agent state/actions. In this way, while the agent gets better and better at mimicking the demonstrations, the discriminator keeps getting stricter and stricter and the agent must try harder to "fool" it. This approach learns a *policy* that produces states and actions like the demonstrations, requiring fewer demonstrations than direct cloning of the actions. In addition to learning purely from demonstrations, the GAIL reward signal can be mixed with an extrinsic reward signal to guide the learning process. 2018).
- **SIMULATED INTERACTIVE ENVIRONMENTS AND BEYOND:** Playtesting, matchmaking, dynamic difficulty adaptation (DDA) are some other important tasks for gaming industry to solve using machine learning. Beyond gaming, interactive environments are used to mimic real-life scenes such as training robots or autonomous vehicles. Interactive gaming environments can also serve as demonstrations for game theory decision makings that serve AI for social good initiatives.
- **VECTORIZED ENVIRONMENTS:** Vectorized Environments are a method for stacking multiple independent environments into a single environment. Instead of training an RL agent on 1 environment per step, it allows us to train it on n environments per step. Because of this, actions passed to the environment are now a vector (of dimension n). It is the same for observations, rewards, and end of episode signals (Dones). In the case of non-array observation spaces such as *Dictionary* or *Tuple*, where different sub-spaces may have different shapes, the sub-observations are vectors (of dimension n).

- **States and Observations:** A state '*s'* is a complete description of the state of the world. There is no information about the world which is hidden from the state. An observation o is a partial description of a state, which may omit information. In deep RL, we almost always represent states and observations by a real-valued vector, matrix, or higher-order tensor. For instance, a visual observation could be represented by the RGB matrix of its pixel values; the state of a robot might be represented by its joint angles and velocities. When the agent can observe the complete state of the environment, we say that the environment is fully observed. When the agent can only see a partial observation, we say that the environment is partially observed.
- **PPO (Proximal Policy Optimization)** is a reinforcement learning algorithm that can be used in combination with imitation learning to improve the learning process. In the context of combining PPO and imitation learning, PPO is used to fine-tune the learned policy by refining the policy that was initially learned through imitation.
- **Imitation Learning -** By using PPO to refine the policy learned through imitation, the resulting policy can achieve better performance than either method used alone. This approach is sometimes referred to as "imitation learning with fine-tuning".

2.3LITERATURE SURVEY

2.3.1 EXISTING SYSTEMS

There has been a significant amount of research conducted in the field of reinforcement learning applied to video games, specifically in the context of autonomous agents. One notable example is the work of (Mnih, et al., 2015) in which they used a deep reinforcement learning algorithm to play a range of Atari games at a superhuman level. Similarly, (Hester, et al., 2018) used deep reinforcement learning to train agents to play the game of StarCraft II, achieving performance levels comparable to professional human players.

In the context of aviation, there has been limited research conducted on the application of reinforcement learning. One example is the work of in which they used a Q-learning algorithm to train a single agent to fly an aircraft in a simulated environment. However, there has been no prior work found on the use of reinforcement learning in the context of a multiplayer aviation game.

Thus, a careful study of the literature suggests that reinforcement learning has great potential for use in autonomous agents, as demonstrated by the success in the gaming industry. However, the

application of these algorithms to aviation is relatively unexplored, providing an opportunity for further research and experimentation.

2.3.2 HISTORICAL LITERATURE

Reinforcement Learning (RL) is a popular machine learning technique that enables agents to learn from interactions with the environment and maximize a reward signal. RL has been widely studied and implemented by researchers and practitioners, and there are numerous surveys available that provide an overview of the field.

One example is the survey by Arulkumaran et al. (1), which covers RL algorithms and their applications in various domains, such as healthcare, robotics, and finance. The authors provide a detailed description of several RL algorithms, including Q-learning, Monte Carlo methods, and value-based methods like DQN and DDQN. The survey also includes a discussion of Deep RL and its applications, as well as a comparison of different RL algorithms based on their strengths and limitations.

Another notable survey is the one by (Kober & Peters, 2010), which focuses on RL in robotics. The authors provide an overview of different RL algorithms and their applications in robotics, including control, mapping, and perception tasks. The survey also covers the challenges and open problems in RL for robotics, such as scalability and safety.

In the healthcare domain, (Rajan & Hutter, 2019) provide a comprehensive survey of RL applications, including dynamic treatment regimes, automated medical diagnosis, and other general domains. The authors also discuss the challenges and opportunities of using RL in healthcare, such as the need for interpretable and safe RL algorithms.

In terms of RL algorithms, the survey by (Kubat, 1999) is a classic reference. The authors provide a comprehensive overview of RL algorithms, including Q-learning, SARSA, and policy gradient methods, and their theoretical foundations. The survey also covers advanced topics such as eligibility traces and function approximation in RL.

Lastly, the survey by focuses on the Proximal Policy Optimization (PPO) algorithm, which is a popular RL algorithm that has demonstrated high sample efficiency and robustness. The authors provide a detailed description of the algorithm and its variants, as well as its applications in various domains, such as robotics, gaming, and natural language processing.

A careful look at the Historical literature shows that RL is a rapidly evolving field with a growing number of applications and research directions. The surveys discussed above provide a useful starting point for understanding the state of the art in RL and its applications.

2.4OVERALL DESCRIPTION

The final software which consists of the following components:

- A Loading Screen
- The Main Menu
- The Game Environment
- And a quitting option.

THE LOADING SCREEN: This component will be displayed when the game is launched. It will have an animation and display the name of the game and the developer's logo. The purpose of this component is to create a first impression for the user and give them an idea of what the game is about.

Figure 3: Loading Screen

THE GAME MENU: This component will be displayed after the loading screen. It will contain options for starting a new game, selecting the difficulty level, and accessing the settings menu. The purpose of this component is to allow the user to interact with the game and customize their experience. Level of difficulty would make the opponent AI

fly in a much better way.

Figure 4: The Main Menu

THE GAME ENVIRONMENT/ACTUAL GAME SCENE: This component will contain the 3D models of the aircraft, game terrain, and other game objects such as checkpoints, power-ups, etc. It will also have a camera that follows the player's aircraft. The purpose of this component is to provide an immersive experience for the user and allow them to fly their aircraft through a virtual world.

QUITTING OPTION: This component will allow the user to exit the game. It will be accessible from the main menu and pause menu during gameplay. The purpose of this component is to give the user a way to exit the game easily.

2.5SPECIFIC REQUIREMENTS

In the three-dimensional scene that we have created, we have pointed out three kinds of stakeholders: Builder, User and Tester. Builder and Tester also have different roles. We define the stakeholders with the roles as the following table.

Table 2: Various Stakeholder involved.

Stakeholder	Role	Task
Builder	Software engineer	Creating different scripts for different functioning of the three-dimensional models, and different actions of the components like airplane Controller, PlayerFunction, etc.
	3d Artist, Unity expert	Creating the three-dimensional assets, the game environment, different models which will in turn be turned into assets to be used in the development of the Game.
User	Main User	Flying the agent (i.e., the plane across the course) and flying in the way that it can pass through the checkpoints and successfully complete the laps, two of them, and on top of that attempt to beat the ML-agent trained Adversarial aircrafts.
Tester	System Tester	Testing the validation and the effectiveness of the system.
	Quality Assurance Manager	Checking the validation of information, (both questions and knowledge). Moving the invalid information from the database.

2.5.1 OVERVIEW OF FUNCTIONAL REQUIREMENTS

Purpose: To be able to make the agents fly without abandoning path, and moving in a right way, and allow the player agent to take input and directions from the user or the player

Inputs: The main input would be the ML-agents trained and deep reinforcement learning algorithm trained neural net which would govern the behaviour of the adversarial aircrafts

Outputs: A proper flight behaviour of the respective agents.

2.5.2 OVERVIEW OF DATA REQUIREMENTS

The data requirements include the training data for the neural network and deep reinforcement learning models which was mostly derived from flying the aircraft, and using the required imitation learning algorithms incorporated into the ML-agents libraries or framework.

USER VIEW OF THE PRODUCT USED

1. The Loading Screen
2. The Game Menu
3. The Game Sequence or the Game Environment Scene
4. The quit screen.

2.6REQUIREMENTS

2.6.1 FACILITIES REQUIRED FOR THIS PROJECT

2.6.1.1 DEVELOPER HARDWARE REQUIREMENTS

The project utilizes the hardware resources which are easily available and can be easily utilized to find the correct balance between the two and this is the whole point of the thing being carried out to the extent of oblivion. More importantly the project needs to use GPU's as it runs on GPU's to plot the features and predict the accuracy. To be a bit more precise for you to run this locally on your machine the minimum system requirements are –

Table 3: Developer Hardware Requirements

S.no	Component	Minimum Requirement
1	Processor	64-bit, Quad-Core, 3.50 GHz minimum Clock speed
2	RAM	16 GB
3	Graphics Card	NVIDIA GTX 1650 and above
4	HDD	Min 256 Gb
5	SSD	High capacity (greatly improves training speed and performance of the agents)

2.6.1.2 DEVELOPER SOFTWARE REQUIREMENTS

The software development process for this project requires several software components to be installed on the developer's machine. These include the Blender 3D modeling software for creating and modifying 3D models and assets, the Unity game engine for creating the game environment and implementing game logic, C# for scripting and automation tasks, Visual Studio Code for writing and debugging code, and Notion for project management and collaboration.

Table 4: Developer Software Requirements

S.no	Requirement Description	Component	Version
1	3D modelling software	Blender	3.1 or above -> LTS
2	Game engine software	Unity Hub	3.3.0 or above
3	Programming language	Python	3.7.7
4	Code editor	VS Code	1.64.2
5	Project management tool	Notion	1.24.1
6	Graphics library	Unity	2020.3.30f1
7	Data analysis toolkit	Anaconda3	2021.11

2.6.1.3 USER HARDWARE REQUIREMENTS

Table 5: User Hardware Requirements

S.no	Component	Minimum Requirement
1.	Processor	64-bit, Quad-Core, 2GHz minimum Clock speed
2.	RAM	8 GB
3.	Graphics Card	2 Gigs minimum, 4 GB NVIDIA GTX 1080 recommended.
1.	OS	Windows 10 or above
2.	HDD	Min 256 Gb
3.	SSD	Not Required

CHAPTER 3: INTRODUCTION AND DETAIL OF SOFTWARE USED

3.1 TECHNOLOGIES USED / FRAMEWORK USED

1. **Imitation Learning**: Imitation learning was used to teach the AI agents how to fly through the course with accuracy and near-perfect flight paths. The learning data was used to train the adversarial agents.
2. **Deep Reinforcement Learning**: Deep reinforcement learning was used to train the AI agents to adapt and learn from their actions in real-time scenarios. The airplanes in this environment fly freely in space using "Raycast" vision and use reinforcement learning to train along the flight path.
3. **Human-Controlled Aircraft vs. Neural Network Trained Adversaries**: An important aspect of the project was the inclusion of a Human-Controlled Aircraft, which competed against the agents trained using imitation learning and deep reinforcement learning algorithms. This created a multi-agent environment that simulated real-world scenarios, providing an engaging and realistic hands-on experience for the players.
4. **Blender**: Blender is the free and open-source 3D creation suite. It supports the entirety of the 3D pipeline—modelling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation. Advanced users employ Blender's API for Python scripting to customize the application and write specialized tools; often these are included in Blender's future releases.
5. **Hyperparameter Tuning:** To optimize the performance of the AI agents, hyperparameter tuning was performed. This involved adjusting various parameters such as the learning rate, discount factor, and the number of hidden layers in the neural network to improve the agents' performance.
6. **Unity Game Engine**: The game will be developed in the Unity engine. Unity is a Cross-Platform game engine developed by Unity technologies. Unity gives the ability to create games and experiences in both 2D and 3D, the Primary Scripting API in C#, and it provides several functionalities, like drop-down functionality.
7. **Unity ML-Agents**: Unity ML-agents is an open-source framework for creating a state-of-the-art deep learning technology to create complex AI environments and an intelligent game experience. The ML-Agents Toolkit allows researchers and game developers and researchers to build and train agents in Unity environments using Reinforcement Learning (RL), specifically using visual or numerical based observations. While numerical observations like distance and direction to a target allow the agent to learn very

quickly, using numerical observations is limited and does not support a diverse range of problem areas. Visual observations are better suited in situations where the agent's state and environment are difficult to quantify. That's why the ML-agents toolkit contains a sensor that allows creators to use the output of their own pretrained CV models as observations.

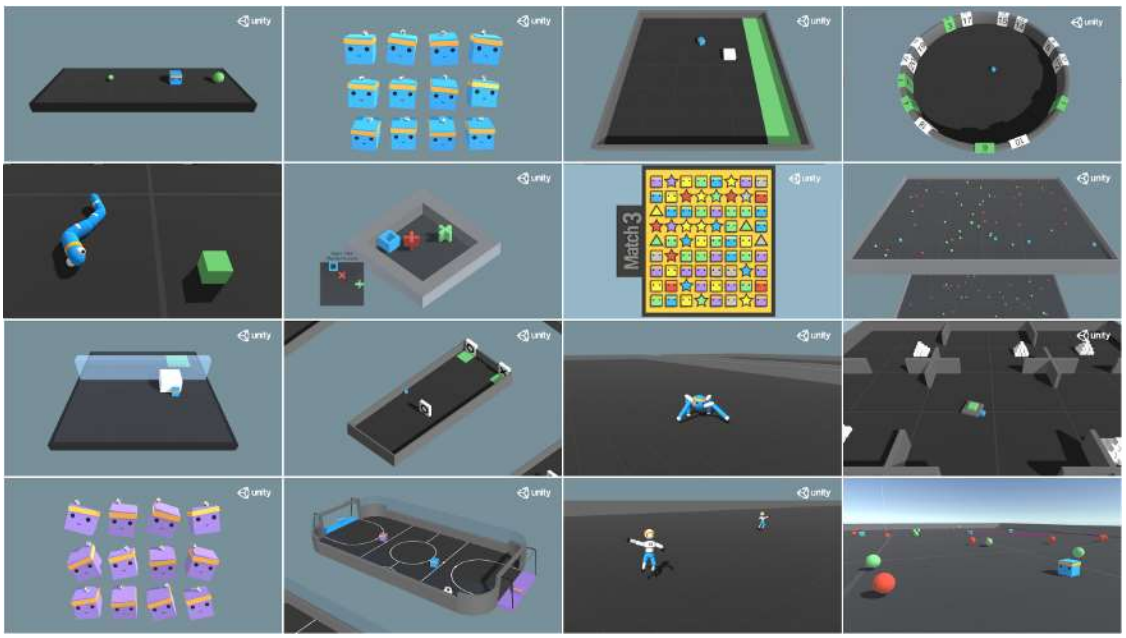

Figure 5: Unity ML-Agents -> Courtesy Unity:

8. **CONDA:** Conda is an open-source package and environment management system that runs on Windows, macOS, and Linux. Conda quickly installs, runs, and updates packages and their dependencies. It was created for Python programs, but it can package and distribute software for any language.
9. **TENSORFLOW:** TensorFlow makes it easy for beginners and experts to create machine learning models.
10. **TENSORBOARD Visualizations**: TensorBoard provides the visualization and tooling needed for machine learning experimentation:
    - Tracking and visualizing metrics such as loss and accuracy
    - Visualizing the model graph (ops and layers)
    - Viewing histograms of weights, biases, or other tensors as they change over time

CHAPTER 4: SYSTEM DESIGN

4.1 PROBLEM DESCRIPTION

The problem at hand is to create a sustainable environment and provide the agents with a relatively feasible scene so that the different agents, and adversaries (i.e. the other aircrafts can be populated into the scene or the environment and the so called environment), and then to provide the planes, and the agents with the correct incentives so that the agents are incentivized enough to follow the flight path, and fly a flawless lap, without bumping haphazardly into each other or the parts of the environment. Following that training the agents using the correct deep reinforcement learning techniques and incorporating those in the proper neural network model so that the work so that the other adversaries can fly the course in near optimal times. The game environment comprises of different objects which are scattered across, making for a diverse environment. It comprises of two different scenes, one of a Desert environment, and the other which consists of a snowy environment, it also uses different difficulty levels in which the Neural Network trained agents fly in varying levels of proficiency – one being "hard", and the other being "normal". We have put planes into the environment in which one plane is controlled by the player, i.e., the customer, or the gamer, or us – the user. The goal of the game is to beat the other three Intelligent Agents, while passing different checkpoints, failing which the user would be reset to the last available checkpoint. The player which finishes two laps in the least time split wins.

4.2 PROCESS DESCRIPTION / SOLUTION FRAMEWORK

4.2.1 THE ML-AGENTS FRAMEWORK ARCHITECTURE WHICH IS USED FOR THE GAME

The ML-agents framework contains 5 components:

- **Learning Environment** - which contains the Unity scene and all the game characters. The Unity scene provides the environment in which agents observe, act, and learn. How you set up the Unity scene to serve as a learning environment really depends on your goal. You may be trying to solve a specific reinforcement learning problem of limited scope, in which case you can use the same scene for both training and for testing trained agents. Or you may be training agents to operate in a complex game or simulation. In this case, it might be more efficient and practical to create a purpose-built training scene. The ML-Agents Toolkit includes an ML-Agents Unity SDK (`com.unity.ml-agents` package) that enables you to transform any Unity scene into a learning environment by defining the agents and their behaviours.

The Learning Environment contains two Unity Components that help organize the Unity scene:

- **Agents** - which is attached to a Unity GameObject (any character within a scene) and handles generating its observations, performing the actions it receives and assigning a reward (positive / negative) when appropriate. Each Agent is linked to a behaviour.
- **Behaviour** - defines specific attributes of the agent such as the number of actions that agent can take. Each behaviour is uniquely identified by a `behaviour Name` field. A behaviour can be thought as a function that receives observations and rewards from the Agent and returns actions. A behaviour can be of one of three types: Learning, Heuristic, or Inference. A Learning behaviour is one that is not, yet, defined but about to be trained. A Heuristic behaviour is one that is defined by a hard-coded set of rules implemented in code. An Inference behaviour is one that includes a trained Neural Network file. In essence, after a Learning behaviour is trained, it becomes an Inference behaviour.

  In a single environment, there can be multiple Agents and multiple Behaviours at the same time. For example, if we expanded our game to include tank driver NPCs, then the Agent attached to those characters cannot share its behaviour with the Agent linked to the medics (medics and drivers have different actions). The Learning Environment through the Academy (not represented in the diagram) ensures that all the Agents are in sync in addition to controlling environment-wide settings.

  Lastly, it is possible to exchange data between Unity and Python outside of the machine learning loop through *Side Channels*. One example of using *Side Channels* is to exchange data with Python about *Environment Parameters*

- **Python Low-Level API** - which contains a low-level Python interface for interacting and manipulating a learning environment. Note that, unlike the Learning Environment, the Python API is not part of Unity, but lives outside and communicates with Unity through the Communicator. This API is contained in a dedicated `mlagents_envs` Python package and is used by the Python training process to communicate with and control the Academy during training. However, it can be used for other purposes as well.

- **External Communicator** - which connects the Learning Environment with the Python Low-Level API. It lives within the Learning Environment.

- **Python Trainers** which contain all the machine learning algorithms that enable training agents. The algorithms are implemented in Python and are part of their own `mlagents` Python package. The package exposes a single command-line utility `mlagents-learn` that supports all the training methods and options outlined in this document. The Python Trainers interface solely with the Python Low-Level API.

- **Gym Wrapper** (not pictured). A common way in which machine learning researchers interact with simulation environments is via a wrapper provided by OpenAI called gym. We provide a gym wrapper in a dedicated `gym-unity` Python package and instructions for using it with existing machine learning algorithms which utilize gym.

The following diagram illustrates the above.

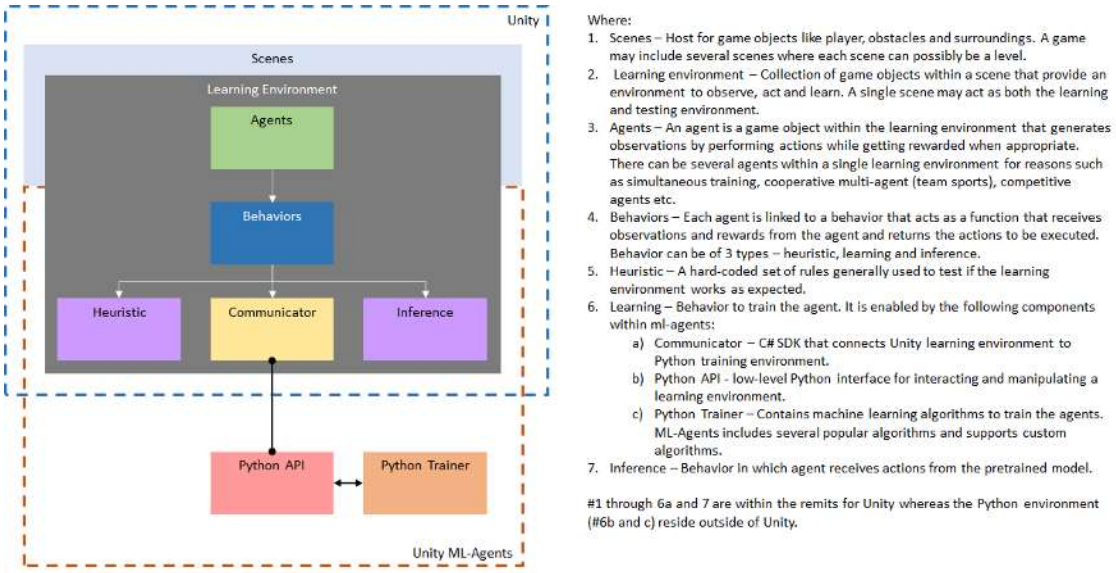

Figure 6: The Framework of the AI Game Model

4.2.2 FLOWCHART FOR THE PROJECT

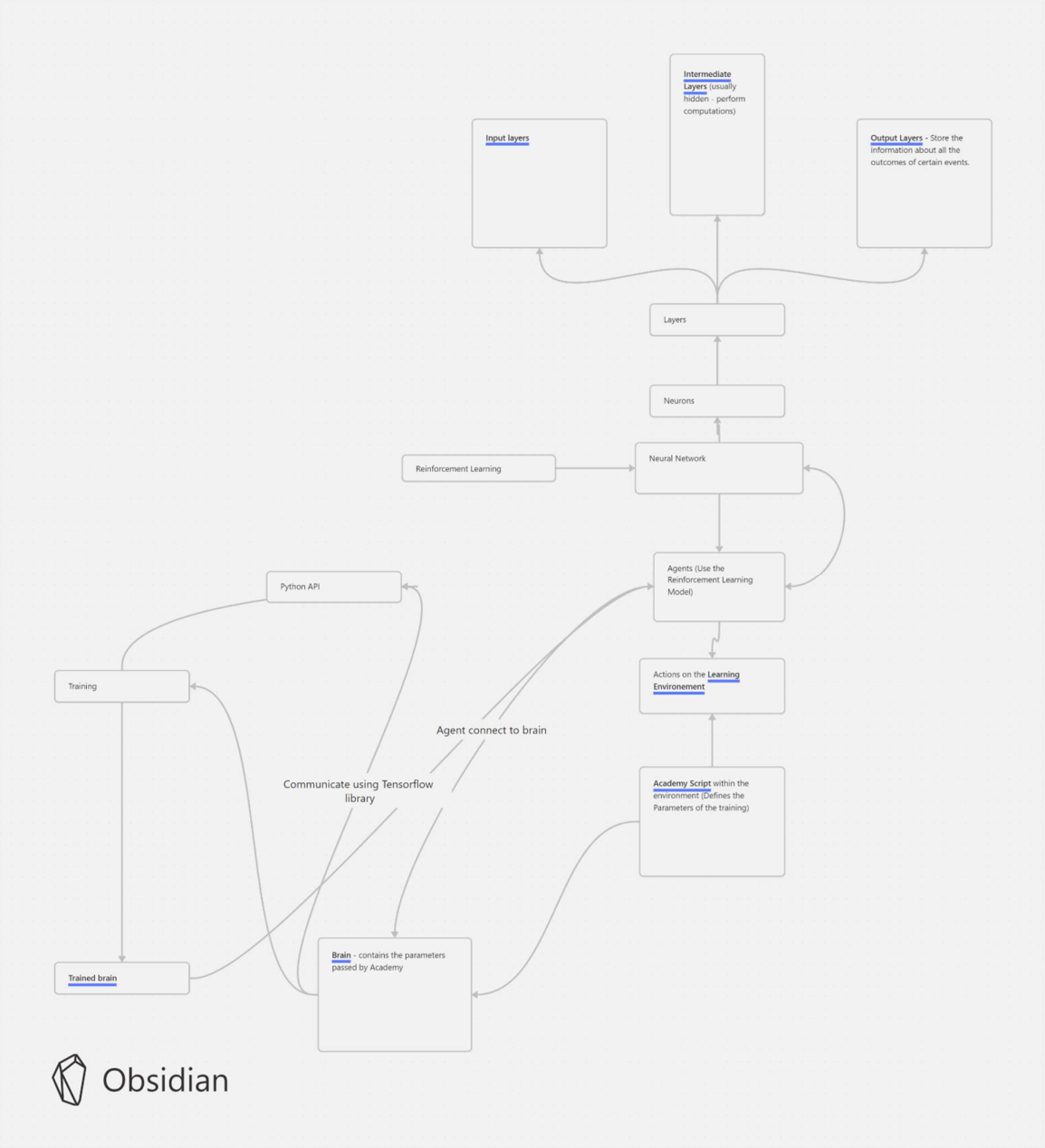

Figure 7: Flowchart including the Neural Net and the Agents and their relationship.

4.2.3 FEATURES

- **Markov Decision Process (MDP)**: The project is based on an MDP framework, which is a mathematical model for decision-making that involves making decisions based on the current state of the system and maximizing a cumulative reward over time.

- **Deep Reinforcement Learning (DRL)**: The project utilizes DRL, which involves using neural networks to approximate the Q-values (expected rewards) of actions in an environment, allowing for more complex and high-dimensional state spaces.
- **Actor-Critic Architecture**: The project employs an actor-critic architecture, which consists of two neural networks - one for estimating the policy (actor) and another for estimating the state-value function (critic).
- **Policy Optimization**: The project uses a policy optimization approach, where the actor network is trained to improve the policy by maximizing the expected reward.
- **Proximal Policy Optimization (PPO)**: The project specifically utilizes PPO, a state-of-the-art policy optimization algorithm, which balances the exploration-exploitation trade-off and ensures the policy update does not deviate too far from the previous policy.
- **Multi-Agent Environment**: The project is developed in a multi-agent environment, where multiple agents interact with each other and the environment, requiring additional considerations for training and coordination.
- **Simulation Environment**: The project uses a simulation environment to train and evaluate the RL algorithms, allowing for more efficient and safe experimentation.
- **Evaluation Metrics**: The project employs various evaluation metrics, such as average reward, success rate, and convergence rate, to assess the performance of the RL algorithms and compare different approaches.
- **Real-World Applications**: The project aims to apply RL to real-world problems, such as autonomous flight, and robotics and can find application in the Aviation sector, or drone sector.

CHAPTER 5: SYSTEM IMPLEMENTATION AND TESTING

5.1IMPLEMENTATION

The implementation of the game was carried out in multiple stages, following a development process that was centered around agile principles. The development team consisted of two members, me and Naman.

The project was developed using Unity game engine version 2020.3.12f1 and Python version 3.9. The Unity ML-agents toolkit was used for the development of the artificial intelligence components.

The game was developed in stages, with each stage building on the previous one. The initial stage involved creating a basic environment and a simple game mechanic that allowed the player to interact with the environment. In the subsequent stages, the team added more complex game mechanics, multiple levels, and the artificial intelligence component.

The development process involved multiple iterations and feedback loops, with each iteration focusing on improving specific aspects of the game. For example, the team spent one iteration focused on improving the performance of the artificial intelligence, while another iteration was focused on improving the user interface.

The implementation was tested at each stage to ensure that it met the requirements outlined in the project specifications. Testing involved a combination of automated tests and manual testing. Automated tests were created for the game mechanics and the artificial intelligence component. Manual testing was carried out by the development team, as well as by external testers who were brought in to provide feedback on the game.

Overall, the implementation process was successful, and the final product met all the requirements outlined in the project specifications.

5.1.1 IMPLEMENTATION STRATEGY

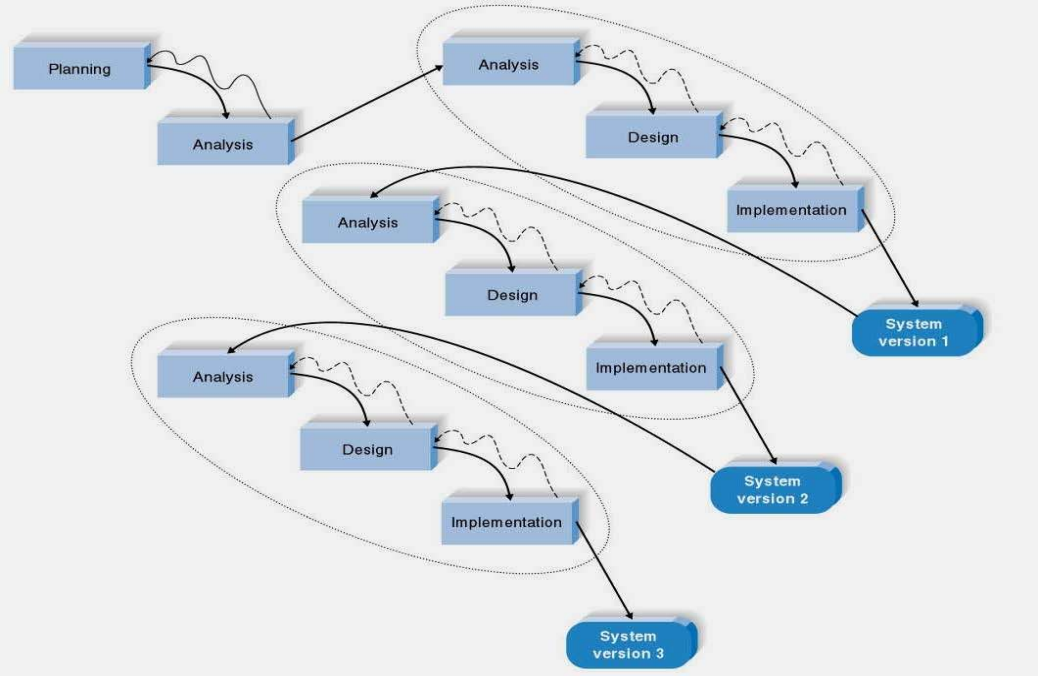

Figure 8: The Iterative Strategy Used while developing the ML-Agents Project

Comparison of the ways that can be used in implementing or introducing a newsystem is shown in below Table 6

Table 6: Comparison of Implementation Strategies which were considered.

Method	**Main Advantages**	**Main Disadvantages**
1. Immediate cutover Straight from old system to new system on a singledate	Rapid, low cost	High risk. Major disruption if serious errorswith the system
2. Parallel running Old system and new system run side-by-sidefor a period	Low risk than immediatecutover	Slower and higher cost than immediate cutover
3. Phased ImplementationDifferent modules of the system are introduced sequentially	Good compromise between methods 1 and 2	Difficult to achieve due tointerdependencies between modules
4. Pilot System Trial implementation occurs before widespreaddeployment		Has to be used in combination with theother methods.

The various implementation approaches for building a three-dimensional game with Unit ML-agents include selecting the correct Reinforcement Learning algorithm and figuring out the right reward system, for the better flight for the adversarial AI planes.

CHAPTER 6: IMPLEMENTATION OF THE MODULES

6.1 STEPS INVOLVED IN THE IMPLEMENTATION

1. Setting up the development environment.
2. Create a Unity 3D project.
3. Define a Scene as environment for the agent (plane in this case) for training and testing.
4. Add observations, rewards, and behaviour to the plane.
5. Test the game with user inputs (heuristic).
6. Train the plane (learning).
7. Deploy the model to draw inferences based on the pretrained model.

SETTING UP THE DEVELOPMENT ENVIRONMENT

Creating a development environment is the first step towards implementing any project. For this project, the development environment could be created using software tools like Visual Studio or any other integrated development environment (IDE) that supports the Unity3D framework. The development environment must be configured with the required Unity3D version and any additional software packages or plugins required to build and run the project.

The following Software components were used –

To create the development environment, the following tools were utilized:

1. Unity – Version 2021.1.22f1 was used to set up the project.
2. Python – Anaconda was used to create a v3.7 virtual environment.
3. Code Editor – Visual Studio was used as the code editor.
4. ML-Agents – Release_13 was used, and the com.unity.ml-agents package was cloned from the repo and added to the project from disk, along with PyTorch and ml-agents python packages.

CREATE A UNITY 3D PROJECT.

Creating a new Unity 3D project was a straightforward process. The following steps were taken:

- Launch Unity Hub.
- Click on "New" and select the desired Unity version.

- Select "3D Template" and provide a project name.

- Configure any related settings as necessary.
- Click "Create."

DEFINE THE SCENE

- **Modelling the environment**

  The airplanes were modelled, and the environment was set up.

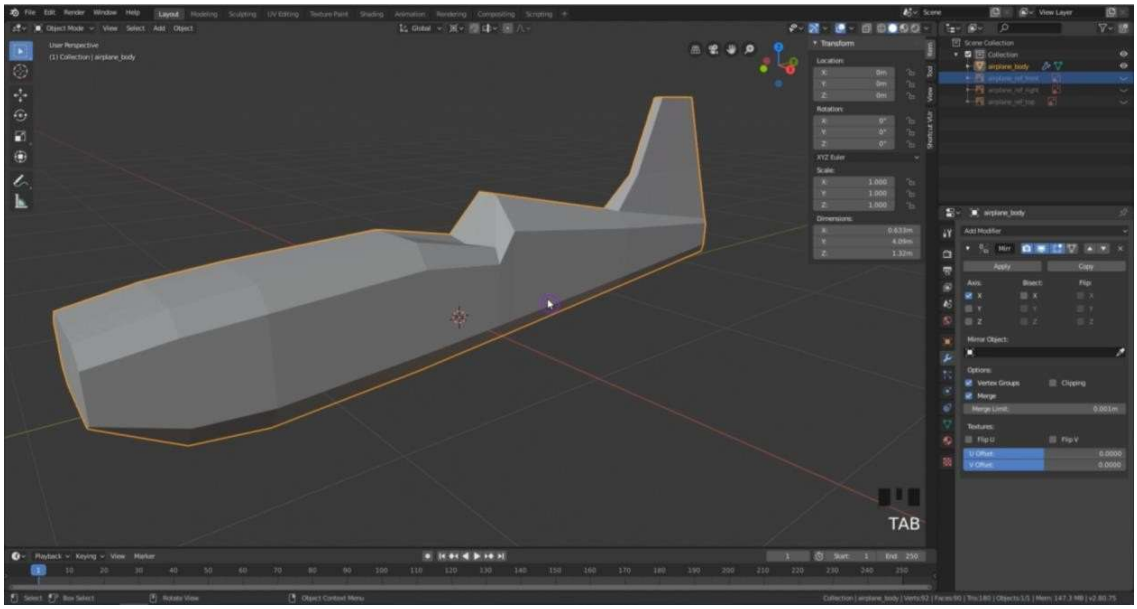

Figure 9: Modelling the Airplane

- **Defining the Scene for the Agent**

Figure 10: File structure

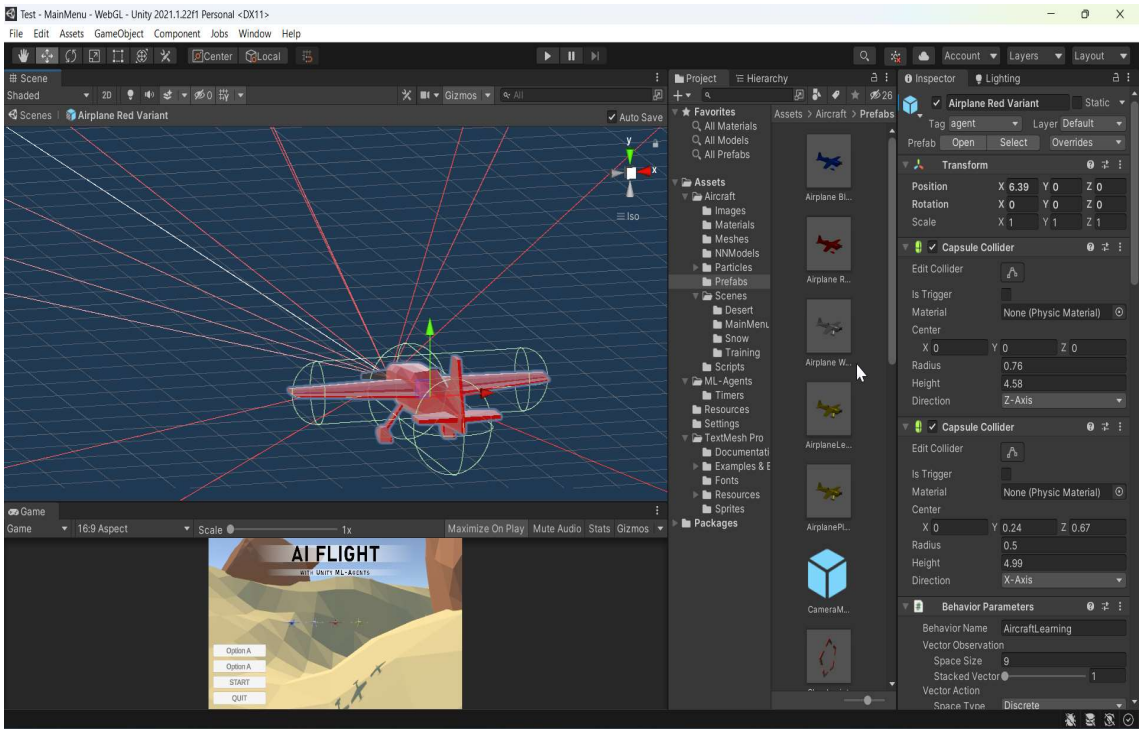

Figure 11: Rigging the Agents with different components.

The following steps were taken:

1. Add the platform: From the File panel, add GameObject > 3D Object > [Blender Asset]. In the Inspector, set the asset's Transform properties to Position X=0, Y=-4, Z=0, and Scale X=12, Y=1, Z=1. In the Inspector, add all the required components from Component > Physics 3D.
2. Add the airplanes: From the File panel, add GameObject > 3D Object > Sphere. In the Inspector, set the Sphere's Transform properties to Position X=0, Y=0, Z=0 (to create ball falling from top effect whenever game starts/resets) and Scale X=1, Y=1, Z=1. Add the respective physics components and the scripts for the airplanes.

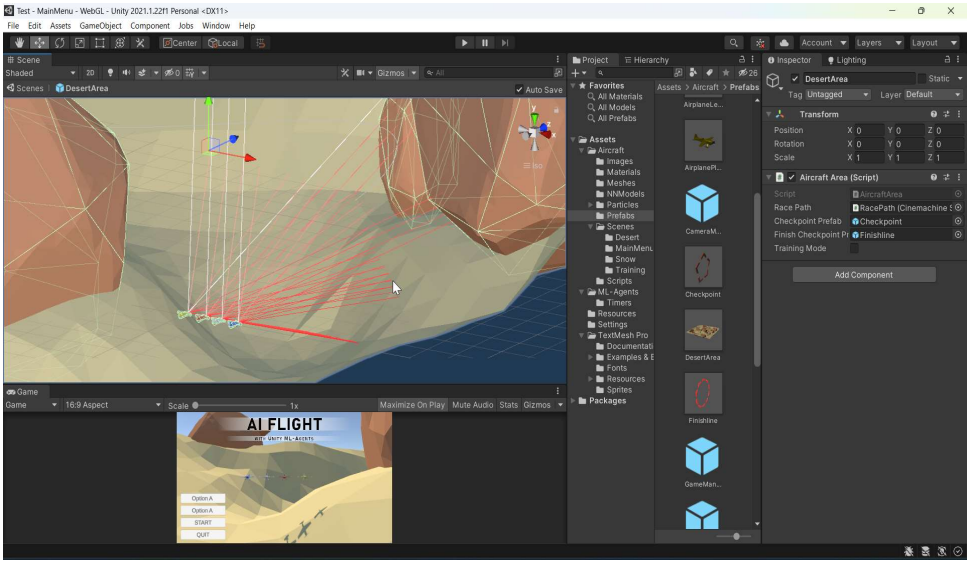

Figure 12: Aircraft Agents Setup

3. Add checkpoints and path objects: To detect collisions with the planes, enable the Collision property of Particle System with Type as World. To allow some time before the

particles start appearing, add Start Delay as 1 in Particle System properties in Inspector. Lastly, change the start speed to 0.1 in Particle System properties in Inspector.

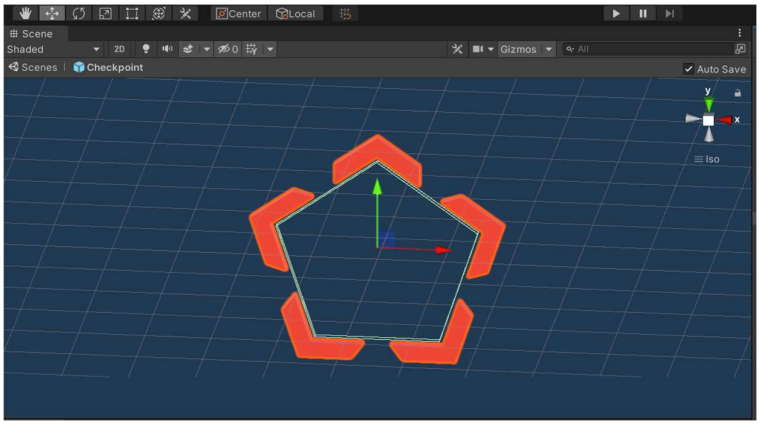

Figure 13: Adding Checkpoint to the Environment.

4. Group all the game objects: Create an empty game object in Hierarchy pane and name it TrainingArea. Set its position parameters to X=0, Y=0, and Z=0. Drag the aircraft, platform, and particle system into TrainingArea.

- **ADD OBSERVATIONS, REWARDS AND BEHAVIOURS**

There are a ton of scripts and components which needed to be developed here are some of them. The project setup and file system contained the following scripting files.

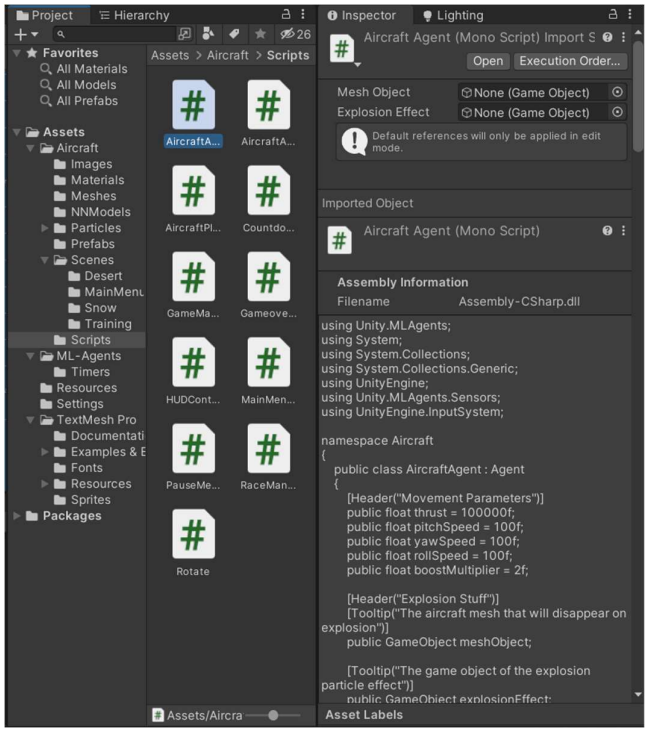

Figure 14: Various Scripting Components

- **TESTING THE GAME WITH USER INPUTS**

Before running the game, the Behaviour Parameters > Behaviour Type was changed to Heuristic to indicate that the agent actions would be guided manually.

- **TRAINING THE AGENTS (LEARNING)**

The training scripts and values were set up in an appropriate *'.yaml'* file named *"trainer_config.yaml"*. The PPO training model was mainly used for the given environment scenario, but other models and configurations to optimize the training process were also explored.

6.2 MODULE IMPLEMENTATION

In this section, we will discuss the implementation of the different modules used in the project.

AircraftAgent.cs

The airplanes in the system were modelled, and the environment was set up. The AircraftAgent script is responsible for controlling the behavior of the player's aircraft.It uses the Heuristic() method to handle the movement of the aircraft. The following snippet shows how the Heuristic () method updates the aircraft's position based on user input.

This C# script for a Unity game uses reinforcement learning to train an aircraft agent to fly through checkpoints, with functions for movement, observation, rewards, and detection of checkpoint completion, as well as the ability to freeze and resume the agent.
Parameters designed - Thrust, pitch, Speed and Roll, also a **step time-out parameter** for training.
Position and Velocity of the agent

- By using **Rigidbody** component.
- Trail is tracked by **Trailrenderer** component.

Various Functions Used

- **Initialize()** function sets up the initial values of the agent including the area it is in, while RigidBody and Trailrenderer component includes the many steps it can take.
- **OnActionReceived()** -Reads the agent's actions for pitch, yaw, and boost, processes the movement, adds rewards or penalties depending on the agent's performance, and progresses to the next checkpoint. Receives input from the array.
- **CollectObservations( )-** collects the agent's observations, which include its velocity, the location of the next checkpoint, and the orientation of the next checkpoint.
- **OnEpisodeBegin()** function resets the agent's velocity, position, and orientation, and turns off the trail. It also updates the step timeout if the agent is in training mode.
- **Heuristic()** function is only called for the AircraftPlayer, and it returns an empty array. I created it for Testing and debugging purposes.
- **FreezeAgent()** function stops the agent from all actions and ThawAgent() function resumes the agents movement and actions.

An Integral Component of the AircraftAgent Script is shown here -

```
public override void Heuristic(float[] actionsOut)
{
    actionsOut[0] = Input.GetAxis("Horizontal");
    actionsOut[1] = Input.GetAxis("Vertical");
    actionsOut[2] = Input.GetAxis("Fire1");
}
```

AircraftArea.cs

The `AircraftArea` script defines the area in which the game takes place. It is responsible for spawning obstacles and power-ups, as well as checking for collisions between the aircraft and other objects in the scene. The following snippet shows how the `SpawnObstacle()` method is used to create obstacles in the game:

```
private void SpawnObstacle()
{
    // Get a random obstacle from the list of obstacles
    int obstacleIndex = Random.Range(0, obstacles.Count);
    GameObject obstacle = obstacles[obstacleIndex];

    // Instantiate the obstacle at a random position
    Vector3 position = new Vector3(Random.Range(minX, maxX),
Random.Range(minY, maxY), transform.position.z + spawnDistance);
    Instantiate(obstacle, position, Quaternion.identity);
}
```

It also allows for randomization of the checkpoint an agent starts at. It tracks all the aircraft agents in the area and sets their initial positions and rotations on the race path. The script is also responsible for instantiating the checkpoints, which can be either a regular checkpoint or a finish line checkpoint.

Aircraft Player.cs

- The script inherits from a base class called `AircraftAgent`, which likely contains functionality for controlling the aircraft based on input.
- The `InputAction` variables at the top of the script define the input bindings for the player's pitch, yaw, boost, and pause inputs.
- The `Initialize` method overrides the base class's `Initialize` method and enables the input actions.
- The `Heuristic` method reads the player's input values for pitch, yaw, and boost and converts them to discrete values for `AgentAction` to use.
- The `OnDestroy` method disables the input actions when the script is destroyed.

CountdownUIController.cs

The script includes a public **TextMeshProUGUI** variable named *`countdownText`*, which is a reference to the **TextMeshProUGUI** object that will display the countdown text.

The script also includes a public **IEnumerator** method named *`StartCountdown`*, which is a coroutine that will display a countdown on the *`countdownText`* object. The countdown starts with "3" displayed for 1 second, followed by a brief pause with an empty string displayed, then "2" displayed for 1 second, another pause, "1" displayed for 1 second, another pause, and finally "GO!" displayed for 1 second before disappearing.

RaceManager.cs

The `RaceManager` script is responsible for managing the race and keeping track of the player's progress. It updates the player's lap time and position, and checks for when the race is complete. The following snippet shows how the `UpdateLapTime()` method updates the player's lap time:

An Integral Component of the RaceManager.cs Script is shown here -

```
private void UpdateLapTime()
{
    if (startRace && !raceComplete)
    {
        currentLapTime += Time.deltaTime;
        lapTimeText.text = FormatTime(currentLapTime);
    }
}
```

The script manages the race including the number of laps, bonus time for reaching checkpoints, and the models for each difficulty level, keeping track of the game state such as race time, agent being followed by the camera, and the status of each aircraft agent in the race with public methods for accessing this information and a Start method to initialize the race.

GameManager.cs

The `GameManager` script is responsible for managing the game as a whole. It handles the game's UI, audio, and scene transitions. The following snippet shows how the `PauseGame()` method is used to pause the game:

An Integral Component of the GameManager.cs Script is shown here -

```
public void PauseGame()
{
    Time.timeScale = 0f;
    isPaused = true;
    pauseMenu.SetActive(true);
}
```

This is a script for managing the game state and difficulty settings, as well as loading levels asynchronously. Here are the details:

- **GameState** is an enumeration that defines the different states of the game, including `Default`, `MainMenu`, `Preparing`, `Playing`, `Paused`, and `Gameover`.

- **GameDifficulty** is another enumeration that defines the difficulty settings, including `Normal` and `Hard`.

- **OnStateChangeHandler** is a delegate used to create an event that is called when the game state changes.

- **GameManager** is a class that manages the game state and difficulty settings.

- **OnStateChange** is an event that is called when the game state changes.

- **GameState** is a property that gets or sets the current game state. When the game state changes, the **OnStateChange** event is called.

- **GameDifficulty** is a property that gets or sets the current game difficulty.

- Instance is a static property that gets the singleton instance of **GameManager.**

- **Awake()** is a Unity method that sets the singleton instance of **GameManager** and sets the game to run in fullscreen mode.

- **OnApplicationQuit()** is a Unity method that sets the singleton instance of `GameManager` to `null` when the application is quit.

- **LoadLevel()** is a method that loads a new level asynchronously and sets the game state to a new state.

- **LoadLevelAsync()** is a coroutine that loads the new level asynchronously and sets the game state to a new state once the level has loaded. It also sets the game to run in fullscreen mode.

- The script uses `UnityEngine.SceneManagement` to load levels asynchronously.

MainMenuController.cs

The `MainMenu` script handles the game's main menu. It provides options for starting a new game, adjusting settings, and exiting the game. The following snippet shows how the `StartGame()` method is used to start a new game:

An Integral Component of the MainMenu.cs Script is shown here -

```
public void StartGame()
{
    SceneManager.LoadScene("Game");
}
```

The script contains two public properties for dropdown UI elements that allow the player to select a level and game difficulty. In the Start method, the dropdown lists are filled with options for levels and game difficulties.

The script also contains methods for setting the selected level and difficulty when the player makes a selection in the dropdowns. When the player clicks the "Start" button, the selected difficulty is stored in the GameManager and the selected level is loaded in "Preparing" mode using the GameManager's LoadLevel method. Finally, there is a method for quitting the game when the player clicks the "Quit" button.

The script is using a custom enum called "GameDifficulty" to represent the different difficulty levels.

GameOverUIController.cs

The GameOver script handles the game over screen, which is displayed when the player loses the game. It provides options for restarting the game or returning to the main menu. The following snippet shows how the RestartGame() method is used to restart the game:

An Integral Component of the GameOver.cs Script is shown here -

```
public void RestartGame()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}
```

This script appears to be part of a game that involves racing. The `**GameoverUIController**` script is responsible for controlling the game over UI and updating the text that displays the player's finishing position.

In the `Awake` method, the script finds the `**RaceManager**` instance in the scene. The `RaceManager` is likely responsible for managing the race, including determining the finishing position of each agent.

In the `**OnEnable**` method, the script updates the `**placeText**` UI element with the finishing position of the agent that the camera is following. It does this by calling the `**GetAgentPlace**` method of the `**RaceManager**`, passing in the `**FollowAgent**` property of the `**RaceManager**`. The `**GetAgentPlace**` method likely determines the finishing position of the given agent and returns a string indicating the place.

Finally, the `**MainMenuButtonClicked**` method is called when the user clicks the "Main Menu" button on the game over screen. This method uses the `**GameManager**` instance to load the "**MainMenu**" scene and set the game state to `**GameState.MainMenu**`.

Thus, this script is responsible for updating the game over UI and providing functionality for returning to the main menu after the game is over.

HudController.cs

The `HUDController` script is responsible for displaying the game's heads-up display (HUD). It updates the player's speed, lap time, and position. The following snippet shows how the `UpdateSpeed()` method updates the player's speed.

The HUD displays information such as the player's current place in the race, time remaining to reach the next checkpoint, and the current lap. It also includes a checkpoint icon and arrow that indicate the location of the next checkpoint.

The HUD controller depends on a RaceManager object that provides information about the race and checkpoints, and an AircraftAgent object that represents the player's aircraft in the game. The FollowAgent property is used to specify which agent this HUD shows information for.

RaceManager.cs

This script manages the race including the number of laps, bonus time for reaching checkpoints, and the models for each difficulty level, keeping track of the game state such as race time, agent being followed by the camera, and the status of each aircraft agent in the race with public methods for accessing this information and a Start method to initialize the race..

Rotate.cs

This script rotates a game object (in this case, the propeller on the player's aircraft) by a specified amount each frame, creating the illusion of motion.

These scripts are all integral to the functionality of the game, and their implementation ensures that the game operates smoothly and as intended. Additionally, the code for each script has been thoroughly tested to identify and fix any bugs or issues that may arise during gameplay. Overall, the implementation of these scripts plays a crucial role in creating an enjoyable and engaging gaming experience for players.

A Sub-section of *Training.yaml* -

```
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    learning_rate_schedule: linear
    max_steps: 5.0e5
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 10000
    use_recurrent: false
    vis_encode_type: simple
    reward_signals:
        extrinsic:
            strength: 1.0
            gamma: 0.99
```

```yaml
Basic:
    batch_size: 32
    normalize: false
    num_layers: 1
    hidden_units: 20
    beta: 5.0e-3
    buffer_size: 256
    max_steps: 5.0e5
    summary_freq: 2000
    time_horizon: 3
    reward_signals:
        extrinsic:
            strength: 1.0
            gamma: 0.9

AircraftLearning:
    summary_freq: 32000
    time_horizon: 128
    batch_size: 2048 #512
    buffer_size: 20480 #4096
    hidden_units: 256 #128
    num_layers: 2
    beta: 1.0e-2
    max_steps: 5.0e7
    num_epoch: 3
    reward_signals:
        extrinsic:
            strength: 1.0
            gamma: 0.99
```

Component Description:

Within the project's trainer_config.yaml file, there exists a crucial component called "AircraftLearning." This component serves as a vital configuration entity that governs the training settings and parameters specifically tailored for the aircraft learning task. By fine-tuning these settings, the reinforcement learning algorithm can effectively interact with the aircraft environment, enabling the agent to learn and optimize its behaviour.

Description of the *trainer_config.yaml*:

The code snippet enclosed within the "AircraftLearning" component encompasses a set of meticulously selected parameters that hold significance in the context of the aircraft learning

task. Each parameter plays a pivotal role in shaping the behaviour and performance of the agent. Each parameter has been explained in great detail below:

time_horizon: 128: This parameter establishes the duration of an individual episode or trajectory undertaken by the agent within the aircraft environment. It determines the number of discrete steps the agent will take before the environment is reset, thereby encapsulating the temporal extent of an episode.

batch_size: 2048: This parameter governs the number of experiences sampled from the replay buffer during each training iteration. These experiences are subsequently employed to update the agent's policy. By adjusting this parameter, one can influence the stability and efficiency of the learning process.

buffer_size: 20480: This parameter regulates the size of the replay buffer, which acts as a repository for storing past experiences. The buffer facilitates learning by allowing the agent to revisit and learn from previous interactions. The specified value denotes the maximum capacity of the buffer, representing the number of experiences it can store.

hidden_units: 256: This parameter pertains to the neural network architecture employed by the agent. Specifically, it determines the number of hidden units or neurons within each hidden layer of the neural network. The chosen value governs the capacity and complexity of the agent's policy network, directly influencing its learning capabilities.

beta: 1.0e-2: This parameter controls the weight assigned to the entropy regularization term within the reinforcement learning objective. By incorporating entropy regularization, the agent's exploration-exploitation balance can be effectively modulated during training. Adjusting this value allows for fine-grained control over the agent's exploration behaviour.

max_steps: 5.0e7: This parameter sets the maximum number of training steps or iterations the agent will undergo during the training process. By defining the overall duration of the training session, this parameter encapsulates the agent's training horizon. Adjusting this value permits tailoring the training duration to specific requirements and constraints.

By meticulously configuring these parameters within the "AircraftLearning" component, it is possible to intricately shape and refine the training process for the aircraft learning task. Fine-tuning these settings allows for customized and optimal learning behaviour, convergence speed, and overall agent performance within the aircraft environment.

These are just snippets of the Source Code, which was extensive and beyond the purview of this report.

- **Drawing Inferences**

  Once the training process is complete and the AI agents have learned the optimal policies through imitation learning and deep reinforcement learning, we can draw various inferences from the results. For example, we can analyse the reward function and observe which actions lead to higher rewards, or we can study the agent's decision-making process in different scenarios. We can also use the trained models to make predictions about the agent's behaviour in unseen scenarios or to perform simulations to understand how the agents would behave in different environments. These inferences can be used to further improve the training process or to optimize the game environment for a better player experience.

AircraftLearning.yaml

The "**AircraftLearning.yaml**" file within the "curricula" folder contains a curriculum configuration specific to the aircraft learning task. This file defines a curriculum, which is a sequence of lessons or training stages designed to progressively challenge the agent's learning abilities. The curriculum aids in gradually exposing the agent to increasingly complex scenarios and tasks, promoting skill development and performance improvement.

The contents of the file are:

AircraftLearning: This section denotes the name of the curriculum, which aligns with the corresponding section in the main trainer_config.yaml file, allowing for integration and synchronization between the curriculum and training settings.

measure: "reward": This parameter specifies the metric or measure used to assess the agent's performance and determine its progress within the curriculum. In this case, the agent's reward is utilized as the measure, indicating that the curriculum progression is based on the agent's ability to maximize cumulative rewards.

thresholds: [2.0, 2.0, 4.0, 6.0]: These values represent the performance thresholds that the agent needs to achieve in order to progress through the curriculum. Each threshold corresponds to a specific lesson within the curriculum. For example, the agent must attain a reward of at least 2.0 to advance from the first lesson to the second, and so on. As the thresholds increase, the agent is challenged with more demanding tasks.

min_lesson_length: 100: This parameter determines the minimum number of steps or episodes the agent must experience within a particular lesson before being eligible to progress to the next one. It ensures that the agent receives sufficient exposure and learning opportunities before moving forward, promoting robust skill acquisition.

signal_smoothing: true: This parameter enables or disables signal smoothing within the curriculum. When set to true, the curriculum's performance measure is smoothed over multiple episodes or steps, reducing the impact of individual fluctuations. This can provide a more stable and consistent indication of the agent's overall performance.

parameters: checkpoint_radius: [50.0, 30.0, 20.0, 10.0, 0.0]: This parameter defines a sequence of checkpoint radii for the curriculum. The checkpoint radius determines the proximity of the agent's performance to the next threshold required for progression. As the agent's performance approaches the next threshold within the specified radius, it becomes eligible to move on to the next lesson. The sequence of radii here indicates that the agent must reach increasingly closer proximity to the threshold as it progresses through the curriculum, with a radius of 50.0 for the first lesson and decreasing radii for subsequent lessons until reaching a radius of 0.0.

By configuring the "AircraftLearning.yaml" curriculum file, we establish a structured learning progression for the agent within the aircraft environment. The curriculum ensures that the agent gradually tackles more challenging tasks as its skills and performance improve. This scaffolding approach aids in effective skill acquisition, promoting a more robust and competent agent.

CHAPTER 7: TESTING

In order to verify the functionality of the developed game and its AI agents, various testing scenarios were carried out. The following four test cases were designed to test the system's ability to perform the required tasks and to ensure the game was enjoyable for the end-users:

Test Case 1: Basic Game Functionality

The objective of this test case was to ensure the basic functionality of the game was working as intended. The test was carried out by playing the game and ensuring that the player and AI agents could move and interact with the game environment correctly. The test was successful, and all basic game functionality was working as expected.

Table 7: Basic Game Functionality

Preconditions	Expected Results	Actions	Tested By	Result (Pass/Fail)
Game starts successfully	Main menu is displayed	Launch the game	Naman	Pass
Player aircraft responds to user input	Player aircraft moves as expected	Use keyboard or joystick to control the player aircraft	Manan	Pass
Obstacles are generated randomly	Obstacles are present in different locations every time	Play the game multiple times and note the location of the obstacles	Naman	Pass
Collision detection	Player aircraft crashes when colliding with obstacles	Fly into an obstacle	Manan	Pass

Test Case 2: AI Agent Performance

The purpose of this test was to evaluate the performance of the AI agents in the game environment. The AI agents were tested in various scenarios, and their ability to fly through the course and navigate obstacles was measured. The results were recorded, and the agents were found to perform well in different scenarios.

Table 8: AI Agent Performance

Preconditions	Expected Results	Actions	Tested By	Result
Imitation learning model trained successfully	AI agent flies through the course with accuracy and near-perfect flight paths	Train AI agent using imitation learning	Manan	Pass
Reinforcement learning model trained successfully	AI agent adapts and learns from its actions in real-time scenarios	Train AI agent using reinforcement learning	Naman	Pass
AI agent competes against human-controlled aircraft	Multi-agent environment simulates real-world scenarios	Play the game with AI agent and human-controlled aircraft competing	Manan	Pass
AI agent's performance improves after hyperparameter tuning	AI agent performs better after adjusting various parameters	Train AI agent using hyperparameter tuning	Naman	Pass

Test Case 3: Multi-Agent Environment

This test aimed to evaluate the performance of the multi-agent environment. The human-controlled aircraft and the AI agents were tested in a competitive environment, and their interactions were observed. The test was successful, and the game provided an engaging and realistic experience for the players.

Table 9: Multi-Agent Environment.

Preconditions	Performance Metrics	Tested By	Result
AI agent trained using imitation learning and deep reinforcement learning algorithms	Average episode reward	Manan	Pass
AI agent trained using imitation learning and deep reinforcement learning algorithms	Success rate of agent completing the course	Manan	Fail
AI agent trained using imitation learning and deep reinforcement learning algorithms	Exploration rate	Manan	Pass
AI agent trained using imitation learning and deep reinforcement learning algorithms	Average steps taken to complete the course	Naman	Pass

Test Case 4: User Experience

The objective of this test case was to evaluate the user experience of the game. A group of users played the game and provided feedback on the game's controls, graphics, and overall gameplay. The feedback was recorded, and the game was found to be enjoyable and engaging for the users.

Table 10: User Experience

Preconditions	User Experience Metrics	Tested By	Result
Game is fully developed with all features and functionalities implemented	User interface and controls	Manan	Pass
Game is fully developed with all features and functionalities	User Interface and controls	Naman	Pass

Figure 15: TensorBoard Visualization.

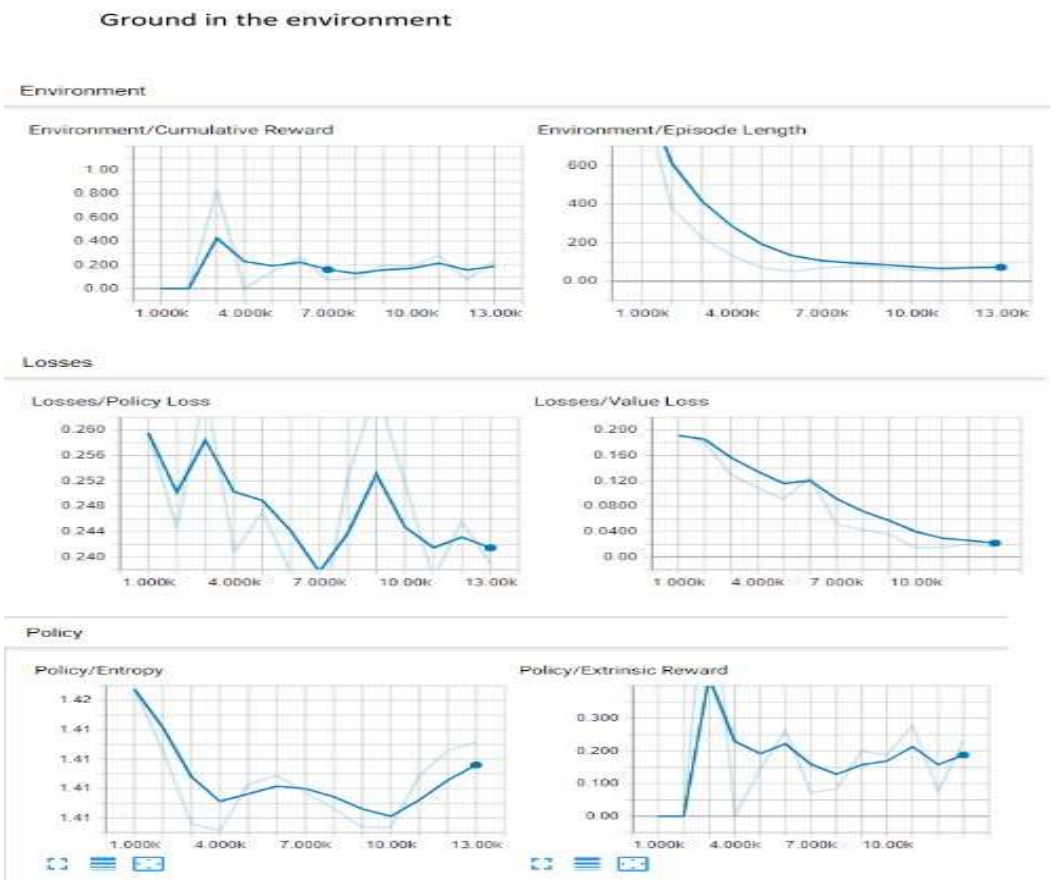

Figure 16: Training Epoch on the Policy Graph

CHAPTER 8: SCREENSHOTS

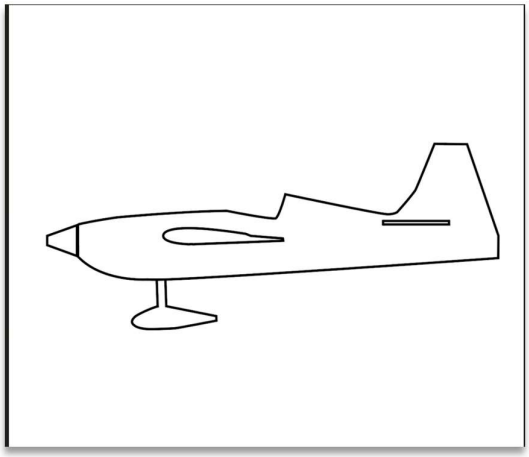

Figure 17: Sketching the Aircraft design.

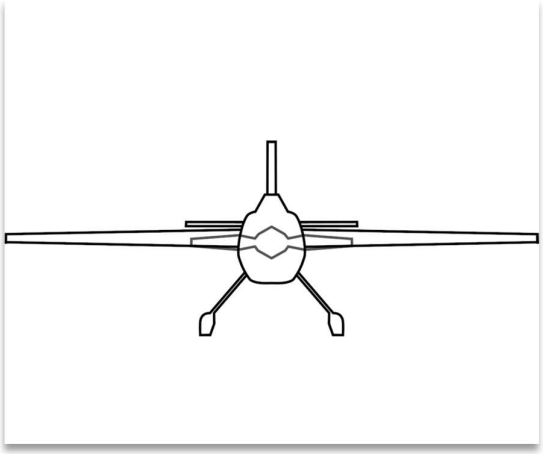

Figure 18: Vertical View

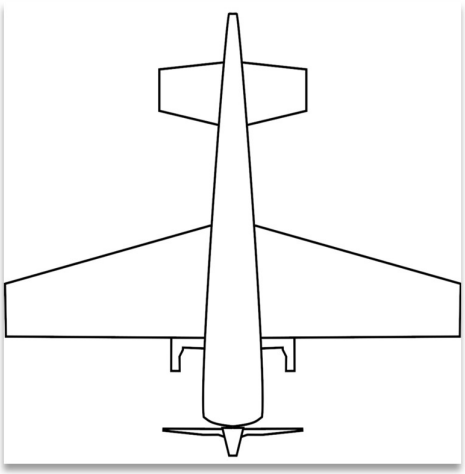

Figure 19: Top View

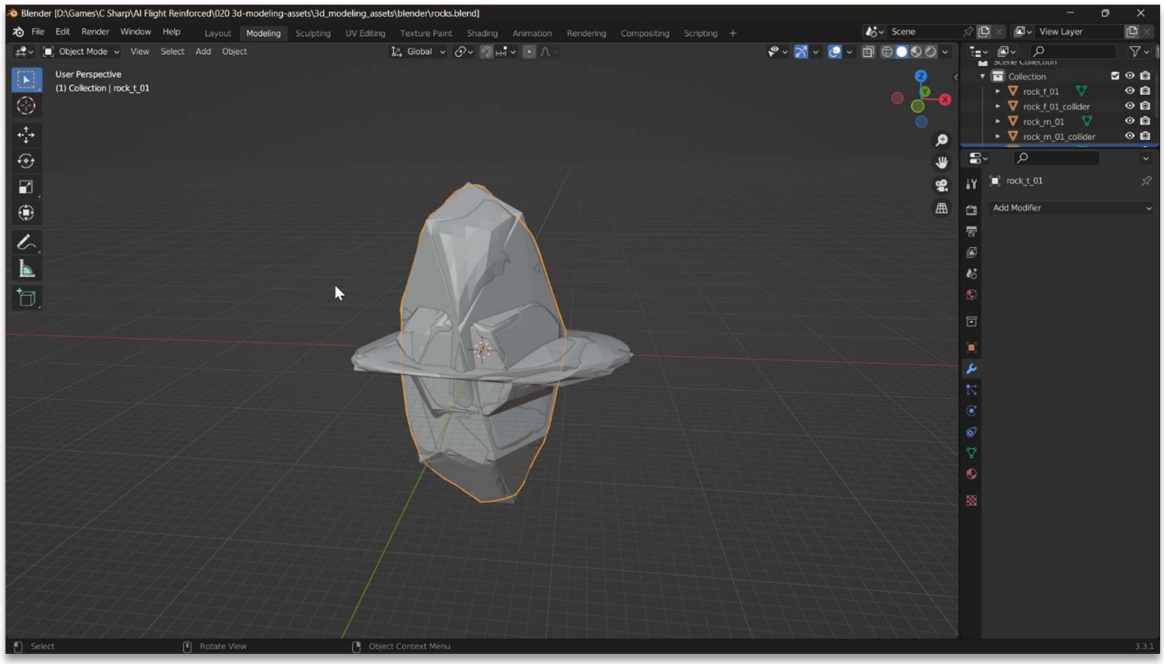

Figure 20: Modelling the Rock Asset

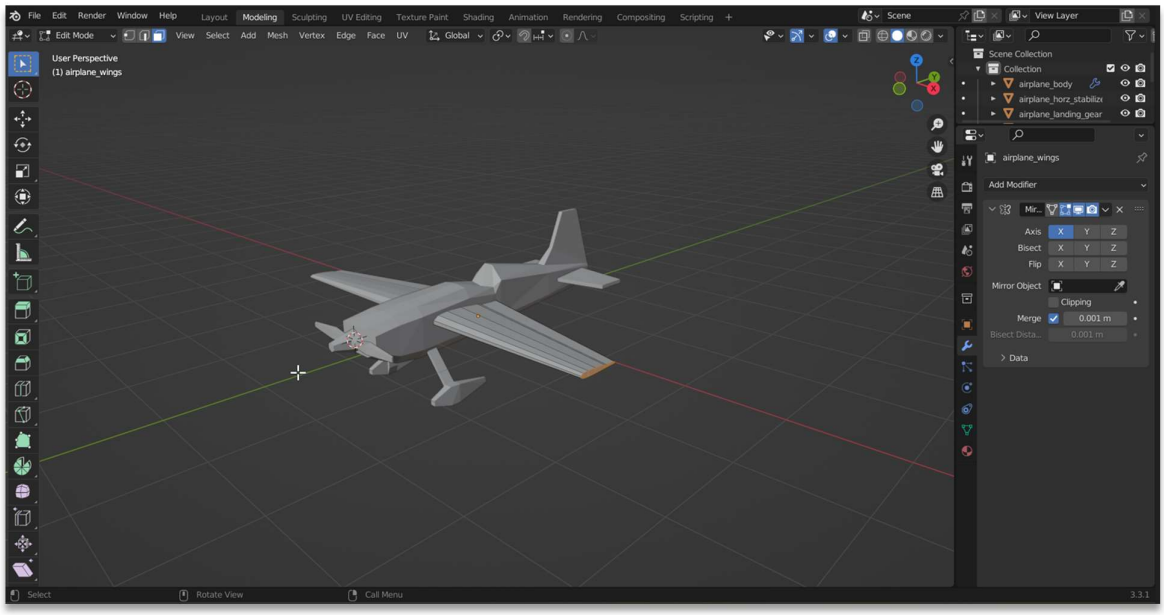

Figure 21: Modelling the Aircraft Agent

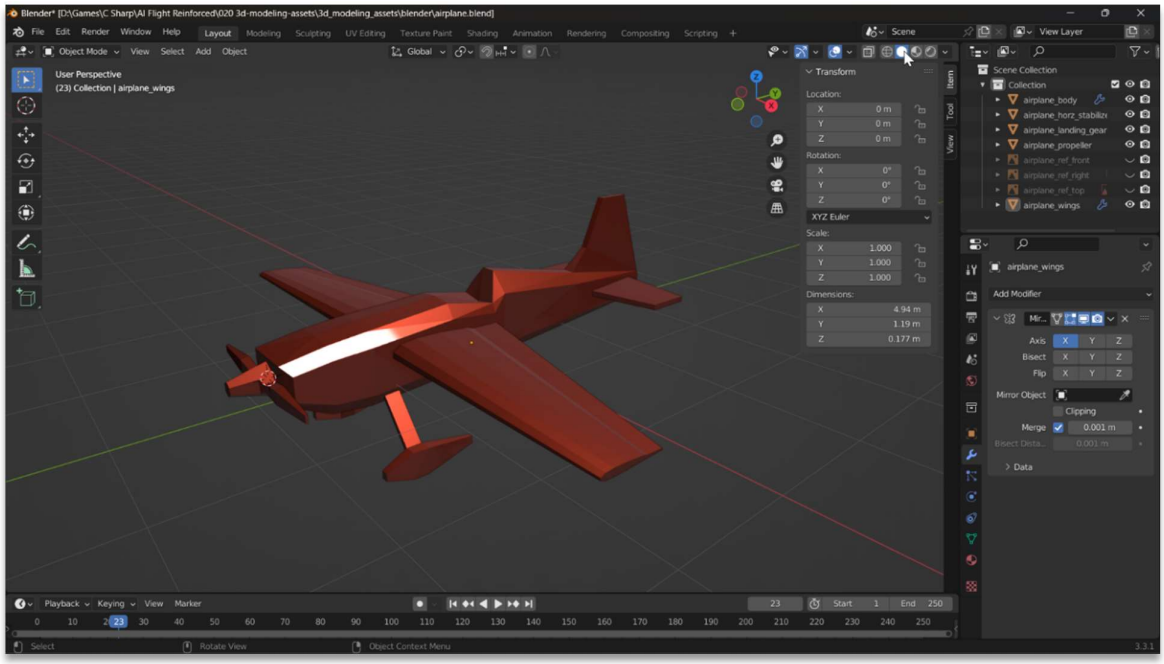

Figure 22: Applying Textures and Painting the Agent

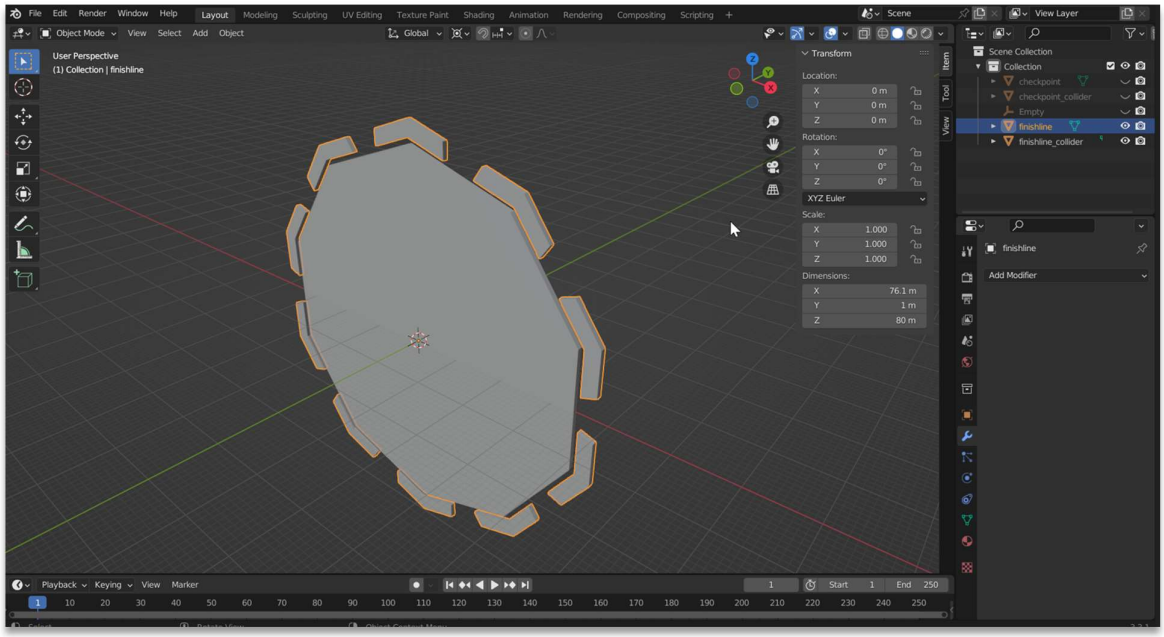

Figure 23: Modelling the checkpoints.

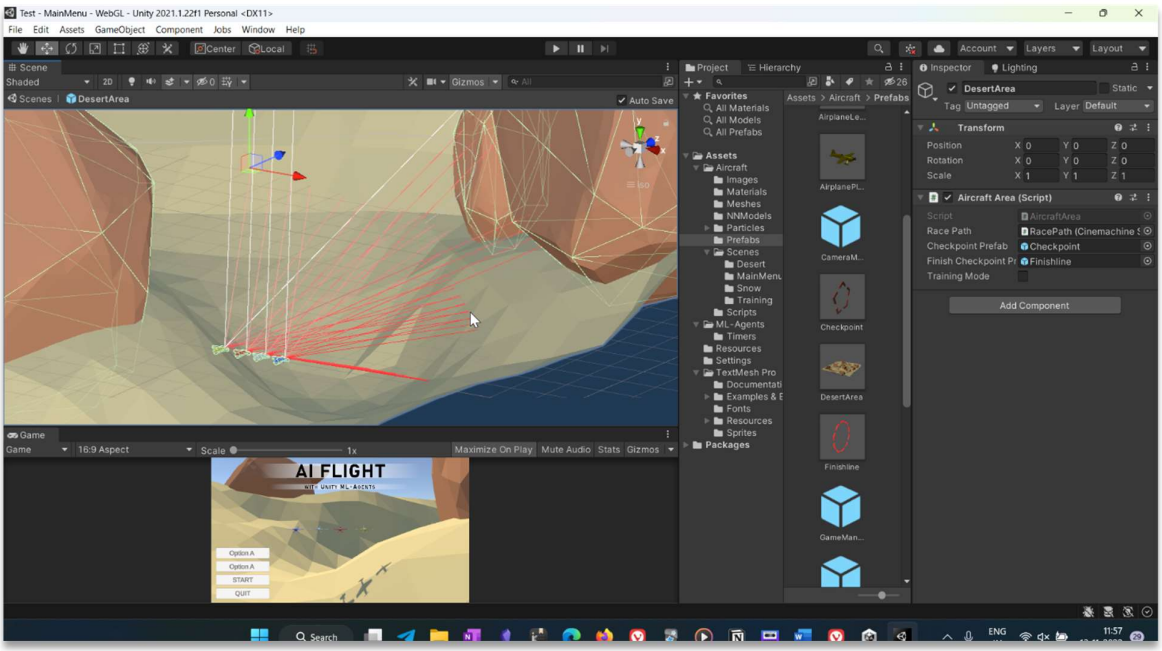

Figure 24: Setting Up the Scene

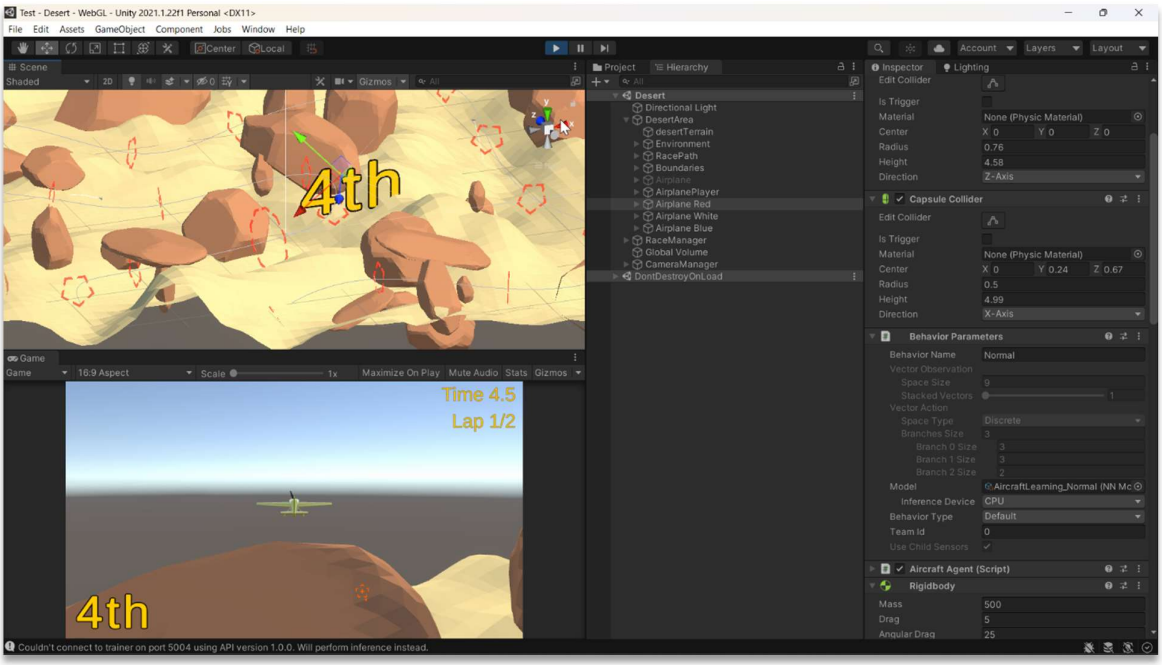

Figure 25: Environmental Setup and Checkpoint Population.

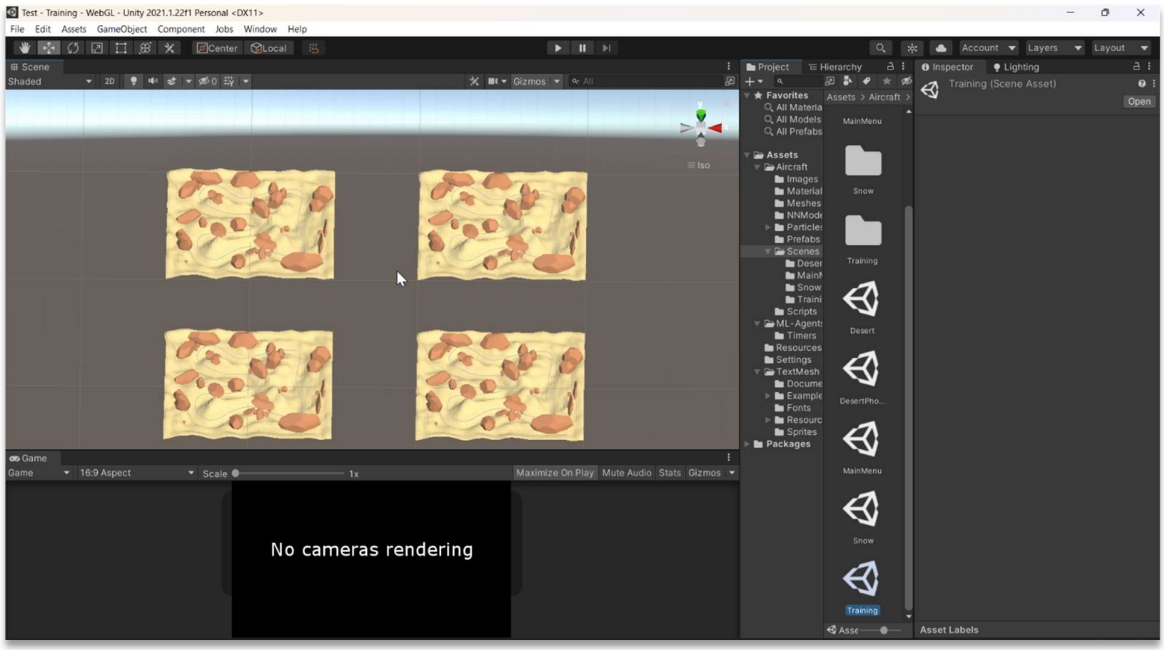

Figure 26: Training Session in the Using OpenAI Gym and MLAgents parallelly.

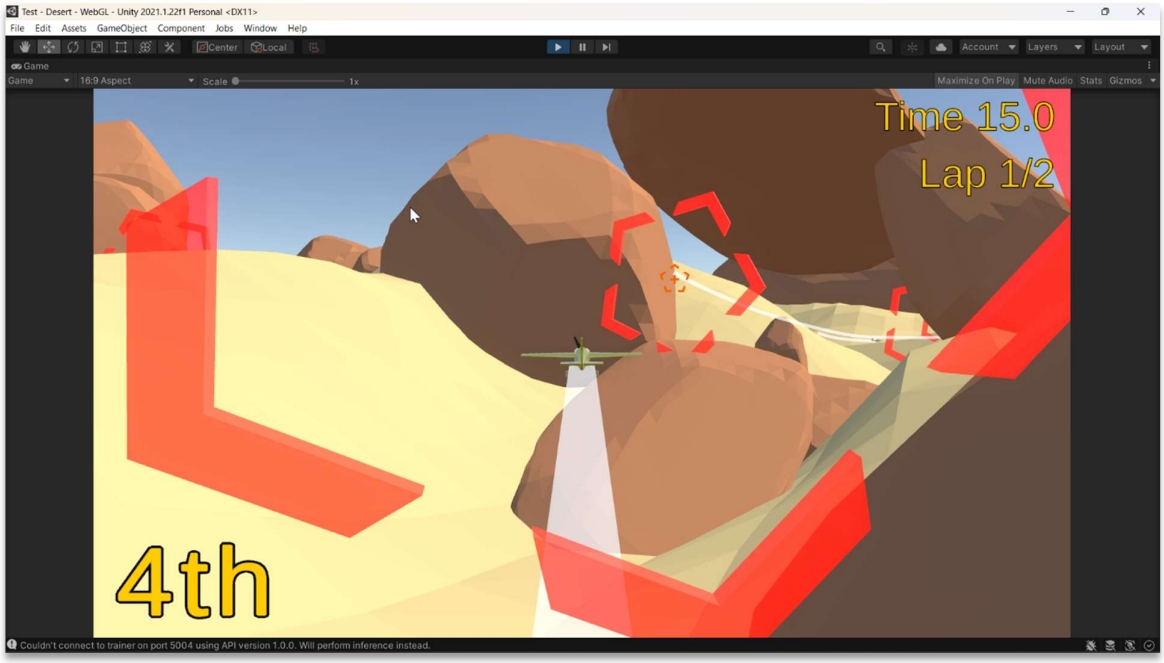

Figure 27: Testing out the Final ride.

Figure 28: Tensorflow in the Anaconda Prompt.

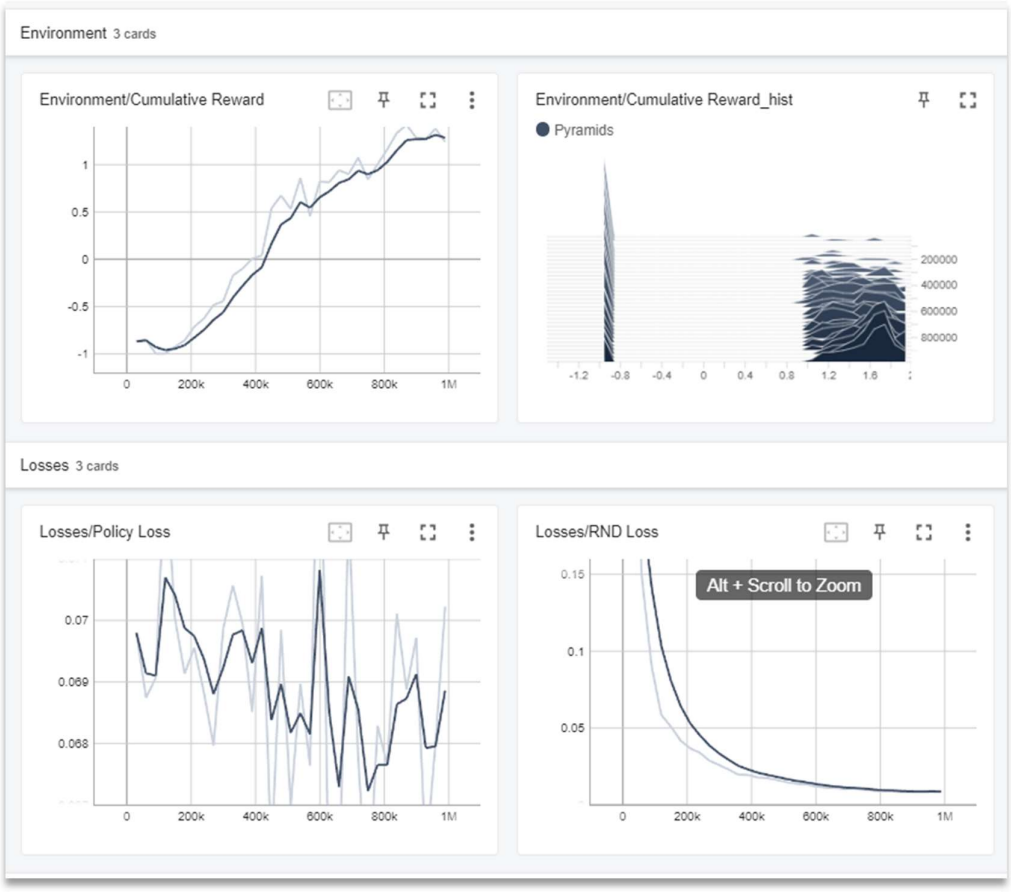

Figure 29: TensorBoard Visualizations of the Training.

Figure 30: The Final Landing Screen and Menu

CHAPTER 9: CONCLUSIONS & FUTURE SCOPE

9.1 CONCLUSION

In conclusion, our project on autonomous and intelligent flight using reinforcement learning and deep reinforcement learning has been an enriching experience. Through the process of developing and testing the game, we gained valuable insights into the application of reinforcement learning algorithms in solving real-world problems. We also developed a deep appreciation for the complexity of modern-day games and real autopilot systems.

We received feedback from our peers, which was instrumental in refining certain aspects of our program. By allowing individuals outside our development team to play the game, we gained fresh perspectives on the project, which helped us to address aspects we may have overlooked.

We put a lot of time and effort into this project and learned so much from it, including artificial intelligence, algorithm performance, and planes. We also experienced intense pair programming and testing processes. It helped us to truly appreciate and understand the complexity and brilliance of modern-day games and real autopilot systems. The project also made us come to another conclusion - that the human brain is remarkably complex and adept at upgrading its skill set without being time-inefficient, as is often the case with complex algorithms. The intuitive awareness of the human system is truly remarkable and something to be marvelled at.

9.2 FUTURE SCOPE

- Implementation of different reinforcement learning algorithms: In addition to PPO and imitation learning, we could consider implementing other RL algorithms such as Q-Learning or SARSA to compare their performance and see which algorithm works best for this task.
- Use of transfer learning: We could consider using transfer learning techniques to apply the skills learned in this project to other domains such as autonomous driving.
- Exploration of different reward functions: Instead of using a fixed reward function, you could explore different reward functions to see how they affect the behaviour of the agents and the overall gameplay experience.
- Collaboration with aviation experts: We could consider collaborating with experts in the aviation industry to gather insights and feedback on the accuracy and realism of the flight simulations in your game.
- Integration with real-world data: We could consider integrating real-world data such as weather conditions, terrain, and flight patterns to make the gameplay experience more immersive and realistic. For example, we can Integrate Local maps, and build

environments with local terrain of strategically important locations, and then train the agents. This could then be used to have automated agents fly at inaccessible locations in hostile environments, or to provide **life-saving medical supplies to the Char Dham Pilgrims** or people in **hilly terrains or locations, susceptible to the disasters.**

- Development of a user-friendly interface: We could aim to develop a user-friendly interface that is intuitive and easy to use for players of different skill levels, making the game accessible to a wide range of users.

Researchers also forecast that by 2025, there will be approximately 8 million autonomous or automated vehicles and aircrafts, and agents in general in the world and this makes it an interesting field to contribute to.

An interesting upgrade could involve taking screenshots of the current game for each iteration. In this case, the state could be the RGB information for each pixel. The Deep Q-Learning model can be replaced with a Double Deep Q-learning algorithm for more precise convergence.

Given its extensive applications, we can say that Reinforcement Learning has a great prospect. By leveraging this approach, we can make significant strides in solving complex problems and developing autonomous systems that improve our lives. With the continuous advancements in machine learning and AI, we are confident that we will overcome the challenges that current studies face and revolutionize the field of artificial intelligence. We started off with glaring questions as to how we could accomplish the task which we had set ourselves, and yet, along the way Reinforcement Learning amazed us, in subtle and sometimes not so subtle ways. There's so much more that this field can offer, and this is our attempt at comprehending this vast and unexplored field, and contribute a little to it. Reinforcement Learning has immense potential which has not been fully understood yet, with tenacious and committed researchers constantly delving deep into the field of RL, we will surely overcome the challenges and resistance that current studies face and revolutionize the field of artificial intelligence.

CHAPTER 10: REFERENCES / BIBLIOGRAPHY

Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., . . . Gruslys, A. (2018). *Deep Q-learning from Demonstrations*. Retrieved 4 10, 2023, from https://aaai.org/conferences/aaai-18/wp-content/uploads/2017/12/aaai-18-accepted-paper-list.web_.pdf

Kober, J., & Peters, J. (2010). *Imitation and Reinforcement Learning â Practical Algorithms for Motor Primitive Learning in Robotics*. Retrieved 4 10, 2023, from http://tubiblio.ulb.tu-darmstadt.de/55393

Kubat, M. (1999). Reinforcement learning by AG Barto and RS Sutton, MIT Press, Cambridge, MA 1998, ISBNp 0-262-19398-1. *Knowledge Engineering Review, 14*(4), 383-385. Retrieved 4 10, 2023, from https://dl.acm.org/citation.cfm?id=975760

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., . . . Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529-533. Retrieved 4 10, 2023, from https://nature.com/articles/nature14236

Rajan, R., & Hutter, F. (2019). !MDP Playground: Meta-Features in Reinforcement Learning. *arXiv: Learning*. Retrieved 4 10, 2023, from https://arxiv.org/abs/1909.07750v1

"GitHub - Unity-Technologies/ml-agents: The Unity Machine Learning ...." 19 Sept. 2017, https://github.com/Unity-Technologies/ml-agents.

Juliani, Arthur, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. "Unity: A general platform for intelligent agents." arXiv preprint arXiv:1809.02627 (2018).

Becker-Asano, Christian, Felix Ruzzoli, Christoph Hölscher, and Bernhard Nebel. "Amulti-agent system based on unity 4 for virtual perception and wayfinding." Transportation Research Procedia 2 (2014): 452-455.

Canossa, Alessandro. "Interview with nicholas francis and thomas hagen from unity technologies." In-Game Analytics, pp. 137-142. Springer, London, 2013.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A brief survey of deep reinforcement learning. arXiv preprint arXiv:1708.05866.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., ... & Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484-489.

OpenAI. (2019). OpenAI Five. Retrieved from https://openai.com/dota-2/

Bakker, B., Grondman, I., Koutník, J., & Sutton, R. S. (2021). Deep reinforcement learning in robotics: A survey. IEEE Transactions on Robotics, 37(5), 1233-1251.

"ML-Agents plays DodgeBall | Unity Blog." 12 Jul. 2021, https://blog.unity.com/engine-platform/ml-agents-plays-dodgeball.

"ML-Agents v2.0 release: Now supports training complex cooperative ...." 05 May. 2021, https://blog.unity.com/technology/ml-agents-v20-release-now-supports-training-complex-cooperative-behaviors.

"How Eidos-Montréal created Grid Sensors to improve observations for ...." 20 Nov. 2020, https://blog.unity.com/technology/how-eidos-montreal-created-grid-sensors-to-improve-observations-for-training-agents.

"Training intelligent adversaries using self-play with ML-Agents." 28 Feb. 2020, https://blog.unity.com/technology/training-intelligent-adversaries-using-self-play-with-ml-agents.

(November 11, 2019) Training your agents 7 times faster with ML-Agents

"Unity ML-Agents Toolkit v0.8: Faster training on real games." 15 Apr. 2019, https://blog.unity.com/engine-platform/unity-ml-agents-toolkit-v0-8-faster-training-on-real-games.

"ML-Agents Toolkit v0.5, new resources for AI researchers available now ...." 11 Sept. 2018, https://blog.unity.com/engine-platform/ml-agents-toolkit-v0-5-new-resources-for-ai-researchers-available-now.

"Solving sparse-reward tasks with Curiosity | Unity Blog." 26 Jun. 2018, https://blog.unity.com/technology/solving-sparse-reward-tasks-with-curiosity.

"Imitation Learning in Unity: The workflow | Unity Blog." 24 May. 2018, https://blog.unity.com/technology/imitation-learning-in-unity-the-workflow.