CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Obstacle Detection and Avoidance Algorithm in Drones: Airsim Simulation with HIL

A graduate project submitted in partial fulfillment of the requirements
For the degree of Master of Science in Computer Engineering

By

Vikas Vasudevan

May 2021

The graduate project of Vikas Vasudevan is approved:

_____	_____
Prof. Sevada Isayan	Date
_____	_____
Dr. Shahnam Mirzaei	Date
_____	_____
Dr. Xiaojun Geng, Chair	Date

California State University, Northridge

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Obstacle Detection and Avoidance Algorithm in Drones: Airsim Simulation with HIL

By

Vikas Vasudevan

Master of Science in Computer Engineering

This project is realization of Obstacle Detection and Avoidance algorithm in Drones. The proposed project is a simulation in Airsim with hardware-in-loop (HIL). This report begins with setting up the necessary environments for simulation and review of the algorithm. In this project, the simulation is carried out in Unreal Engine with a preferred simulation environment. The hardware used is a PX4 based Pixhawk 2.4.8. It is configured with a ground control center application called QGroundControl. Unreal Engine was chosen over other game development engines since Airsim libraries are better built for the former and provides a good platform to various testing scenarios. The Drone used in the simulation is a generic quadcopter enabled with three sensors. The three sensors are lidar, stereo depth sensor and a camera sensor. The environment data is logged through these sensors into a csv file which is then used in the algorithm.

The algorithm chosen for this purpose is deep Q-network or DQN, a reinforcement learning algorithm. The algorithm is scripted in Python to use cameras to capture the data from the environment in the simulation. The captured data is collected from the moving drone and the data is trained. This algorithm trains the drone to detect and avoid obstacles in its planned path.

SECTION 1: INTRODUCTION

Obstacle detection and avoidance algorithms have been the backbone supporting autonomous vehicle traversal with minimum to no human interaction. Rapid technological growth has enabled developers to work hands on and implement these algorithms to make vehicular movement “smart”. The algorithm implemented is DQN (Double deep Q Network), it is a reinforcement learning algorithm that teaches the vehicle, in this case a generic quadcopter to avoid obstacles in its planned path of traversal. The algorithm has been implemented through Airsim in this project.

Airsim is an open source community driven project meant to simulate drones and study the behavior of algorithms in drone traversal. Airsim has been developed to run with a game development engine like Unreal and Unity. The goal of Airsim is to allows developers to program and run supporting APIs and simulate the behavior of the drone in a realistic environment with dynamic/static environment models. Airsim APIs connect to the Unreal engine environment through Remote Procedure Call (RPC) [9].

Unreal engine v 4.25+ is supported to run all latest Airsim APIs. The environment for drone simulation can be designed and developed as required or pre-designed environments can be imported and used in the project [2]. The Unreal environment calls the Airsim APIs and the python script executes to implement the algorithm for simulation.

The PX4 software stack is used on a Pixhawk 2.4.8 for Hardware-in-the-loop simulation. The pixhawk is a flight controller specifically designed to support the flight software stack and can be used to run scripts and simulate a drone before it can be deployed onto a real drone. The flight controller comes with a package of sensors that needs to be manually calibrated by the developer.

The second section of this report provides an in-depth review of how Airsim has been used to simulate the drone. This section shows all the methodologies followed in Airsim and all the features available to the developer to implement on to a drone.

The third section gives an overview of Unreal engine and all the steps followed to setup the simulation environment.

Section four gives an in-depth explanation and analysis of the flight controller Pixhawk 2.4.8 and the PX4 software stack used in this project. This section also provides a feature list and specifications of the flight controller.

The focus of the next section is the process involving setting up the hardware to run the script and test the algorithm for simulation. It also provides information about the MAVLink communication protocol and QGroundControl base station.

The following section shows a detailed view of setting up the simulation for testing with all the required configurations and APIs used.

The final section gives a detailed explanation of the DQN algorithm and its implementation in the project to achieve the objective of obstacle detection and avoidance using reinforcement learning.

SECTION 2: AIRSIM

2.1 Overview

Airsim is an Unreal Engine based simulator for drones and other terrestrial vehicles. It is a community developed, open-source, cross-platform simulator intended to simulate a vehicle (typically a drone or a car). It supports both software-in-loop (SIL) and hardware-in-loop (HIL) simulation [9]. SIL is supported with many popular flight controllers like the ArduPilot and HIL is supported with PX4 based flight controllers like the Pixhawk series. The simulations in Airsim are physically and visually realistic. Airsim has been developed as plugin that can be dropped into any user specific environment in Unreal Engine [11].

The simulator is under continuous development and serves as a platform of research to experiment with algorithms on deep learning, computer vision and reinforcement learning for autonomous vehicles.

Airsim simulator has APIs which enables developers to retrieve data and control the vehicle as per requirement.

In this project, Airsim is built on windows operating system since many relevant APIs and libraries have been developed for it. The APIs are configured and programmed using python on Visual Studio 2019.

The Airsim simulator is configured and set-up the following way:

- The required binary files are downloaded in the local repository
- Clone the Airsim repo in the VS 2019 developer command prompt
- Build the repo to generate plugins required for Airsim to work in an Unreal environment.

2.2 Using Airsim

Airsim simulator allows programmatic control, gathering training data, manipulating weather conditions and has a computer vision mode [9].

2.2.1 Programmatic Control

The simulator exposes APIs and allows developers to interact with the vehicle in the simulation programmatically. These APIs can be used to retrieve images, get vehicle state and control the vehicle [11].

The APIs have been exposed by remote procedure call (RPC), and can be accessed via programming languages, including C++, Python, C# and Java.

The APIs can be deployed directly on a computer with the simulator or vehicle. APIs are available as a cross-platform library. This allows developers to test code in the simulator, and later execute it in real vehicles.

2.2.2 Acquiring training data

Airsim provides two ways to access and gather the data required for deep learning. The easier way is to hit the record button at the bottom of the screen to take screenshots of each frame in the simulator [11].

The second approach is a better way to generate the training data, in this way users can access APIs and allows full control of how the data should be handled.

2.2.3 Computer Vision Mode

This mode allows developers and users to use APIs to place cameras in required locations and pick up depth, disparity, surface normal from an image

2.2.4 Manipulating Weather Conditions

For working with dynamic weather conditions, Airsim provides access to two ways of accessing APIs. The first is a simple way to control weather conditions and attributes through the weather control menu which is accessed by pressing the F10 key.

The second approach is by using APIs. By default, all the weather effects and conditions will be disabled. To enable weather effects, first call:

```
simEnableWeather(True)
```

Airsim provides access to multiple combinations of weather effects which the developer can manipulate to simulate the required weather effect [9].
The weather API parameters are as follows:

```
class WeatherParameter:
    Rain = 0
    Roadwetness = 1
    Snow = 2
    RoadSnow = 3
    MapleLeaf = 4
    RoadLeaf = 5
    Dust = 6
    Fog = 7
```

For some of these effects like Roadwetness, RoadSnow and RoadLeaf , users need to add the materials to the simulation profile.

In general, Airsim is a simulator which allows developers to deploy APIs to simulate real-life conditions

SECTION 3: UNREAL ENVIRONMENT

3.1 Overview

Unreal Engine is a widely regarded game engine developed in the late 90s to develop games and simulation environments.

To set-up the Unreal Engine in this project, Unreal Editor (UE) 4.26.1 was installed and configured with a blank project.

Figure 1: Relationship diagram

In the above shown figure 1, the two links links show the data flow between the game engine, Airsim and VS2019.

- Airsim receives and exposes APIs to manipulate drone movement in unreal engine.
- The exposed APIs are programmed as per requirement in VS2019
- The programmed script is then executed on VS2019 and the APIs linked to the Unreal engine cause it to function as per the script in links

3.2 Building the setup Environment

Visual Studio 2019 developer command prompt is used to clone the airsim repository from github to the chosen local directory

git clone https://github.com/Microsoft/Airsim.git

To create the necessary Plugins to connect Airsim with the Unreal Environment, run build.cmd.

3.3 Testing the Unreal Environment

A test mode environment called Blocks is used to test the Airsim plugins with Unreal engine before shifting to any preferred environment [13]. The main reason the for testing the environment is to check if all static and dynamic models in the environment are glitch free. The template shown in figure 2 is a test flying template generated by unreal engine in the blocks environment. Moving the camera manually around this template allows the user to test the physics of the environment before simulation.

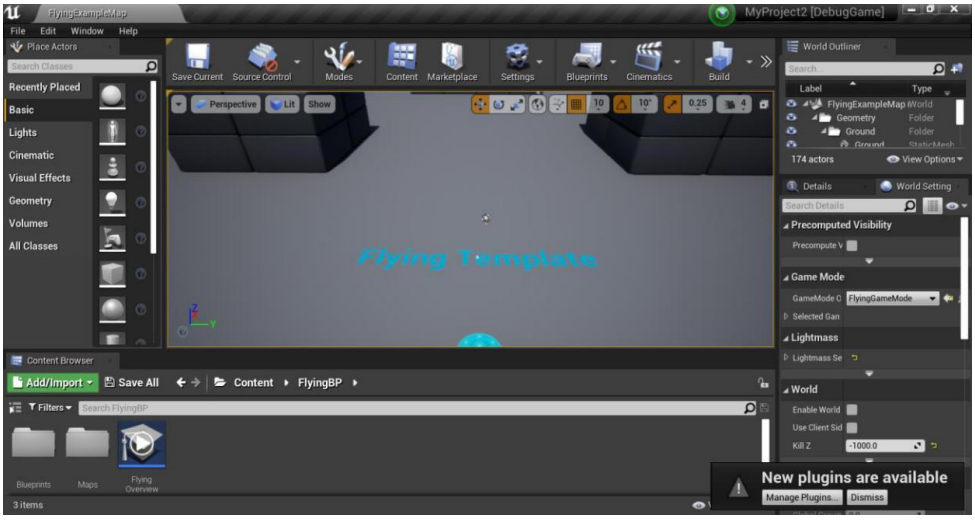

Figure 2. Unreal Engine

SECTION 4: THE HARDWARE – PIXHAWK 2.4.8

4.1 Overview

Pixhawk is a very popular general-purpose flight controller and its architecture is based on the FMUv2 open hardware design. It is configured and designed to run the PX4 software stack. The controller can be configured for both autopilot and manual flight with a RC transmitter.

4.2 PX4 Software Stack

It is an open source software stack for flight controllers and supports a wide range of boards and sensors. It has a sophisticated built-in capability for achieving higher level of tasks like mission planning [14].

It can be used in a variety of situations, from consumer drones to industrial applications. PX4 is the most widely used site for drone testing. This software stack has also been used on warships and underwater vehicles with great success.

PX4 provides a plethora of development tools for drone technology developers to create solutions for drone applications. It also provides with a high standard to give maximum hardware support and a software stack. This gives rise to an ecosystem to build and sustain hardware and software in a highly scalable way.

The features of PX4:

- Modular Architecture
    - PX4 has a highly modular architecture. It is extensible both in terms of software and hardware. A port-based architecture is utilized in the stack that allows developers to add multiple components and this does not cause any loss of performance or robustness.

- Open Source
    - The software stack is an open-source community developed project to encourage drone developers to work and build a highly sustainable software package which is highly modular to flight controllers.

- Configurability
    - A plethora of APIs and SDKs are offered by PX4 to developers working on integrations. Because PX4 is highly modular, all the inner modules can be swapped with a different module without affecting any core APIs. Also, feature deployment into the stack can be done easily and reconfigured to the developer's liking.
- Autonomy Stack
    - For autonomous capabilities, the software stack has been developed to work with an integrated computer vision system. Also, the offered framework has a lowered barrier of entry for software developers working on obstacle detection and avoidance algorithms.
    - Pixhawk controllers from 3DR Pixhawk v2 to Pixhawk 4 have been tested by developers with Airsim.

For implementing obstacle detection and avoidance algorithms in drones, Pixhawk PX4 2.4.8 was chosen.

4.3 Pixhawk PX4 2.4.8

The Pixhawk 2.4.8 is a flight controller board configured with the PX4 software stack and can be integrated into simulations and finally into real-life drones [14].

4.3.1 Specifications

The flight controller integrates with PX4FMU + PX4IO. It has a 32 bit processor and integrated sensors [14].

- Processor:
    - 32 bit 2M flash memory STM32F427 cortex M4 with a hardware floating point processing unit

- Main frequency of 256K, 168MHz RAM.
- Sensor
    - L3GD20 3 axis digital 16 bit gyroscope
    - LSM303D 3 axis 14 bit magnetometer/accelerometer
    - MPU6000 6 axis magnetometer/accelerometer
    - MS5611 high precision barometer

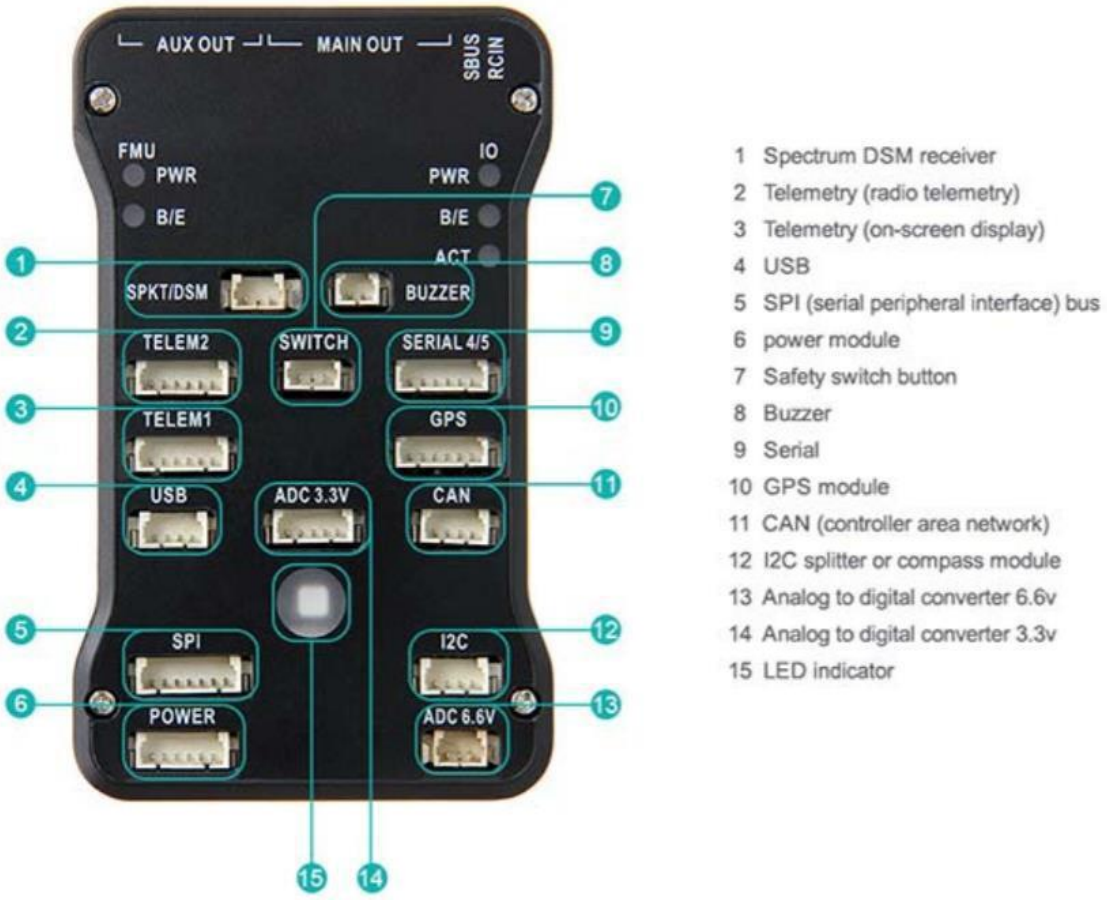

Figure 3: Pixhawk 2.4.8 Flight Controller [www.pixhawk.org]

The hardware shown in figure 3 is a Pixhawk 2.4.8 controller with PX4 software stack. The controller was considered for this work since Airsim supports a multitude of libraries and APIs for this particular flight controller.

- Interface
    - 5xUART, 1xcompatible high voltage, 2xhardware flow control
    - 2xCAN
    - Spektrum DSM/DSM2/DSM-X satellite receiver compatible output
    - SBUS compatible input and output
    - PPM signal input
    - RSSI input
    - I2C
    - SPI
    - 3.3 and 6.6V ADC input
    - External USB Micro interface

- Features
    - Can run RTOS NuttX real time OS
    - Integrated backup power supply and the main controller can be safely switched to backup control
    - Redundant power input and fault transfer function
    - 14x PWM/actuator output
    - Bus Interfaces
    - Can be used in autonomous or manual model.

SECTION 5: SETTING UP PX4 HARDWARE-IN-LOOP

In order to setup the Pixhawk in HIL, a ground control center (base station) application called QGroundControl has to be installed on a computer. The flight controller is the connected to the computer via USB.

The Pixhawk PX4 software stack is built to communicate with the base station using MAVLink protocols. These protocols allow the base station to talk to the drone.

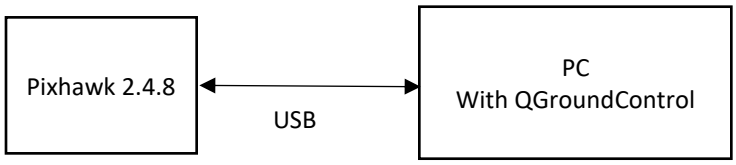

Figure 4: Hardware in the Loop

In figure 4, the Pixhawk flight controller is connected to the PC. The PC has all the required files and dependencies installed. The communication between the two happens through MAVLink communication protocol.

5.1 MAVLink

It is a highly lightweight messaging protocol which is mainly used to communicate with drones. The protocol follows a super modern approach of hybrid publish-subscribe and p2p design pattern. This approach allows data streams to be sent/published as topics while protocols such as mission protocol and parameter protocol are p2p with retransmission [12]

5.1.1 Features of MAVLink

- MAVlink 1 has only 8 bytes overhead per packet and this includes a start detection and packet drop detection. MAVLink 2 has 14 bytes overhead. The protocol has been designed to suit applications with little operable communication bandwidth.

- MAVLink is highly reliable since it provides methods for detection of packet loss, packet corruption and also provides packet authentication.
- Supports multiple programming languages and runs on numerous microcontrollers/os.
- Allows a maximum of 255 concurrently working systems on a network.
- Enables a communication with both onboard and base station systems.

5.2 QGroundControl

QGroundControl is a highly intuitive and powerful ground control station for MAVLink protocol to control the drone. It provides with a full flight control and mission planning for drones with MAVLink enabled flight controllers such as the Pixhawk 2.4.8. It is an open-source project and its main purpose of development is ease of use for users and developers.

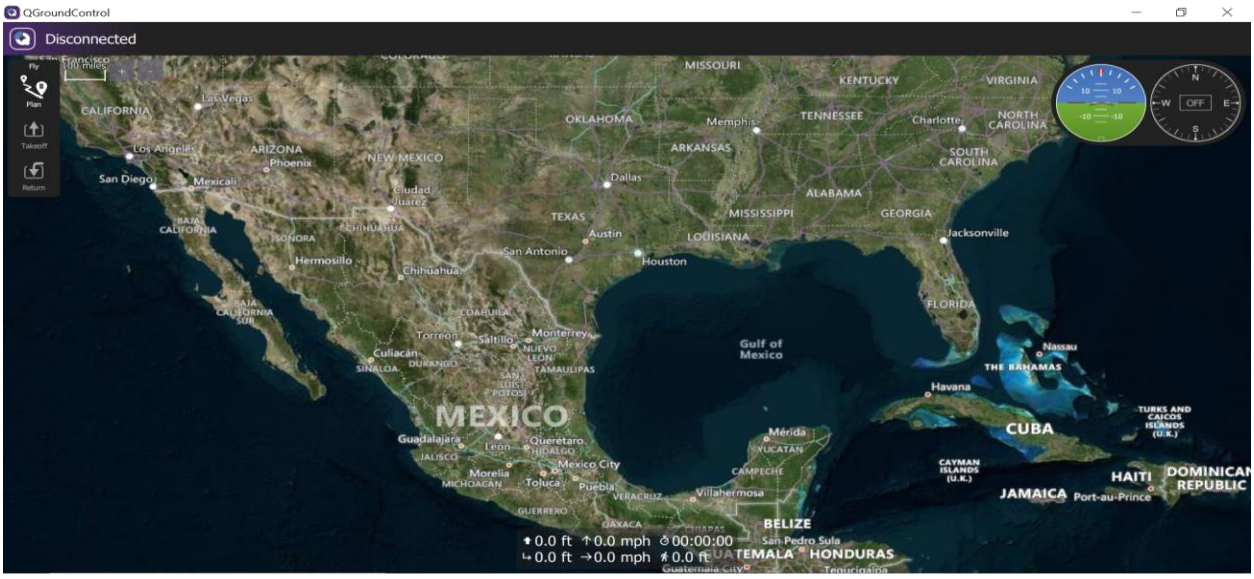

Figure 5: QGroundControl Disconnected mode

Initially when the QGroundControl application is launched it will be in disconnected(unarmed) mode. Figure 5 shows the controller is off-mode, this is shown in the UI of QGroundControl on the top left.

Next, the flight controller is connected to the computer running QGroundControl via USB. The base station software detects the connected hardware.

The following figure 6 shows the controller in connected mode but the controller is not yet armed for take-off.

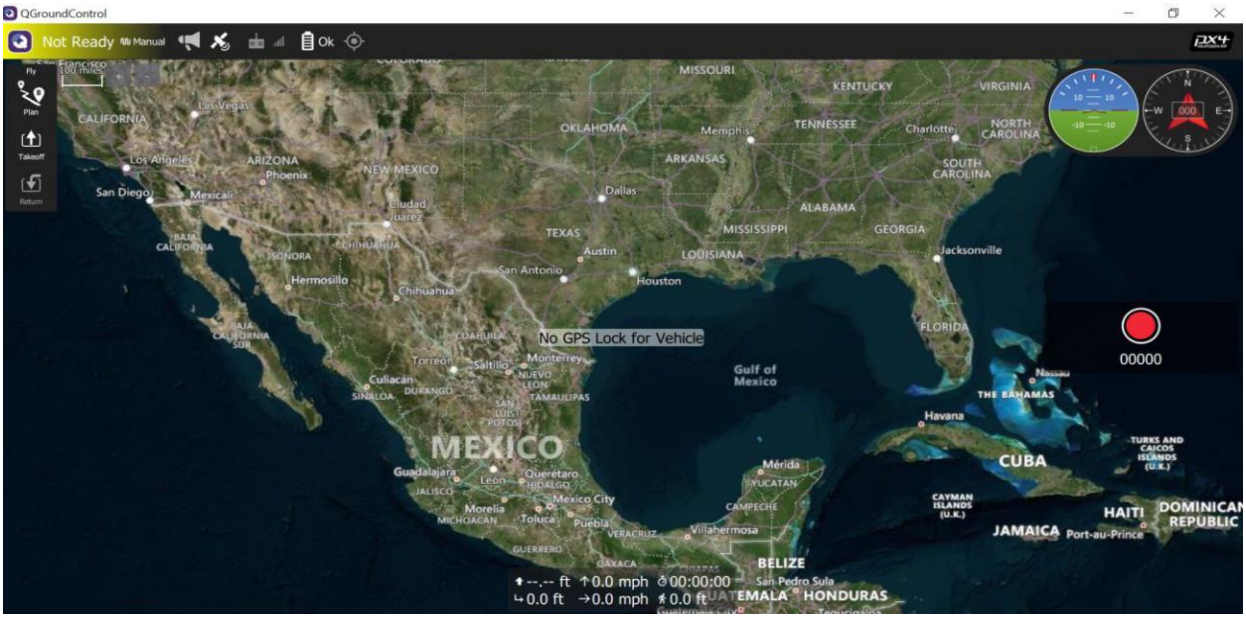

Figure 6: QGroundControl Connected mode

The PX4 flight stack has to be downloaded and the latest firmware is flashed onto the flight controller.

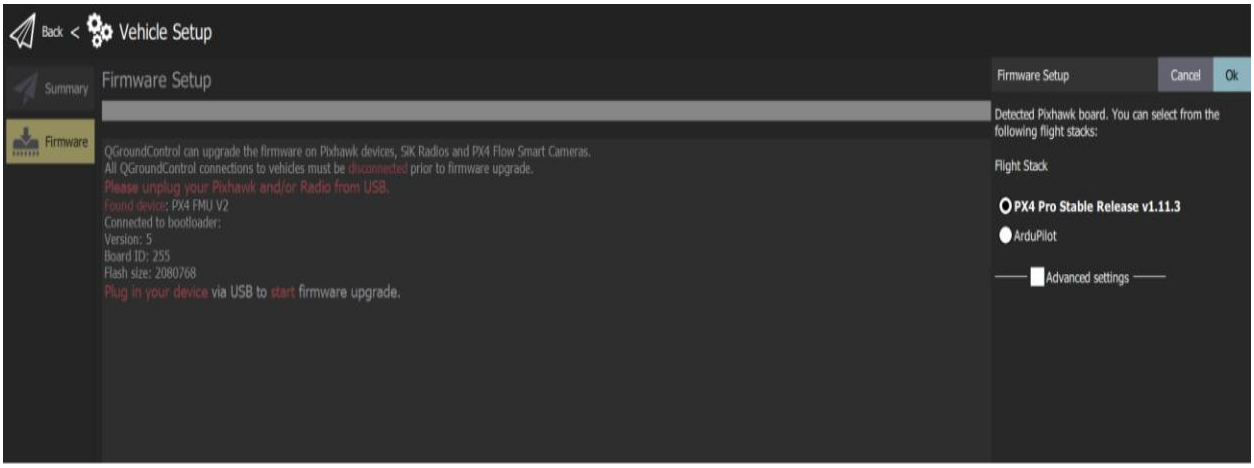

Figure 7: Pixhawk Firmware

The vehicle setup shown in figure 7 enables users to update and install the firmware of the PX4 software stack on to the flight controller.

The flight controller configurations are done as shown in figure 8. The drone simulation is done for a generic quadcopter. For the airframe configuration, generic quadcopter is chosen.

Figure 8: Selecting Airframe for drone

Finally, the sensors on board are calibrated. The compass is calibrated with respect to the true north [12]. Figure 9 demonstrates all the orientations of calibration required to complete the setup of hardware.

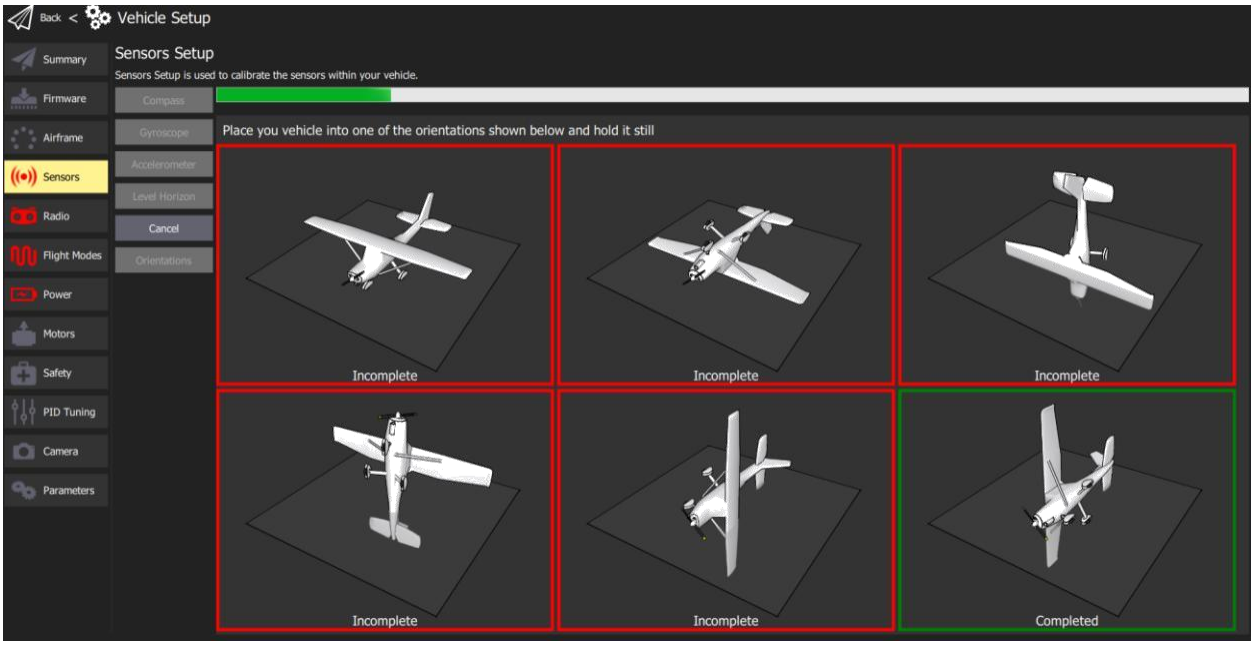

Figure 9: Manual calibration of sensors.

If the hardware is not calibrated in the intended way, an error log file is displayed to the user as shown in figure 10.

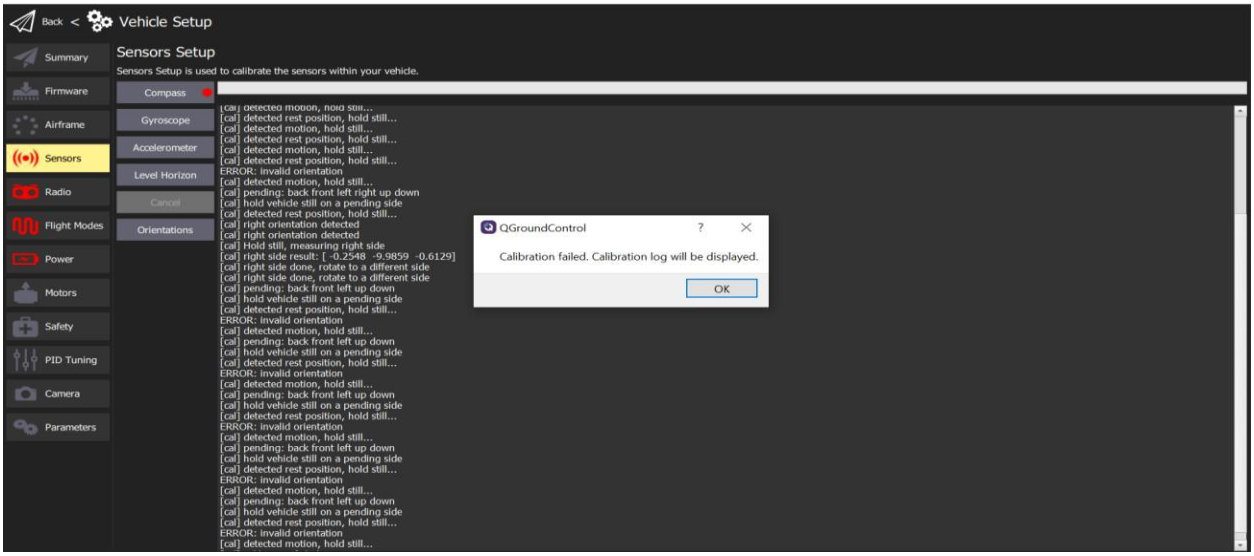

Figure 10: Faulty sensor calibration

SECTION 6: SIMULATION TEST ENVIRONMENT

The Blocks environment is chosen as the test environment since it does not require a powerful GPU. The environment is set up with a settings.json file, which allows the user to deploy a drone and make custom settings. The json file is parsed and the environment is set up [11].

The test environment is static and the interactions with the environment are considered to be collisions. The blocks environment is shown in figure 11.

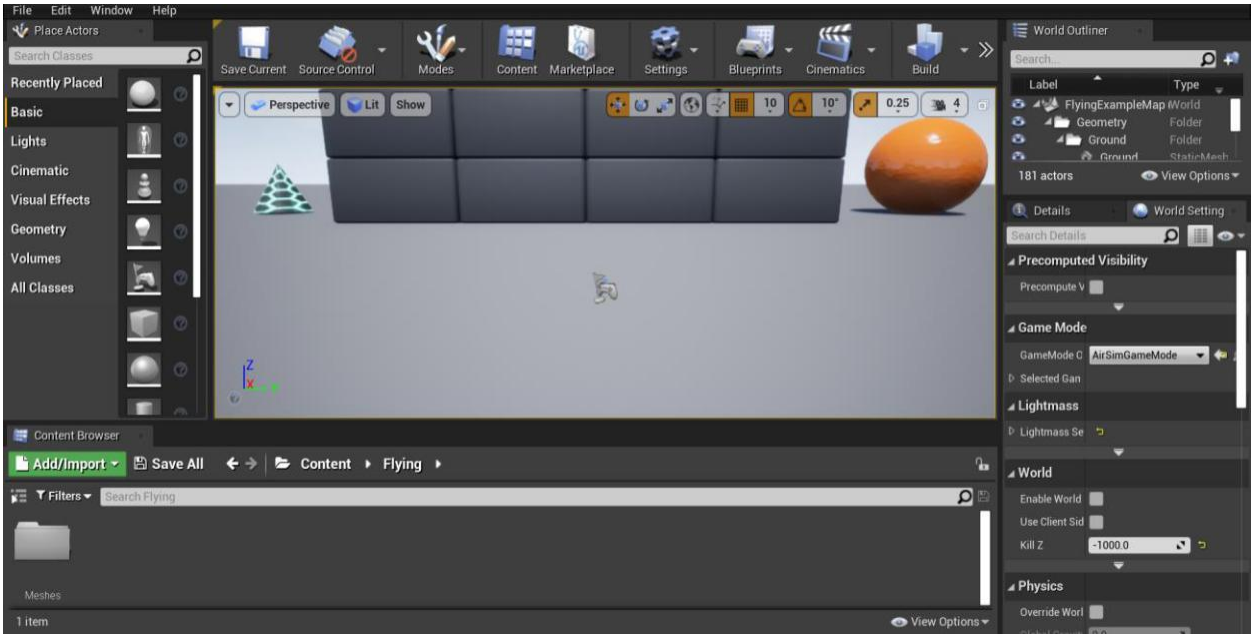

Figure 11: The test Blocks environment

The settings.json file is configured and key are added into the json objects as per the requirement.

In this case, the seetings.json file is as shown in figure 12.

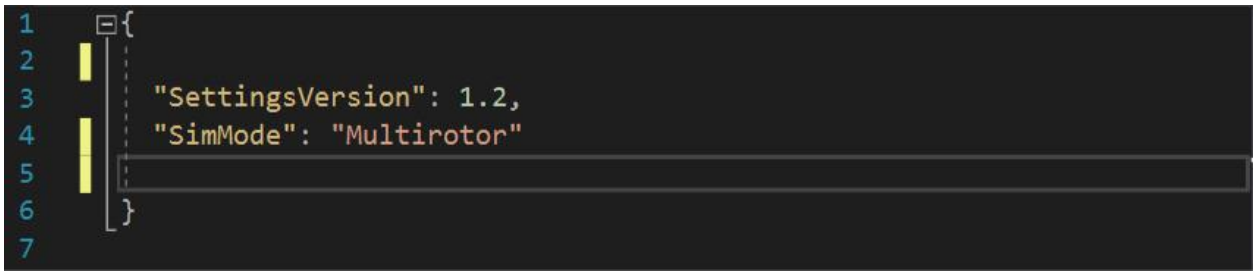

Figure 12: Setting json file

There are many more key values that can be added to the .json file, for this case only the SimMode is used to call the drone API.

Once the configurations are done, the drone can be tested manually in the environment. The available APIs allows a developer to program the drone as per requirement. Here, the drone is programmed to move around the environment as per user input. The manual drone control is achieved through a Bluetooth game controller connected to the same computer as show in figure 13.

Figure 13: Manual drone control

With manual control, the responsiveness of the drone is tested and APIs for lidar, depth camera and camera sensors are called as seen in figure 14.

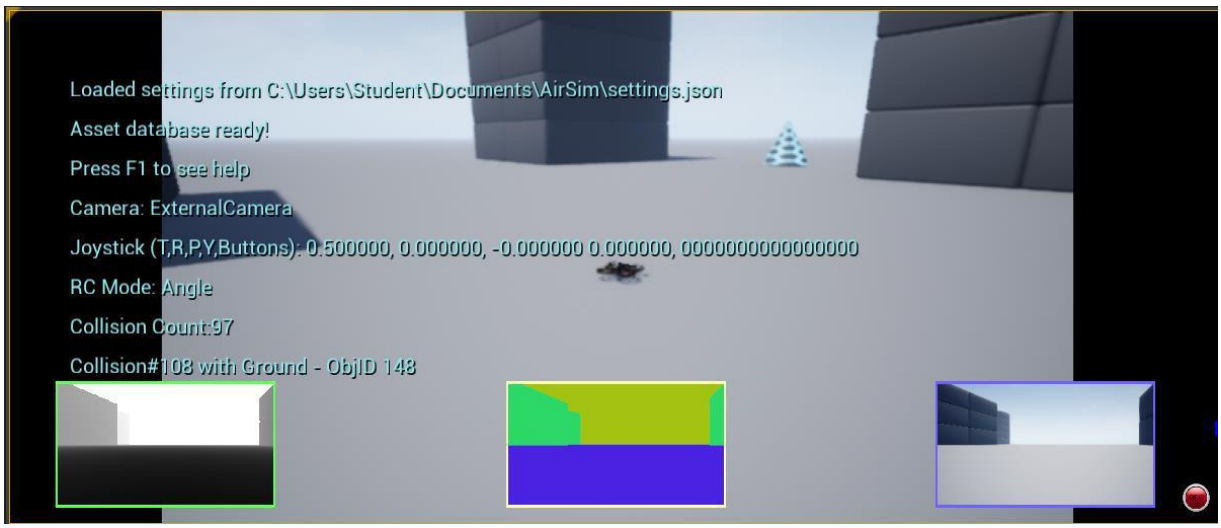

Figure 14: Camera APIs

The APIs are python files, programmed specifically to simulate these sensors are read data. Also, a counter is setup in the python file to detect number of collisions [11]. These collision counts are picked up every time the drone physically interacts with any of the object models in the environment. The objective is implementing the obstacle detection and avoidance algorithm to keep the collision count at zero. The data from the three cameras are stored as a csv file. The point cloud data generated from the lidar sensor is used to pick up the collision count.

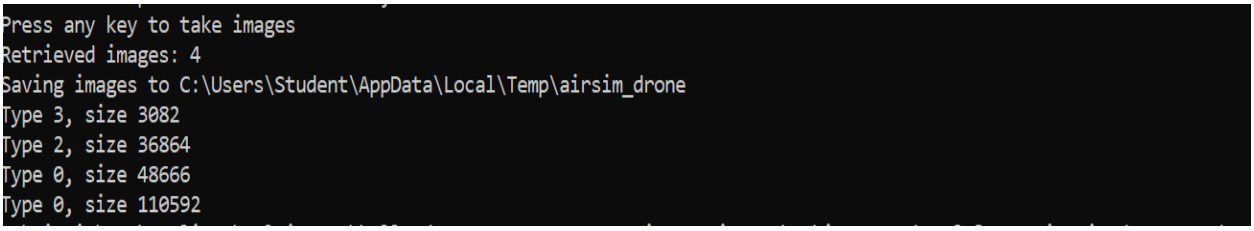

Figure 15: The camera API retrieves and stores the images in a file

Figure 15 is the output generated in the command prompt, camera APIs were called through a python script. The script captures and saves 4 images from the drone in the simulation.

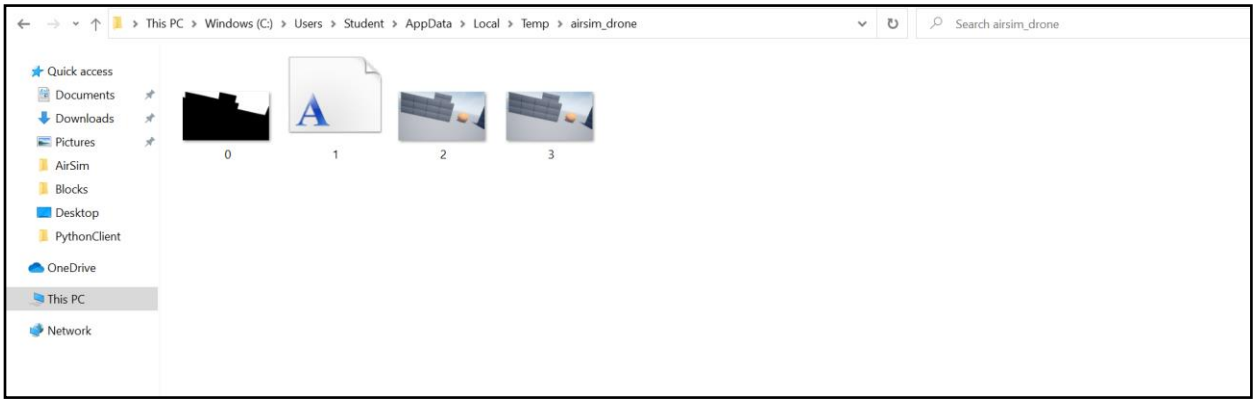

Figure 16: The stored images

The images captured and retrieved by the python script is stored in a local repository as seen in figure 16.

SECTION 7: THE ALGORITHM

Firstly, the deep Q- network or DQN is a reinforcement learning algorithm. It is a combination of Q-Learning with deep neural networks to let the algorithm work for highly sophisticated multi-dimensional environments [1].

The DQN algorithm tends to overestimate the values to tied to a specific algorithm []. The Double Q Learning is implemented to correct this.

It is a reinforcement learning algorithm that teaches the drone to detect and avoid static obstacles in its planned path in an environment.

7.1 DQN algorithm

The RL algorithm is the combination of Q learning with deep neural networks.

7.1.1 Q Learning

The Q-learning is a reinforcement learning algorithm that seeks to determine the best possible action to be taken given the current state. It's a reinforcement learning algorithm that's not based on a policy. It is classified as an off-policy algorithm because the Q-learning function continuously learns from actions outside of the current policy, such as taking or performing a random action, and therefore no policy is needed. Q-learning, in particular, aims to learn a policy that maximizes the overall incentive [1].

The ‘Q’ in Q-learning stands for quality. In this case, quality refers to how valuable a given action is in obtaining some potential reward [6].

Whenever Q-learning is executed, a q-table or a matrix is created that follows the function [state, action] and all values are initialized to zero. The q values in the table are then updated after an episode. This table then serves as a guide for the drone in determining the best course of action based on the q-value [4].

The next step involves the drone to interact with the simulated environment and make updates to the state action pairs in the created q-table. There are two ways this interaction can be achieved:

- Exploiting:
    - The drone communicates with the virtual environment in this action by referring to the q-table and viewing all of the potential actions for a given state. The drone then chooses an action based on the actions' maximum values.
- Exploring:
    - An action is selected at random, this is very crucial because acting randomly enables the discovery of new states that would not otherwise be available during the exploitation process.

The final step in Q-learning involves updating the q-table. These updates occur after each step or action and ends when an episode is done. The drone will not be able to achieve the objective in the first episode. With enough exploration, it will eventually converge and learn the optimal q-values.

The 3 steps in an episode can be viewed as:

1. (s1) is the start state, (a1) is the action taken and (r1) is the reward received
2. The Q-table is referenced with the (max) which is the highest value and the agent takes an action.
3. The Q-values are updated

The Q-learning rule:

Q [state1, action1] = Q[state1, action1] + learningRate(lr) * (reward1 + γ * np.max(Q[newState, :]) – Q[state1, action1]

Here, lr is the learning rate and it is interpreted as how much the new value is accepted compared to the old value.

γ (Gamma) is called the discount factor, it is typically used to balance the immediate rewards and also the future rewards.

Reward is the reward for performing a specific action in a specific state.

max() returns the maximum future reward and apply it to the current state.

The above rule can be represented as a mathematical formula:

$$Q^{\pi}(S,a) = (1 - \alpha)\,.\,Q^{\pi}(S_t,a_t) + \alpha\,.\,Q*(S,a) \quad (1)$$

Here, α is the learning rate
Q is the q-value
S is the state
a is the action

In equation (1) the agent will acquire the reward about action. With the reward, the q-values saved in the matrix can be updated and the action is made better the following time.

The DQN then replaces the q-table to a neural network with the weights θ. Next, the parameters are adjusted to make Q^{θ} very close to Q*.
To do this, the network is trained by a loss function $L_i(\theta_i)$.

$$\nabla_{\theta} L_i(\theta_i) \sim (y_t - Q_{\theta}(S,a))\nabla_{\theta} Q_{\theta}(S,a) \quad (2)$$

In equation (2), $y_i = R + \gamma\,.\,maxQ_{\theta}(S_{t+1},\ a_{t+1})$ if i is the initial step. Otherwise, $y_i = R$

In general, the algorithm updates the q-table by replay experience. The drone records all possible programmed experiences (state, action, reward, next state). The method where the weights are updated with mini batch on a training model is called replay experience [1].

7.1.2 The DQN structure for obstacle avoidance

The deep network has three types of layers. Each layer is a mathematical function to filter the data from the depth images captured by the drone.

- Input layer:
    - Depth image captured by the camera on the drone is taken as the input. This input image is available through Airsim using the relevant camera APIs. He obtained

image is resized to 80x80 pixels. The input state is then initialized with the first image as 80x80x4. This queues the images, replacing the old one every single time a new image enters the queue.

- Convolution and Pooling layer:
    - The image is preprocessed at the input layer, the image is then passed to the convolution layer and pooling layer. Then, each convolution layer will then pass this on to the rectifier activation function and connect with one pooling layer. Here, three convolution and max pooling layers have been deployed. The convolution layers stride is set to be 2. All pooling layers kernel is set to be 2 [1].
- Fully connected layer:
    - This layer is implemented to flatten the output of max pooling layer and convolution layer to acquire 1600 nodes. The flattened nodes are then entered into a fully connected layer, resulting in 512 outputs. The 512 nodes are then sent to the rectifier activation function (ReLU), these nodes are entered and output 13 nodes are obtained. These 13 nodes act as the Q-value for 13 possible actions [1].

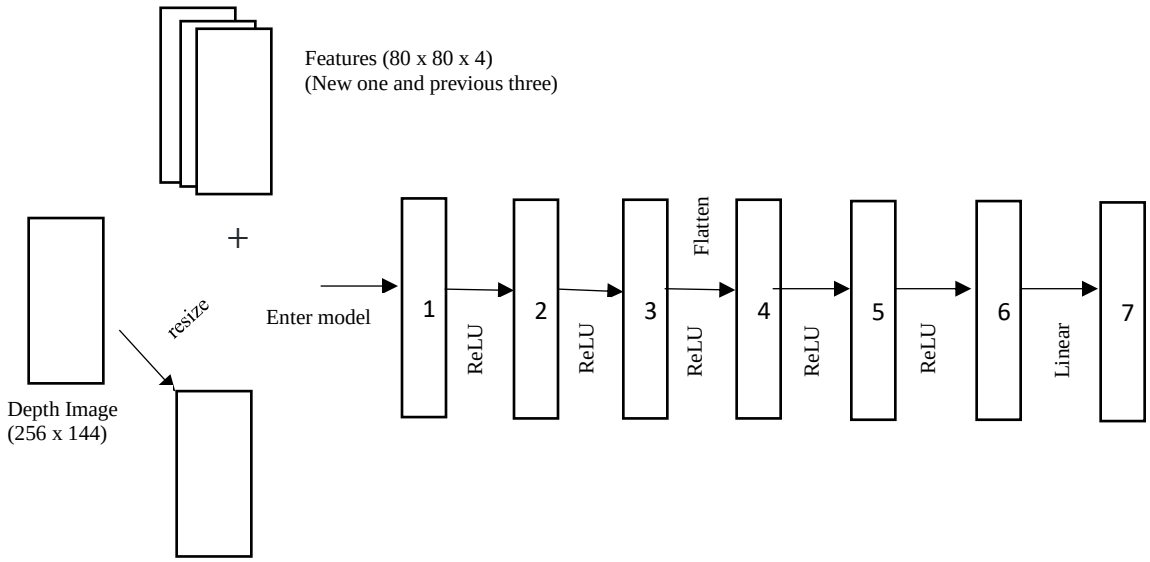

Figure 17: DQN model structure

The above model shown in the figure 15 is the overall deep Q-learning model structure for obstacle avoidance [1]. The layers 1 through 7 are as follows:

Layer 1 to layer 3 are Convolutional (stride = 2) and Pooling (kernel size = 2)

Layer 4 is the Flatten layer, layer 5 is the Fully connected layer with 1600 nodes.
Layer 6 is the Fully connected layer with 512 nodes and the final layer gives an output of 13 q-values.

7.1.3 Training the network

Training the network refers to updating the q-table or matrix. Practically, there are 2 types of Q-networks in training. The first one is Q_{value} - network and the other one is Q_{target} – network. The former network will be updated in each training state whereas the latter will be updated every 500 steps [3] [1].
For every 50 steps, a training state is entered and mini-batch is randomly selected from experience.

7.2 Implementing DQN for obstacle detection

The DQN algorithm was implemented by integrating Airsim with openAI gym and keras-rl for autonomous flight with obstacle avoidance.

- openAI gym:
    - openAI gym is a toolkit for developing and comparing reinforcement learning algorithms. This is an open source library, which grants access to standardized set of environments.

- keras-rl:
    - keras-rl is used to implement the reinforcement learning algorithm in python and integrates with the deep learning library Keras.

keras-rl works with openAI gym out of the box. This allows developers to evaluate and play around with different algorithms.

7.3 Simulation results

The DQN algorithm has been implemented using python. The algorithm has been tested for both open and closed environments.

The drone movements are defined as takeoff for initial liftoff at the start location or the initial geo point, Rotation direction using yaw and roll are also defined.

An episode during the training of DQN is considered to be the max distance traveled by the drone before a collision occurs.

```
Episode:00001,Step:00022 The drone is moving right   , yaw=1.0,  roll=0.167,  reward=-0.08, distance=247.35,experience len=00021,inner loop=0.6427s
episode=00001,step=00023,we are observing the env,the action is random......
Episode:00001,Step:00023 The drone is moving left    , yaw=0.5,  roll=-0.667,  reward=-0.08, distance=247.40,experience len=00022,inner loop=0.6641s
episode=00001,step=00024,we are observing the env,the action is random......
Episode:00001,Step:00024 The drone is moving left    , yaw=1.0,  roll=-0.333,  reward=-0.08, distance=247.45,experience len=00023,inner loop=0.6678s
episode=00001,step=00025,we are observing the env,the action is random......
Episode:00001,Step:00025 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=247.50,experience len=00024,inner loop=0.6749s
episode=00001,step=00026,we are observing the env,the action is random......
Episode:00001,Step:00026 The drone is moving left    , yaw=0.5,  roll=-1.000,  reward=-0.08, distance=247.55,experience len=00025,inner loop=0.6687s
episode=00001,step=00027,we are observing the env,the action is random......
Episode:00001,Step:00027 The drone is hovering, yaw=0.6,  roll=0.000,  reward=-0.08, distance=247.62,experience len=00026,inner loop=0.6717s
episode=00001,step=00028,we are observing the env,the action is random......
Episode:00001,Step:00028 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=247.69,experience len=00027,inner loop=0.6626s
episode=00001,step=00029,we are observing the env,the action is random......
Episode:00001,Step:00029 The drone is moving right   , yaw=1.0,  roll=0.333,  reward=-0.08, distance=247.77,experience len=00028,inner loop=0.6620s
episode=00001,step=00030,we are observing the env,the action is random......
Episode:00001,Step:00030 The drone is moving right   , yaw=0.0,  roll=0.667,  reward=-0.08, distance=247.86,experience len=00029,inner loop=0.6672s
episode=00001,step=00031,we are observing the env,the action is random......
Episode:00001,Step:00031 The drone is moving right   , yaw=0.0,  roll=0.167,  reward=-0.08, distance=247.96,experience len=00030,inner loop=0.6683s
episode=00001,step=00032,we are observing the env,the action is random......
Episode:00001,Step:00032 The drone is moving left    , yaw=0.0,  roll=-1.000,  reward=-0.08, distance=248.07,experience len=00031,inner loop=0.6851s
episode=00001,step=00033,we are observing the env,the action is random......
Episode:00001,Step:00033 The drone is moving left    , yaw=0.0,  roll=-1.000,  reward=-0.08, distance=248.19,experience len=00032,inner loop=0.6594s
episode=00001,step=00034,we are observing the env,the action is random......
Episode:00001,Step:00034 The drone is moving left    , yaw=0.0,  roll=-1.000,  reward=-0.08, distance=248.30,experience len=00033,inner loop=0.6688s
episode=00001,step=00035,we are observing the env,the action is random......
Episode:00001,Step:00035 The drone is moving right   , yaw=0.0,  roll=0.667,  reward=-0.08, distance=248.39,experience len=00034,inner loop=0.6681s
episode=00001,step=00036,we are observing the env,the action is random......
Episode:00001,Step:00036 The drone is moving right   , yaw=0.0,  roll=0.333,  reward=-0.08, distance=248.46,experience len=00035,inner loop=0.6667s
```

Figure 18: Command prompt output

The logs shown in figure 18 are the steps taken by the drone in each episode, a reward and experience is generated and q-values are updated in the table until a collision occurs.

Figure 19: DQN implementation in an open environment

Figure 19 shows the algorithm implementation in an open environment. The drone resets to the start point (initial geo point location).

```
pisode:00001,Step:00098 The drone is moving right   , yaw=0.0,  roll=0.833,  reward=-0.08, distance=257.56,
pisode:00001,Step:00099 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=257.68,
---------Random Action----------
pisode:00001,Step:00100 The drone is moving left    , yaw=0.5,  roll=-0.667,  reward=-0.08, distance=257.79
pisode:00001,Step:00101 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=257.89,
pisode:00001,Step:00102 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=257.99,
pisode:00001,Step:00103 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=258.08,
pisode:00001,Step:00104 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=258.17,
---------Random Action----------
pisode:00001,Step:00105 The drone is moving left    , yaw=0.5,  roll=-1.000,  reward=-0.08, distance=258.24
---------Random Action----------
pisode:00001,Step:00106 The drone is moving left    , yaw=1.0,  roll=-0.333,  reward=-0.08, distance=258.30
pisode:00001,Step:00107 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=258.37,
pisode:00001,Step:00108 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=258.45,
pisode:00001,Step:00109 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=258.53,
pisode:00001,Step:00110 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=258.60,
pisode:00001,Step:00111 The drone is moving right   , yaw=0.0,  roll=0.833,  reward=-0.08, distance=258.66,
pisode:00001,Step:00112 The drone is moving right   , yaw=0.0,  roll=0.833,  reward=-0.08, distance=258.71,
pisode:00001,Step:00113 The drone is moving right   , yaw=0.0,  roll=0.833,  reward=-0.08, distance=258.75,
---------Random Action----------
pisode:00001,Step:00114 The drone is moving left    , yaw=0.0,  roll=-0.833,  reward=-0.08, distance=258.77
pisode:00001,Step:00115 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=258.79,
pisode:00001,Step:00116 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=258.81,
pisode:00001,Step:00117 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=258.83,
pisode:00001,Step:00118 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=-0.08, distance=258.83,
pisode:00001,Step:00119 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=0.08, distance=258.82,e
pisode:00001,Step:00120 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=0.08, distance=258.80,e
pisode:00001,Step:00121 The drone is moving right   , yaw=0.5,  roll=0.833,  reward=0.08, distance=258.78,e
```

Figure 20: Agent does a random action

The above shown figure 20 depicts the random action done by the agent as explained in section 7.1.1. It is a step the agent takes during Q-learning. Figure 21 shows the obstacle avoidance in a closed environment like an office space.

Figure 21: DQN implementation in a closed environment

CONCLUSION

The work in this project started with the study of simulation environments, choosing drone controller hardware. Unreal engine has been chosen as the game engine for simulating the test environment. This game engine was preferred because stable Airsim libraries and APIs have been developed to work with Unreal engine. The PX4 stack has been picked as the software stack since it provides drone developers with a scalable collection of tools to share technology and create a customized solution for multiple drone applications. The Pixhawk 2.4.8 is the flight controller used in this work. The flight controller has been manually calibrated with respect to true north. The PX4 stack runs in the Pixhawk flight controller which in turn executes python scripts to simulate drone movement. A generic quadcopter is the drone of choice for the algorithm implementation since most drones available to the public are quadcopters.

Deep Q-network or DQN has been the algorithm of interest in this work. In order to achieve obstacle detection and avoidance, the DQN algorithm has been implemented in python to simulate the movement of a drone between the start location and end location without interacting with static models in the simulation environment. The Deep Q-network implemented has 3 types of layers each layer acting as a mathematical function to generate a q-value. The generated q-value is updated in the matrix and can be then used by the agent to plan its path and avoid any obstacles in that path.

The DQN algorithm has been implemented as a python script and connected to Airsim with Unreal engine as an API. Although obstacle avoidance cannot be achieved in the first run, after multiple episodes the drone successfully moves from its initial location to the destination without coming into contact with the surrounding environment.

The Algorithm was implemented for two types of environment, a closed environment and an open environment. The agent was able to train itself faster in the closed environment since the duration of episodes are smaller.

REFERENCES

[1] T. Wu, S. Tseng, C. Lai, C. Ho and Y. Lai, "Navigating Assistance System for Quadcopter with Deep Reinforcement Learning," 2018 1st International Cognitive Cities Conference (IC3), 2018, pp. 16-19, doi: 10.1109/IC3.2018.00013.

[2] G.Lample and D. S.Chaplot, "Playing FPS Games with Deep Reinforcement Learning.," *in AAAI*, 2017, pp. 2140–2146.

[3] Lei He, Nabil Aouf, James F. Whidborne, Bifeng Song." Deep Reinforcement Learning based Local Planner for UAV Obstacle Avoidance using Demonstration Data". 2020

[4] J.Hwangbo, I.Sa, R.Siegwart, andM.Hutter, "Control of a Quadrotor with Reinforcement Learning," IEEE Robot. Autom. Lett., vol. 2, no. 4, pp. 2096–2103, 2017.

[5] P.Long, T.Fan, X.Liao, W.Liu, H.Zhang, andJ.Pan, "Towards Optimally Decentralized Multi-Robot Collision Avoidance via Deep Reinforcement Learning," 2017.

[6] G.Kahn, A.Villaflor, V.Pong, P.Abbeel, andS.Levine, "Uncertainty Aware Reinforcement Learning for Collision Avoidance," 2017.

[7] S.-Y. Shin, Y.-W. Kang, and Y.-G. Kim, "Obstacle Avoidance Drone by Deep Reinforcement Learning and Its Racing with Human Pilot," *Applied Sciences*, vol. 9, no. 24, p. 5571, Dec. 2019.

[8] Chakravarty, P.; Kelchtermans, K.; Roussel, T.; Wellens, S.; Tuytelaars, T.; Van Eycken, L. "CNN-based single image obstacle avoidance on a quadrotor". In Proceedings of the 2017 IEEE International Conference on Robotics and Automation (ICRA), Singapore, 29 May–3 June 2017; pp. 6369–6374.

[9] C. Ma, Y. Zhou and Z. Li, "A New Simulation Environment Based on Airsim, ROS, and PX4 for Quadcopter Aircrafts," 2020 6th International Conference on Control, Automation and Robotics (ICCAR), 2020, pp. 486-490, doi: 10.1109/ICCAR49639.2020.9108103.

[10] J. G. C. Zuluaga, J. P. Leidig, C. Trefftz and G. Wolffe, "Deep Reinforcement Learning for Autonomous Search and Rescue," NAECON 2018 - IEEE National Aerospace and Electronics Conference, 2018, pp. 521-524, doi: 10.1109/NAECON.2018.8556642.

[11] S.Shah, D.Dey, C.Lovett, and A.Kapoor, “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles,” pp. 1–14, 2017.

[12] A. Koubâa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith and M. Khalgui, "Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey," in *IEEE Access*, vol. 7, pp. 87658-87680, 2019, doi: 10.1109/ACCESS.2019.2924410.

[13] C. M. Torres-Ferreyros, M. A. Festini-Wendorff and P. N. Shiguihara-Juárez, "Developing a videogame using unreal engine based on a four stages methodology," *2016 IEEE ANDESCON*, 2016, pp. 1-4, doi: 10.1109/ANDESCON.2016.7836249.

[14] J. Leško, M. Schreiner, D. Megyesi and L. Kovács, "Pixhawk PX-4 Autopilot in Control of a Small Unmanned Airplane," 2019 Modern Safety Technologies in Transportation (MOSATT), 2019, pp. 90-93, doi: 10.1109/MOSATT48908.2019.8944101.

APPENDIX: PYTHON SCRIPT

```
# This python script is the DQN algorithm implementation with convolution layers


# importing all necessary libraries
import numpy as np
import gym   # openAI gym

import gym_airsim.envs
import gym_airsim

import argparse

from keras.models import Model, Sequential  # keras neural network library
from keras.layers import Input, Reshape, Dense, Flatten, Conv2D, concatenate
from keras.optimizers import Adam
from keras.callbacks import TensorBoard  # tensorborad output
from keras.utils import plot_model

from callbacks import *
from rl.agents.dqn import DQNAgent
from rl.policy import LinearAnnealedPolicy, EpsGreedyQPolicy
from rl.memory import SequentialMemory
from rl.processors import MultiInputProcessor
import os
os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/vikas/airsim/bin/'

parser = argparse.ArgumentParser()
parser.add_argument('--mode', choices=['train', 'test'], default='train')
parser.add_argument('--env-name', type=str, default='AirSimEnv-v42')
parser.add_argument('--weights', type=str, default=None)
args = parser.parse_args()


env = gym.make(args.env_name)
np.random.seed(123)
env.seed(123)
nb_actions = env.action_space.n


#Obtaining shapes from Gym environment
img_shape = env.simage.shape
vel_shape = env.stotalvelocity.shape
dst_shape = env.stotaldistance.shape
geo_shape = env.stotalgeofence.shape

#Keras-rl interprets an extra dimension at axis=0
#added on to our observations, so we need to take it into account
img_kshape = (1,) + img_shape

#Sequential model for convolutional layers applied to image
image_model = Sequential()
```

```
image_model.add(Conv2D(32, (4, 4), strides=(4, 4) ,activation='relu', input_shape=img_kshape, data_format =
"channels_first"))
image_model.add(Conv2D(64, (3, 3), strides=(2, 2),  activation='relu'))
image_model.add(Flatten())

#Input and output of the Sequential model
image_input = Input(img_kshape)
encoded_image = image_model(image_input)

#Inputs and reshaped tensors for concatenate after with the image
velocity_input = Input((1,) + vel_shape)
distance_input = Input((1,) + dst_shape)
geofence_input = Input((1,) + geo_shape)
vel = Reshape(vel_shape)(velocity_input)
dst = Reshape(dst_shape)(distance_input)
geo = Reshape(geo_shape)(geofence_input)


#Concatenation of image, position, distance and geofence values.
#3 dense layers of 256 units
denses = concatenate([encoded_image, vel, dst, geo])
denses = Dense(256, activation='relu')(denses)
denses = Dense(256, activation='relu')(denses)
denses = Dense(256, activation='relu')(denses)

#Last dense layer with nb_actions for the output
predictions = Dense(nb_actions, kernel_initializer='zeros', activation='linear')(denses)

model = Model(
      inputs=[image_input, velocity_input, distance_input, geofence_input],
      outputs=predictions
      )


train = True



# configure and compile our agent. You can use every built-in Keras optimizer

memory = SequentialMemory(limit=100000, window_length=1)                    #reduce memory

processor = MultiInputProcessor(nb_inputs=4)

# Select a policy. eps-greedy action selection, which means that a random action is selected
# with probability eps. We anneal eps from 1.0 to 0.1 over the course of 1M steps. This is done so that
# the agent initially explores the environment (high eps) and then gradually sticks to what it knows
# (low eps). We also set a dedicated eps value that is used during testing. Note that we set it to 0.05c
# so that the agent still performs some random actions. This ensures that the agent cannot get stuck.

policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_min=.1, value_test=0.0,
                        nb_steps=100000)
```

```
dqn = DQNAgent(model=model, processor=processor, nb_actions=nb_actions, memory=memory,
nb_steps_warmup=50,
           enable_double_dqn=True,
           enable_dueling_network=False, dueling_type='avg',
           target_model_update=1e-2, policy=policy, gamma=.99)

dqn.compile(Adam(lr=0.00025), metrics=['mae'])

if train:
# taining the agent
   log_filename = 'dqn_{}_log.json'.format(args.env_name)
   callbacks = [FileLogger(log_filename, interval=10)]
   callbacks += [TrainEpisodeLogger()]
   #tb_log_dir = 'logs/tmp'
   #callbacks = [TensorBoard(log_dir=tb_log_dir, histogram_freq=0)]
   dqn.fit(env, callbacks = callbacks, nb_steps=250000, visualize=False, verbose=0, log_interval=100)


   # After training is done, we save the final weights.
   dqn.save_weights('dqn_weights.h5f'.format(args.env_name), overwrite=True)

else:

   dqn.load_weights('dqn_weights.h5f')
   dqn.test(env, nb_episodes=100, visualize=False)


#############################################################################################
```