University of Genova

Master's Degree in Robotics Engineering

Real Time Drone Teleoperation with Virtual Reality and Motion Feedback Assist

by

Luca Covizzi

Thesis submitted for the Master's degree of *Robotics Engineering*

March 2022

Prof. Ing. Matteo Zoppi — Supervisor
Phd Mohamed Sadiq Ikbal — Supervisor

Department of Mechanical, Energy, Management and Transport Engineering

Abstract

Unmanned aerial vehicles (UAVs) are used in several situation, mainly human piloting, statistically more dangerous than a fully autonomous navigation. UAV accidents, hence, happen at a considerably great rate. As a result, there is a need to improve UAV pilot training and performance in order to reduce accidents. In this thesis we hypothesis a method to integrate a simulated teleoperation module for a drone with the SP7 motion simulator, in order to provide motion cue in case of obstacle detection with the aim of preventing collision. To test the implementation a virtual environment is built with Unreal Engine, while the navigation module is implemented with AirSim, a plugin developed by Microsoft for drone simulations. The drone's response in presence of a near obstacle is handled by a Force Field Algorithm. As consequence, the drone motion data are sent to the SP7 in real-time, trough a Motion Cue Algorithm controller. Experiments with a jury are conducted to evaluate if the motion cue improves obstacle avoidance, perception and response control during the piloting.
Keywords: *simulation, UAV, motion feedback, hybrid-navigation*

Contents

List of Figures

List of Tables

Nomenclature

General Mechanics

R Revolute Joint

S Spherical Joint

Acronyms / Abbreviations

AI Artificial Intelligence

API Application Programming Interface

AR Augmented Reality

DOF Degree of Freedom

FF Force Feedback

FOV Field of View

HMD Head Mounted Display

HTL Hardware In The Loop

MAV Micro Air Vehicle

MCA Motion Cue Algorithm

ML Machine Learning

RC Remote Control

STL Software In The Loop

UAV Unmanned Aerial Vehicle

UE4 Unreal Engine 4

VR Virtual Reality

Part I

Section I

Introduction

Drones, also know as Unmanned Aerial Vehicles (UAVs), are aircrafts remotely controlled or capable to be autonomous. They are mainly used for patrolling applications with the purpose of collecting significant information, e.g., using camera and other various sensor such as LiDAR, GPS/GNSS. The latest innovations in consumer hardware increased drones popularity and availability.
Quadricopters, mechanical models mainly designed for Micro Air Vehicles (MAV), aroused a lot of interest in the recent years. Research on MAV covers a wide range of topics including control system, sensor fusion, Machine Learning (ML) and Artificial Intelligence (AI).

Figure 1.1: Quadrotor Model

Although mechanical dynamic of rotor driven models has been widely surveyed, R&D challenges are still focused on the implementation of a robust, efficient and smart autonomous flight system.
Due to the recent technological advancements and research results, Virtual and Augmented reality fields improved functionalities in drones navigation (6). A

virtual environment enables a realistic perspective through the camera acquisition. The user can directly interact with the environment by using head-mounted device (e.g., HTC Vive) and therefore VR is suitable for training and simulation approaches.
VR headsets provide a field of view (FOV) up to 200 degrees that produces a great immersive experience.

Figure 1.2: HTC VIVE PRO

Despite this special features, the lack of tactile feedback limits the perception of the surrounding environment (7). This is due to the fact that the simultaneous stimulation of the visual, somatosensory and proprioception ensures the correct human awareness of motion (8). The integration of haptic and/or tactile devices or motion platform could overcome these issues, enhancing the user experience.

1.1 Problem Definition

In my thesis work, my goal was to integrate a teleoperation module for a drone with the SP7 motion simulator. SP7 motion simulator is a parallel manipulator with 6RSS-R mechanism capable of handling multi-functional virtual scenario simulations. It has been integrated with the Unreal Engine and in details a module that can teleoperate the drone in real-time with both visual and motion cues fed to the motion simulator. Upon validation of motion perception with the real-time camera visuals, the module will be translated to be adapted for a virtually simulated environment. The most common approaches for application in hostile environments have to deal with problems concerning energy and control-command. Thus hardware and software need to be addressed to cope

with these issues. Nevertheless the proposed application could solve these kind of problems. In fact, an hybrid navigation implemented with the aforementioned module enables the user to improve the perception of the scenario.
Moreover, the general idea is to ensure motion feedback from the platform to the user in order to have a better result in case of human navigation. In particular, a cue from the SP7 can advertise the user about the proximity of obstacles. Therefore this implementation is expected to improve the obstacle avoidance and preventing collisions.

1.2 Problem Approach

The environment has been built with Unreal Engine, one of the leading game engine, in order to create a very immersive scenario where to pilot the drone. The user wears the HTC Vice, which provides visual and audio stimuli. AirSim, developed by Microsoft, is a simulator for drones and cars which allows to performs software in the loop simulations. Even if it has been designed with the aim of reinforcement learning algorithm for autonomous navigation, it exposes APIs to retrieve data and control vehicles in a independent way. Therefore a remote control [RC] setup and a force feedback [FF] algorithm have been implemented. The RC (e.g., cloche, throttle) is used for piloting the drone, while the FF has the aim to help avoiding obstacles. Since it is an assisted teleoperation, the SP7 receives data extracted from the UE and it starts moving in such a way that the pilot is able to perceive not only the presence of an obstacle, but also its position. Figure 1.3 illustrates the concept idea to be implemented. Suppose now that sensors detect an obstacle at the side of the drone, then the platform roto-translates in the opposite direction, therefore it is supposed to invite the pilot to move away.

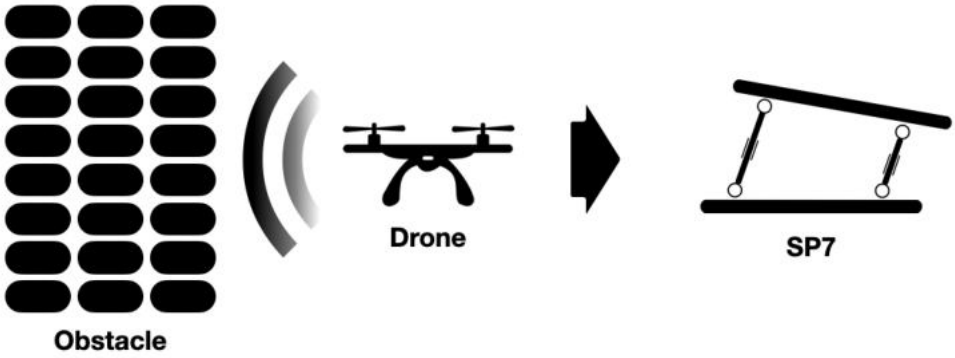


Figure 1.3: Concept Overview

Data firstly pass thought the motion cue controller, a work proposed by PMAR members. The aim of the motion cue controller is to process motion data in order to feed properly the platform. Figure 1.4 shows the final overall architecture. The user in each frame receives both visual and motion signals, that are synchronized and supposed to improve the piloting.

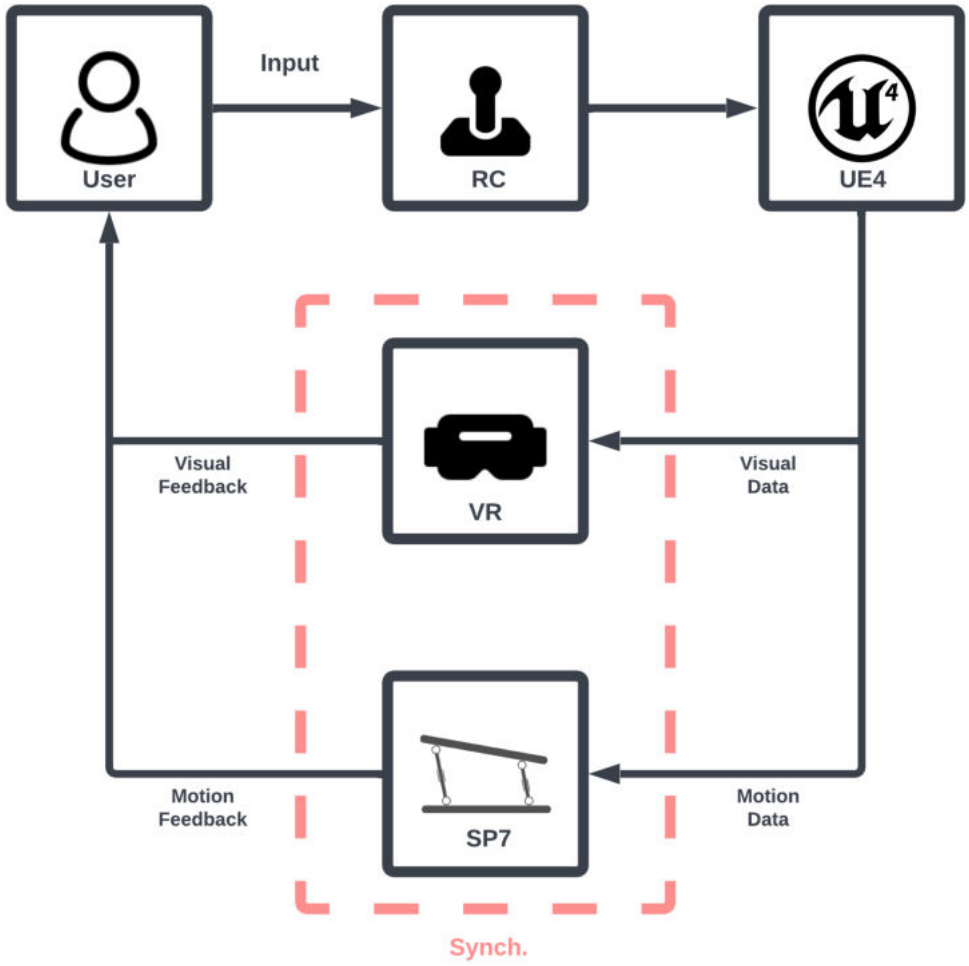


Figure 1.4: Overall Architecture

In order to validate the implementation, several tests have been designed and a jury, after having experienced the framework, has compiled a qualitative questionnaire. At the end the scores have been post process and discussed.

1.3 Document's Structure

The rest of this thesis is structured in the following way :

- **Chapter 2**: in this chapter is described the current State of the Art, with particular attention to the UAV simulation tools, a deeper investigation on

AirSim, Stewart Platform and haptic\motion feedback

- **Chapter 3**: in this chapter the realization is described, starting from the UE environment, the AirSim integration, concluding with the FF algorithm implementation.
- **Chapter 4:** in this chapter the design of the experiment is investigated, by referring to the tests that have been performed.
- **Chapter 5**: in this chapter the results coming from the testing phase are discussed.
- **Chapter 6**: in this chapter the description of the thesis is concluded and some future works are proposed.

State of the Art

In recent years, drones have become more and more subject of interest in the research field and their popularity continues to increase since autonomous and semi-autonomous vehicles have reduced their cost while increasing their utility and ease of use (9). Although drone were designed and used for military purposes in the old years, the possibility to use them in a situation where the presence of humans is impossible or dangerous have prompted the research to focus on drones, in order to carry out several operations such as patrolling and search-and-rescue.

2.1 UAV Simulation Tools

2.2 Simulation Software

During the recent years, the virtual-to-real learning has encountered particular interest.
AirSim (1) is an Open Source and Cross-Platform simulator for drones that provides physically and visually realistic simulations using either Hardware-In-The-Loop (HIL) or Software-In-The-Loop (SIL). The physics engine is based on Unreal Engine, providing real-time HIL simulations due to the high frequency performances.
The reliability and the life-likeness of the virtual environment have a key role during the simulation since it provides data to the sensors that will be used in the train phase. The desired state may be monitored by the user through a remote control or may be programmed autonomously according to a navigation algorithm and with this approach a huge amount of data is involved by using sensors such as cameras (10) or LiDAR (11). For instance, if researchers want to

test a computer vision algorithm in an outdoor environment, advanced rendering is essential to reproduce finer details such as shadows, reverberation, reflections and so on in order to avoid changes in the real application.
Bondi *et Al.* (12) presented AirSim-W: a simulation environment designed specifically for the domain of wildlife conservation with drones. Scenes, segmentation and deep images can be collected during simulation and this approach allows researchers to experiment with deep learning, computer vision, and reinforcement learning algorithms for autonomous navigation. As result, despite of labeling real data, this process is more cost-effective and take less time.

2.2.1 AirSim

AirSim (1) is a Unreal Engine-based simulator that provides physically and graphically accurate simulations. It features a high-frequency physics engine enabling real-time hardware-in-the-loop simulations, as well as compatibility for common protocols. The simulator was built from ground to be adaptable to new vehicle kinds, hardware platforms, and software protocols.

2.2.1.1 AirSim Architecture

The simulator has a modular design with an emphasis on accessibility. The essential component includes environment model, vehicle model, physics engine, sensor models, rendering interface, public API layer and a layer for vehicle firmware. The flight controller firmware (e.g., PX4 or ROSFlight) is often included in the configuration of an autonomous aerial vehicle. The flight controller receives as inputs the target state and sensor data, computes an estimate of the current state, and outputs actuator control signals to achieve the desired state. In the case of quadrotors, for instance, the user may provide desired *pitch*, *roll*, and *yaw* angles as desired states, and the flight controller may use sensor data from the accelerometer and gyroscope to estimate the current angles before computing the motor outputs to reach the desired angles.
The simulator sends sensor data from the simulated world to the flight controller during simulation. The flight controller generates actuator signals, which are read by the simulator's vehicle model component. The vehicle model's objective is to compute the forces and torques produced by the simulated actuators. There may also be forces caused by drag, friction, and gravity. The physics engine then em-

ploys these forces and torques as inputs to determine the next kinematic state of bodies in the simulated environment. This kinematic state of bodies, in conjunction with the environment models for gravity, air density, air pressure, magnetic field, and geographic position (GPS coordinates), serves as the foundation for the simulated sensor models (1). Therefore the human operator by using remote control can set the desired state as well as autonomous algorithms. Therefore it is possible to perform high-level computational task including evaluating the desired trajectory, performing SLAM and so on.
Due to the Unreal Engine graphic performances, computers are able to process reliably data provided by sensors such as camera or LiDAR. As consequence computers directly interface with the simulator thought APIs calls that can access to the data streams, vehicle states and moreover they are able to send commands. This is a key point since due to the simulation is possible to develop and test algorithm in order to move to real application without adding further modifications.

2.2.1.2 Vehicle Model

AirSim is equipped with an architecture for defining a vehicle as a rigid body with an arbitrary number of actuators that produces forces and torques. The vehicle model contains properties such as mass, inertia, linear and angular drag coefficients, coefficients of friction, and restitution, which are utilized by the physics engine to compute rigid body dynamics (1).

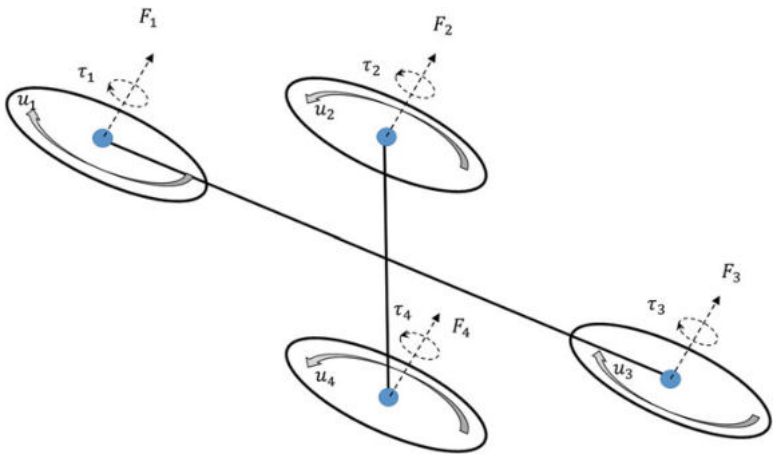


Figure 2.1: Quadrotor Model (1)

The figure above shows the quadrotor model, where the control input u_i drives the rotational speed of the propellers located at the four *vertices*. The forces and the torques generated by the rotors are computed by referencing to (13).

$$F_i = C_T \rho \omega_{max}^2 D^4 u_i \quad \text{and} \quad \tau_i = \frac{1}{2\pi} C_{pow} \rho \omega_{max}^2 D^5 u_i \tag{2.1}$$

Where:

- C_T: Thrust coefficient (based on the physical characteristics of the propeller)
- C_{pow}: Power coefficient (based on the physical characteristics of the propeller)
- ρ: Air density
- D: Propeller's diameter
- ω_{max}: Max angular velocity in revolutions per minute

Due to the propellers actions, it is possible to simulate vehicles that have attributes such as Vertical Take-Off and Landing (VTOL).

2.2.1.3 Physics Engine

Position, orientation, linear velocity, linear acceleration, angular velocity and angular acceleration defines the kinematic state of the vehicle. Therefore the aim of the physic engine is to evaluate the next kinematic state given as input forces and torques acting on it. This is done within an update loop at high frequency (1000 Hz), so that is suitable for real-time simulation scenarios including high-speed drone control (1).

2.2.1.4 Sensors

AirSim currently supports the following sensors:

- Camera
- Barometer
- IMU

- GPS
- Magnetometer
- Distance Sensor
- LiDAR

All the sensor models are implemented as C++ header-only library and they can be independently used outside of the plugin. As described in section 3.2.2 sensor models are defined as abstract interfaces therefore they offers a very versatile implementation.

2.2.1.5 AirSim code structure

The core of the code implementation is located in *AirLib*, a self-contained library that can be compiled thought any **C++ 11** compiler. The aforementioned library is characterized by the following component:

- **Physics Engine**: This is header-only physics engine. It is designed to be fast and extensible to implement different vehicles
- **Sensor models**: This is header-only models for sensor
- **Vehicle models**: This is header-only models for vehicle configurations and models
- **API related files**: This part of AirLib provides abstract base class for our APIs and concrete implementation for specific vehicle platforms. It also has classes for the RPC client and server.

Moreover the vehicle-specific APIs are defined in the `api/` subfolder, along with required structs. The `AirLib/src/` contains `.cpp` files with implementations of various methods defined in the `.hpp` files.

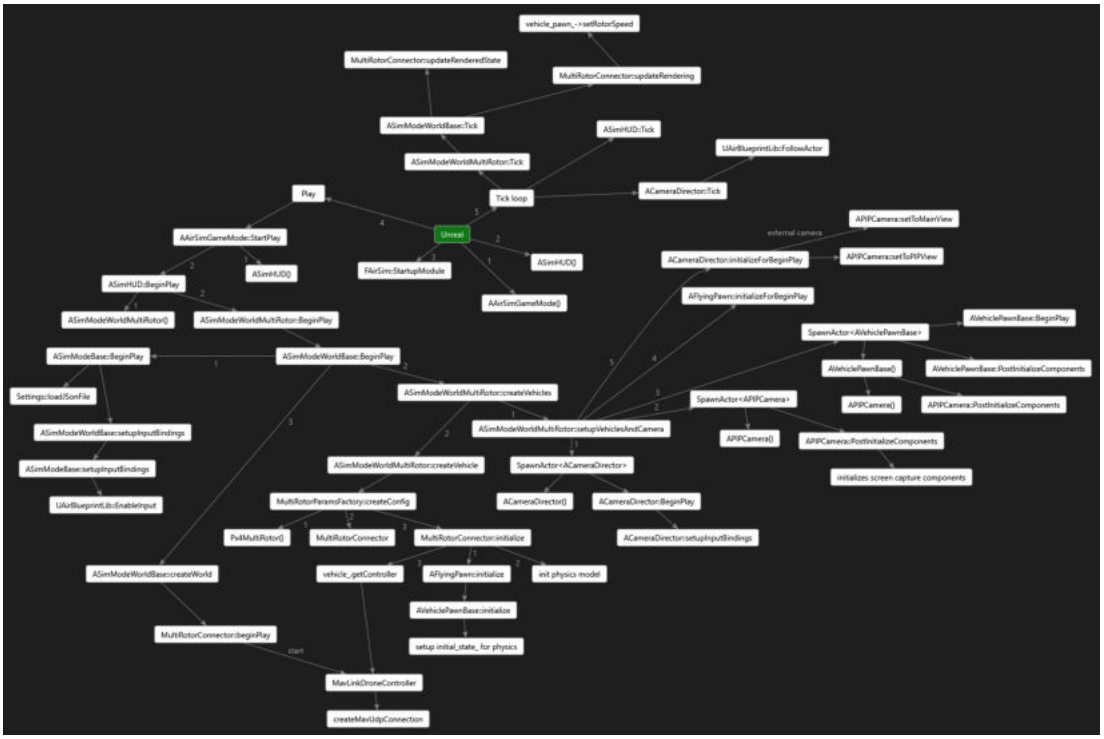


Figure 2.2: How AirSim is Loaded and Invoked by the Unreal Game Engine (1)

2.3 Simulation Platform

The future work will focus on existing hardware with parallel manipulator. In particular, the teleoperation module for the drone will be integrated with the SP7 motion simulator in order to provide motion cues.
A parallel manipulator presents a chain mechanism characterized by a close-loop kinematic. In this structure independent kinematic chains connect the base to the end effector (14). Research on close-loop parallel manipulators focused mostly on kinematic and dynamic analysis (15) (16), finding out efficient technique to apply mainly on Stewart Platform. Six actuators, each placed on the ground, control this type of 6 Degree of freedoms (DOF) mechanism, designed by Stewart (17). Due to the aforementioned specifics this platform is typically employed in flight simulations. The main purpose of the flight simulator is to replicate the aircraft behavior in complete safety, giving pilots the possibility to improve their skills in real applications (18). The number of end-effector DOF and the kind of motion performed in the space allow to classify parallel manipulators. Starting from the Stewart platform, researchers have implemented several types of parallel mechanisms (19). In accordance with the literature, researchers focused widely also on 3 DOF platforms (20) due to the versatility and the lower power consumption. Parallel manipulators present other features (21) such as high load carrying capacities, low inertia, easy control and strong stiffness (22) therefore for these reasons they are subject of interest for several applications. With a view to ensuring a more immersive simulation VR is used to be integrated with the platform. Even if less sophisticated approaches are limited to offer a first person experience through VR headset and motion controllers (23). The Stewart Platform (SP) is the widely used in applications such as military (24) or driving (25) training and depending on the dynamic system the VR architecture is designed. Cheng-Tang *et Al.* (26) proposed a multi-axis platform integrated with a VR equipment and one joystick controller directly connected to the platform and simulating the yaw-pitch-roll of the model.
As introduced before, teleoperation implies the lack of sense perception especially in a VR environments and therefore the risk of collision with obstacle is high. Even if simulators provides visual information, research focused long on how to improve *proprioception* and the control of self-motion (27). J.J. Gibson (28) introduced the concept of *optic flow*, a visual cue outcome of motion expe-

rience. In the simulation software the movement of the textured images of all objects in the scene provides this sort of effect ensuring greater fidelity. Moreover the use of head mounted display (HDM) offers stereoscopic viewing and head movement tracking.

2.3.1 Stewart Platform

In a motion simulator, a robotic manipulator's sole purpose is to move the pilot based on the output of a motion cueing algorithm.
The robotic manipulator, called SP7, it is a parallel robot with 7 DOF. In addition to a yaw axis actuator, the mechanism has a similar design to the 6RSS-R parallel manipulator, also known as the *Stewart Platform* (17).
Since it's a custom platform located at the university of Genoa, PMAR laboratory team members have developed mechanical design, kinematics, control strategies, workspace analysis, actuator selection process, and so on.

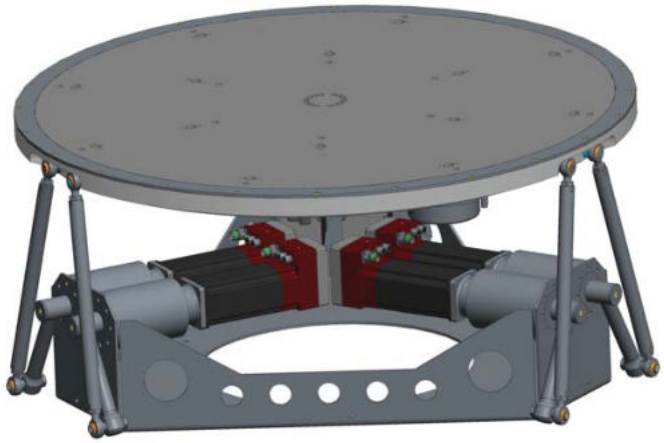


Figure 2.3: SP7 CAD Model

The platform's workspace must remain continuous in all configurations in order to guarantee a smooth motion perception experience for the user. The manipulator can reach several poses among a prescribed workspace, and also the desired configuration avoids passing through singularities. It is necessary to accomplish this even though it severely limits the boundaries of a motion simulator platform (16).
Sharma *et Al.* presented the SP7 workspace computed within the software Maple

(2) (29), which is shown in figure 2.4, while table 4.3 presents all the platform boundaries limit.

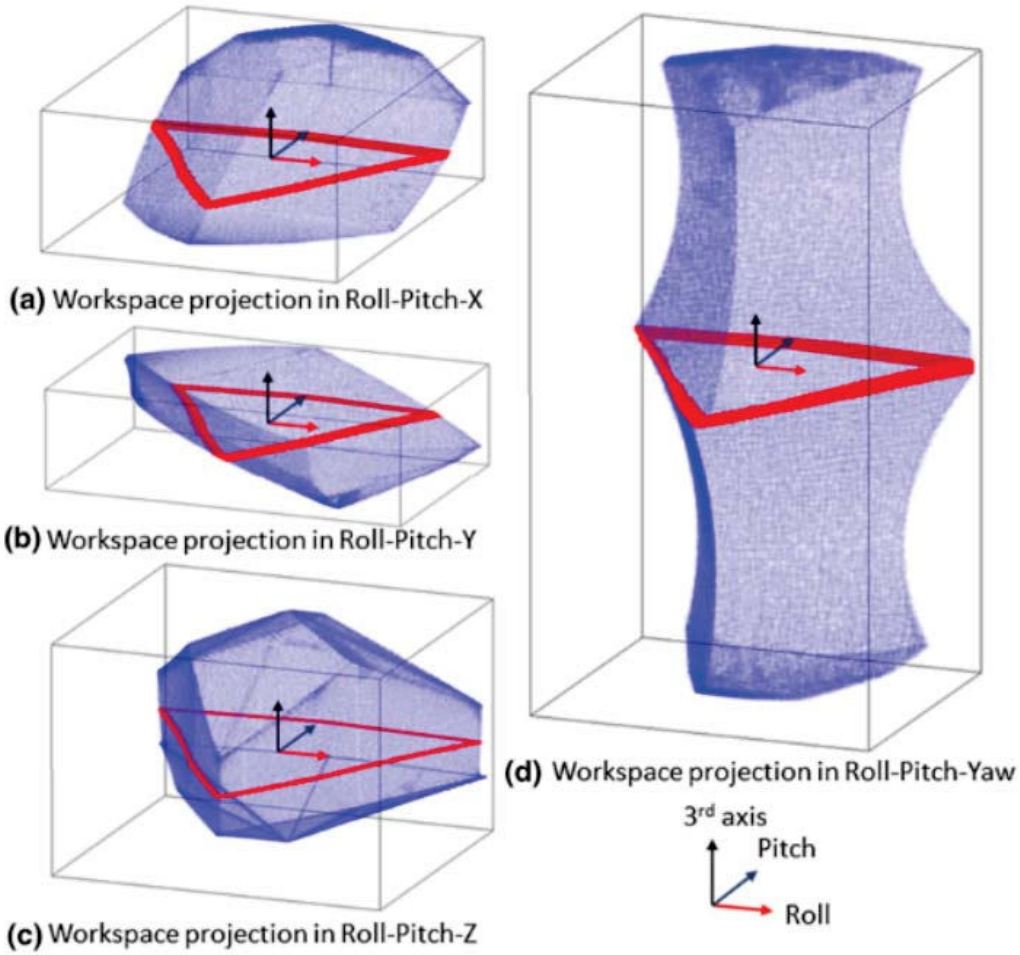


Figure 2.4: SP7 Workspace (2)

In order to validate the SP7 tracking performances, PMAR members (2) performed a simple test using predefined trajectories. HTC Vive trackers (30) recorded the platform trajectory in order to compare it with the desired one. As result, the average precision was less than 10 millimeters while for the high-speed performances the accuracy increased but it was still contained under the 30 millimeters.
Below, figure 2.5 includes some reference trajectories employed for the test.

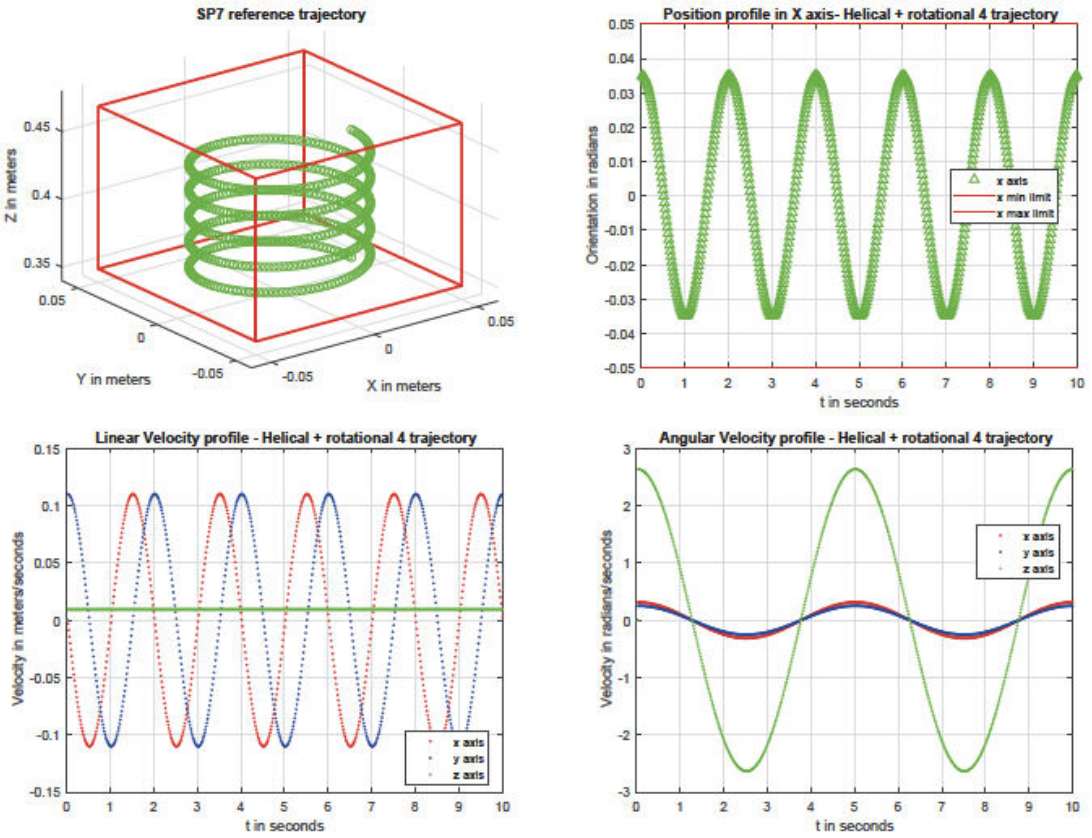


Figure 2.5: SP7 Trajectories (2)

2.4 Haptic and Motion Feedback

During the teleoperation, cameras, as well as HMD, have a limited field of view (FOV) therefore pilots may be unaware of obstacle and this increase the risk of collisions. A leading solution involves the usage of additional information, such as an input from an haptic system. This is a tactile feedback technology which recreates the sense of touch by applying forces, vibrations, or motions to the user (31). Lee *et Al.* (32) measured the utility of force feedback in a mobile robot teleoperation system for safe navigation. This implementation lets the operator to perceiving the presence of near obstacles through other senses: e.g. vibration on the joystick used to remote control the mobile robot. Accordingly, researchers focused on algorithms to generate force feedback able to advertise the operator about forthcoming collision. Additional attention must be payed since the force involves magnitude and direction, so wrong input may affect the teleoperation (33).

The literature deeply investigated into the application of artificial force field (AFF) for the feedback, starting from the autonomous path planning algorithms (34), (35), (36). Regarding the full autonomous navigation, the AFF needs to face up with some problems such as narrow paths and the subsequent risk of the local minima that may cause blocking. Although the research overcame this issue by proposing several solutions (37) (38) or even looking for other algorithms, in the haptic feedback applications this is not a problem since it is the human operator that directly controls the robot and the force field represents just an instruction to follow and it is not blocking.

L. Binet *et Al.* proposed efficient haptic feedback through active (motorized) joystick for the purpose of providing efficient haptic feedback to the pilot when a rotary-wing aircraft is near visible and known obstacles, such as during an emergency avoidance procedure or when navigating in congested areas (3).

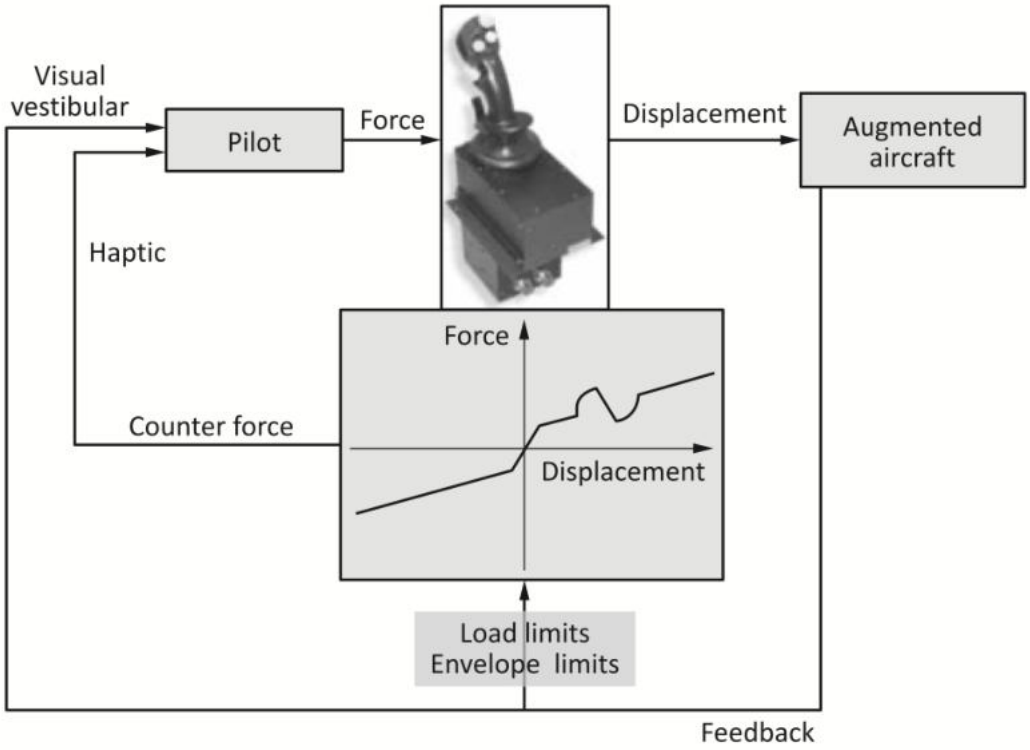


Figure 2.6: Active Joystick in the Pilot Control Loop (3)

2.4.1 Motion Cue Algorithm

Motion cueing describes the projection of the real-world motion data from the virtual reality application to the restricted capabilities of the robotic manipulator with the purpose of ensuring a sense of realism. Thus this is the aim of the motion cue algorithm [MCA].
Regarding the overall architecture, UE4 runs the VR application and it sends visual signals to the HMD (HTC Vive Pro). At the same time, the application feeds the SP7 with the required data in order to perform the motion feedback.
Data are firstly handled by the MCA controller, which is shown in figure 2.7. The controller contains the MCA algorithm implemented in C++ by (4) within a Linux environment. Hence data are processed before being transmitted to the SP7. With the inverse kinematics, implemented in C++, the SP7 controller calculates joint angles by receiving the Cartesian pose and velocity of the end effector. Then PID controllers and PWM are utilized to control the actuators by

processing the joint angles. Ethernet and UDP connections allow all data transmission between the three networks to be managed efficiently with an Ethernet switch.

Figure 2.7: MCA Hardware Connection

Figure 2.8 reports the MCA overall architecture implemented by (4).

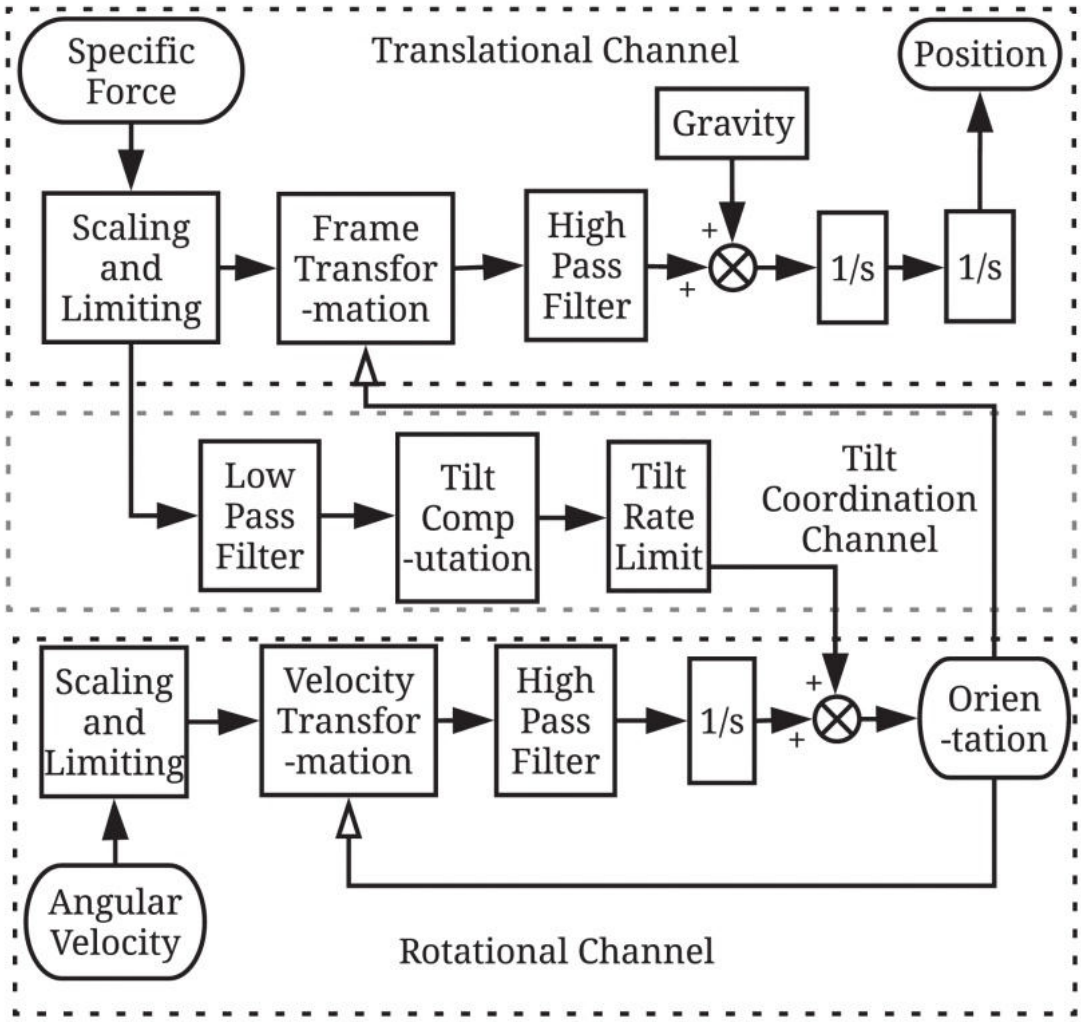


Figure 2.8: MCA Block Diagram (4)

There are four main functional units in classical MCA (39), which are related to the following parameters: scaling, saturation, filtering and limiting.

- **Scaling**: refers to the motion intensity
- **Saturation**: refers to the smoothness of the motion
- **Filtering**: refers to the veracity of the perceived motion
- **Limiting**: refers to the platform boundary limits

So, once the connection between the platform and the simulation has been established, the tuning of the aforementioned parameters allowed to increase the motion feedback reliability.
Plots below show the comparison between the drone pitch (from the UE4) and the SP7 pitch, during a simulation. The values reached by the platform comes from the MCA.

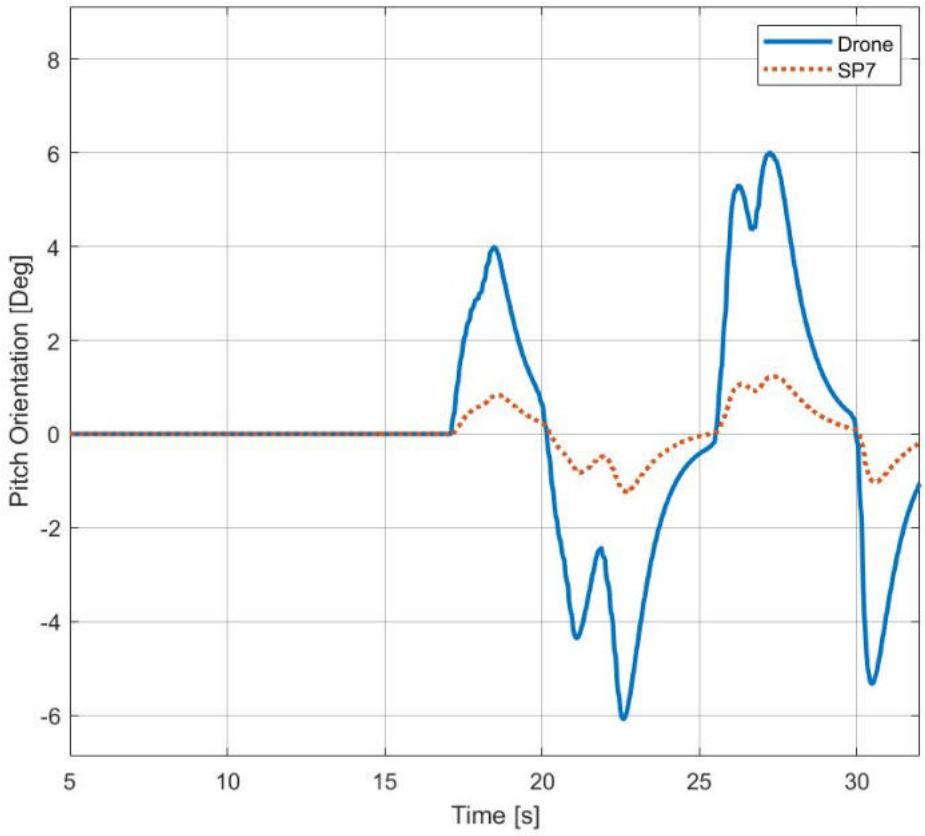


Figure 2.9: Pitch Position Comparison: Drone and SP7

Part II

Section II

Implementation

3.1 Unreal Scenario Design

This section presents the development of the virtual world, in which the user is immersed during the virtual reality experience. As tool, UE4 provides the Landscape Editor which allows to create huge terrain-based worlds with a range of powerful terrain editing tools. The Landscape tool allows you to create immersive outdoor terrain pieces that are optimised and still maintain playable frame rates on a variety of different devices, included HMD (40). Moreover, due to the UE marketplace it is also possible to import custom environment, or even very realistic mesh to add to the desired level.
The SteamVR plugin in the UE transmits video and audio data to the VR device connected to the system and then, inside the application, motion data are sent to MCA controller via an UDP socket. Here, timestamp, visual and motion cue are synchronized within a frequency of 60 Hz.

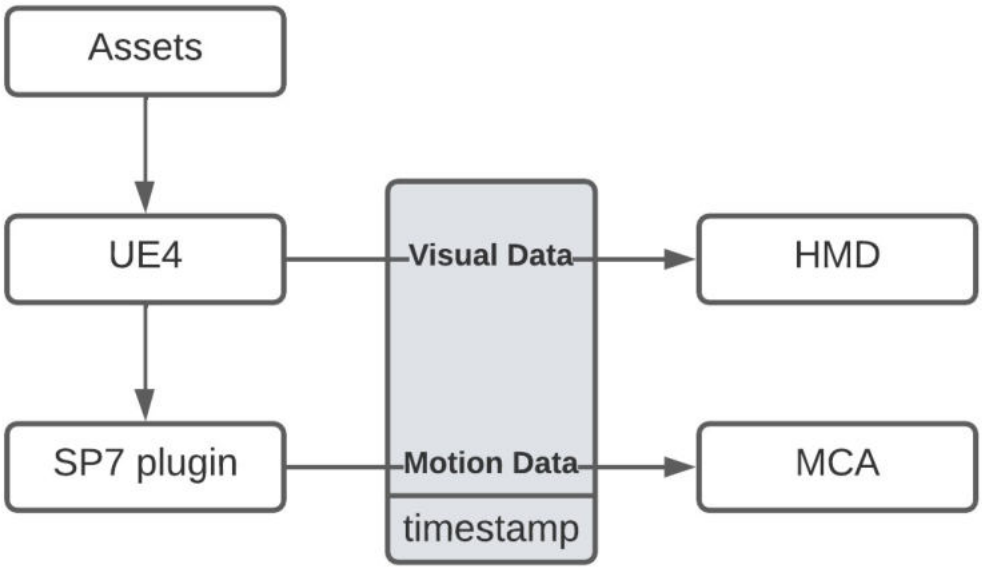


Figure 3.1: Overall Framework Flow Diagram

Even if UE4 provides a very realistic rendering, the environment presents only the necessary assets in order to avoid user distraction and let him focusing on the motion feedback.
Figure 3.2 shows the design of the desired path implemented in the UE environment. Here the pilot is supposed of piloting the drone to reach end point and once that area is reached, the drone triggers an event which quits the simulation. The experience is performed for each test configuration, as described in section 4.5.
Moreover, in order to provide a reliable LiDAR response, obstacles present a complex collision, that will be deeper investigated in section 3.1.1.
For the training phase, instead, a simple environment has been loaded from the marketplace.

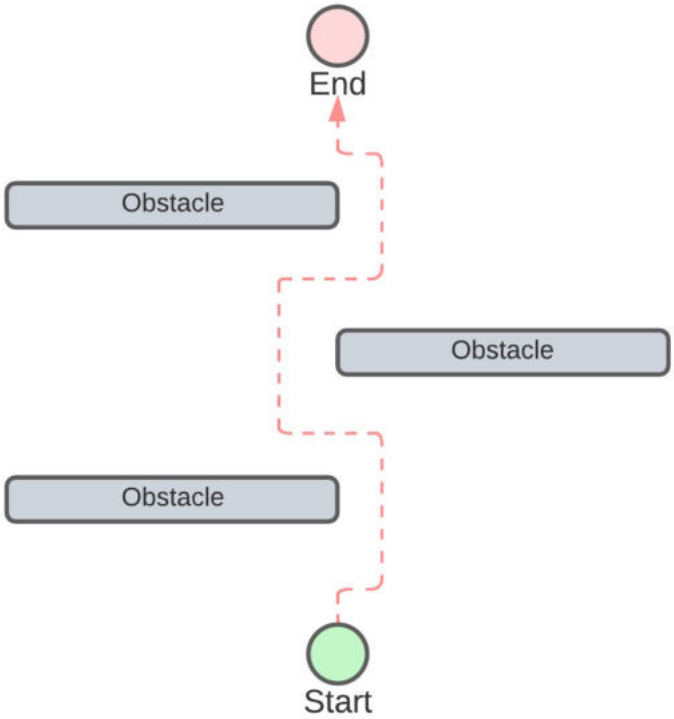


Figure 3.2: Designed Path with Obstacles

Before actually start the real test, HMD needed to be properly set up and the visual cues had to be tuned in order to not annoy the user experience.
In details, once the simulation is initialized, if the HMD is enabled then the main drone mesh disappear so the pilot FOV is not occluded. This is done by the BP node `SetVisibility` which turns off the visibility of the drone body mesh. Then, the node `SetTrackingOrigin` places the VR camera attached to the center of the drone.

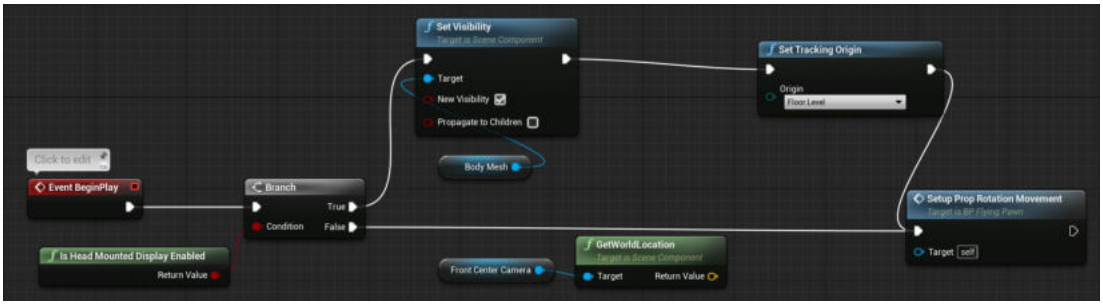


Figure 3.3: Main Drone BP

3.1.1 Collision Filtering

There are several responses to collision, e.g., how to outline the difference between simple and complex collision geometry:

- **Blocking**
- **Overlapping**
- **Ignoring**

For instance a brick wall will *Block* a player, but a trigger will *Overlap* them, allowing them to pass through. Both generate an event (*Hit* or *Overlap* respectively, in UE4 terminology) but it is an important difference. Other objects may *Ignore* the collision altogether.
Imagine if two *Trace Channels* are defined in the simulation, one for *Visibility* and one for *Weapon* queries. A brick wall is set up to block both, a shrub blocks visibility but not weapons, and bulletproof glass blocks weapons but not visibility. So in this way a single trace channel is specified (5).

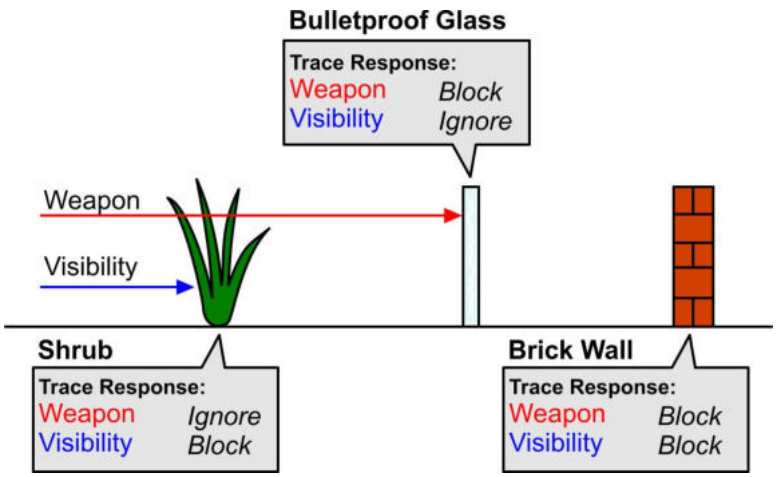


Figure 3.4: Examples of Collision Trace Responses (5)

Things get a bit more complicated when the simulation handles collisions between moving objects, because there can be several combinations. In UE4 each object knows its own *Object Channel*, plus a list of how it responds to other ones. When two objects intersect, the implementation looks at how they respond to each other, and take the least blocking interaction, as shown in the following figure:

Object B \ Object A	Ignore	Overlap	Block
Ignore	Ignore	Ignore	Ignore
Overlap	Ignore	Overlap	Overlap
Block	Ignore	Overlap	Block

Figure 3.5: Collision Table (5) (5)

So, referring to the figure above, the ghost drone (see 4.4) will have its response to the Pawn channel set to **Ignore**, therefore it can move freely thought the environment without corrupting the motion feedback.
Moreover, regarding the collision, each object in UE4 can have both a *complex* and *simple* collision representation. The first one refers to using the actual rendering geometry for collision, thus providing a more realistic behaviour. This is not always required, and so each mesh can also have a **Simple Collision** representation consisting of a collection of spheres, boxes, capsules and convex hulls.
In order to provide a better reliability for the LiDAR scan response, all the obstacle in the environment presents a **complex collision** as shown in the following figure:

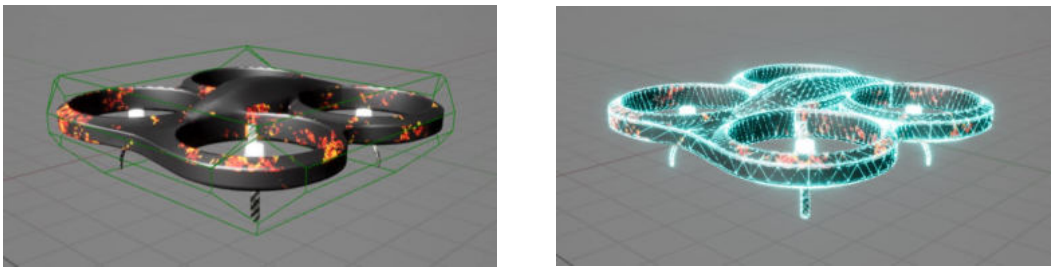


Figure 3.6: Simple and Complex Collision Comparison

3.2 AirSim Drone

At the beginning of the simulation, AirSim looks for the setting configuration that can be specified by the `--settings` command argument. For instance, regarding the Window OS:

```
AirSim.exe --settings 'C:\path\to\settings.json'
```

Otherwise, it searches for the `setting.json` located in the same directory where the simulation executable is launched.
First of all, the configuration file allows to select the simulation mode among two possible solutions: **multirotor** or **car**.
In the AirSim Git documentation (1) there is a complete list of settings available along with their default values, while in the appendix A the current one is shown.
As explained before, regarding the third test, a new drone has been configured. The parameter `Pawnpath` allows to specify the own vehicle pawn blueprint. The BP is located inside the *Content* folder.
In details a new Flying Pawn BP has been imported in order add the ghost one. As explained in section 3.1.1, this Actor has no collision enabled therefore and moreover the mesh is translucent in order to visualize its behaviour during the debug phase.

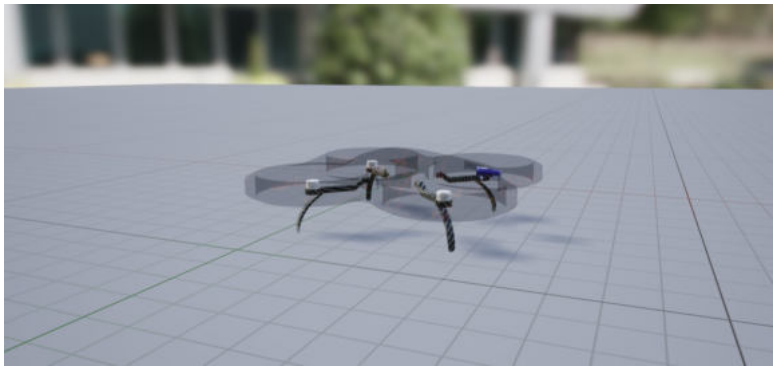


Figure 3.7: Ghost Drone UE Model

3.2.1 Remote Control

In order to fly manually, remote control need to be configured. AirSim, by default, uses the so called mode *simple flight* as described in section **??**, but for this

simulation a new setup has been implemented. As shown in figure 3.8 it consists of:

(a) Thrustmaster replica JOYSTICK

(b) Thrustmaster Dual Replica THROTTLES

(c) MFG CROSSWIND rudder pedals

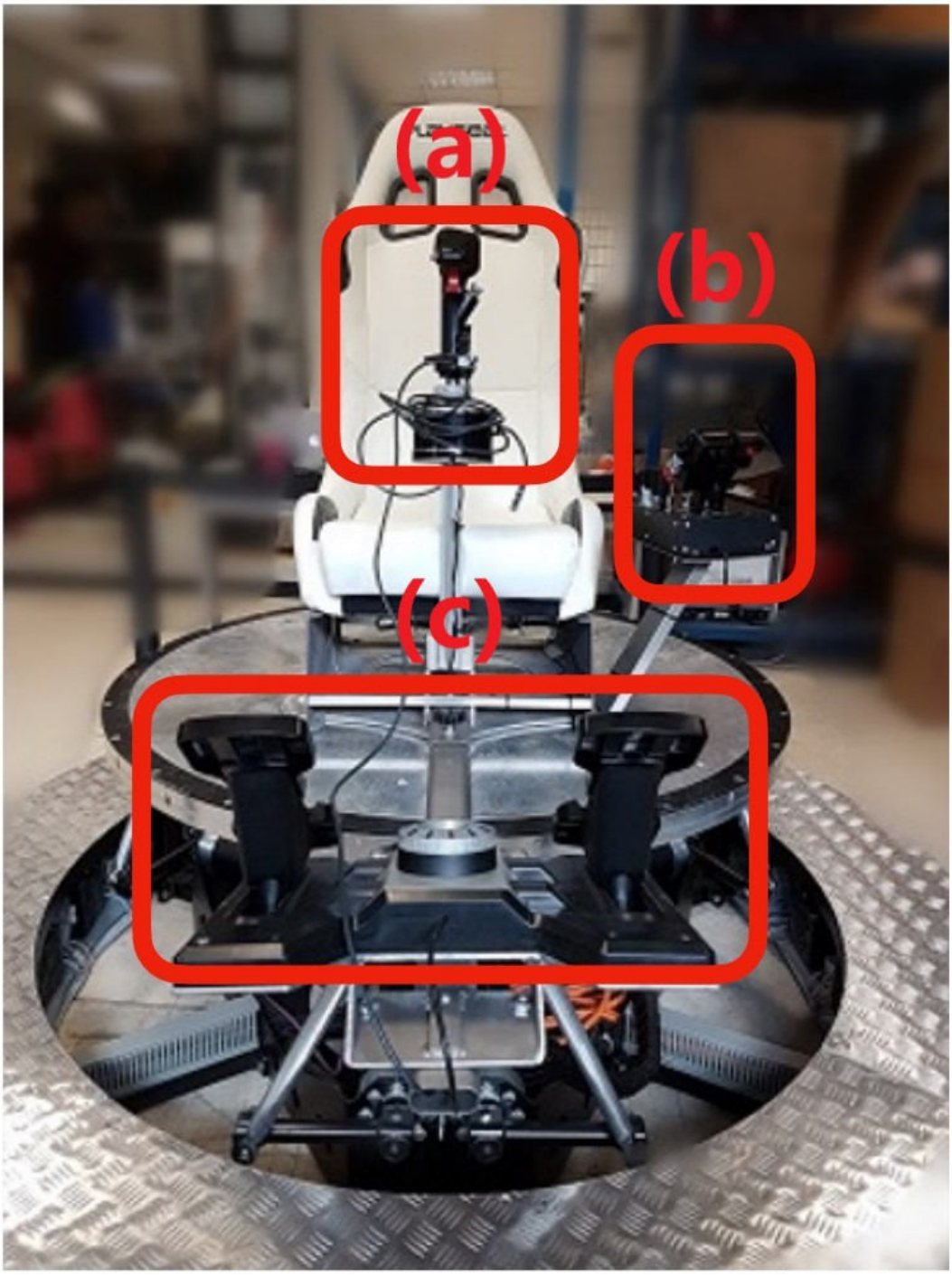


Figure 3.8: Remote Control Final Setup

Regarding the controller input values reading, the Python library **PyGame** (41)

allows to interface within external devices. A Python implementation gives the possibility to call the following API method, which allows to move the drone with a desired velocity expressed with respect to the drone inertial frame:

```
moveByVelocityBodyFrameAsync(vx, vy, vz, duration, drivetrain=0,
                    yaw_mode=<YawMode> {'is_rate': True, 'yaw_or_rate':
                        0.0},
                    vehicle_name='')
```

where:

- **vx**: desired velocity in the X axis of the vehicle's local NED frame
- **vy**: desired velocity in the y axis of the vehicle's local NED frame
- **vz**: desired velocity in the z axis of the vehicle's local NED frame
- **duration** `float`: desired amount of time (seconds), to send this command for
- **drivetrain**: if keeping front pointing ahead or not
- **yaw_mode**: desired angle rotation [deg]
- **vehicle_name**: name of the multirotor to send this command to

First of all the script initializes the controller modules then all the joysticks themselves. The process goes through the accounts of the joysticks that PyGame sees and initializes them with different indexes. The indexes depend on the order in which they are loaded in the operating system. The variable joysticks is a list off all the joystick object you can interface with. Then the following event types will be generated by the joysticks that refer to the figure 3.9:

- JOYAXISMOTION: input for the v_x and the v_y components
- JOYHATMOTION: input for the v_z component
- JOYBUTTONDOWN: input for restart the drone position

Regarding the *restart* button, after the take off, the current pose of the drone is stored, in order to re-spawn the vehicle if desired. So each frame rate, due to

the PyGame method, it is possible to access the aforementioned input values and therefore move the drone as desired.

Figure 3.9: Joystick Configuration

The figure 3.10 shows how the values related to the axes and the hat are read. In details the hat returns two values representing its x and y position where (0, 0) means centered. A value of -1 means left/down and a value of 1 means right/up: so (-1, 0) means left; (1, 0) means right; (0, 1) means up; (1, 1) means upper-right ans so on. This implementation takes into account only the y values: if the value is positive the drone goes up, otherwise it goes down.
The axes values instead range from -1 to 1 with a value of 0 being centered. Some tolerance may be taken into account to handle jitter, and joystick drift may keep the joystick from centering at 0 or using the full range of position values. Therefore a *dead zone* prevents this unwanted scenario.

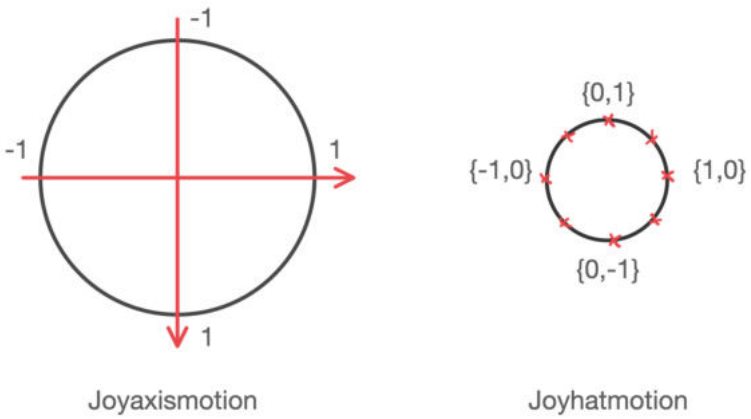


Figure 3.10: Joystick Input Graphs

The throttle is used to control the current velocity, from zero to the max velocity defined during the initialization as follow $V' = T \cdot V$, where V refers to the Joystick input values, while T is a scalar value computed in the following way:

$$T = a + \frac{(x - x_{min})(b - a)}{(x_{max} - x_{min})} \tag{3.1}$$

Where:

- a: 0, the NULL velocity in the simulation. So if the level is put down the drone cannot move
- b = MAX VEL: the maximum velocity set. This parameter is defined inside the `__init__` method of the related python class
- x_{max}: the value that is read if the level is pulled up
- x_{min}: the value that is read if the level is pulled down

Finally, regarding the argument `yaw_mode` introduced before, the MFG crosswind component allows to control the desired drone rotation among its z axis.

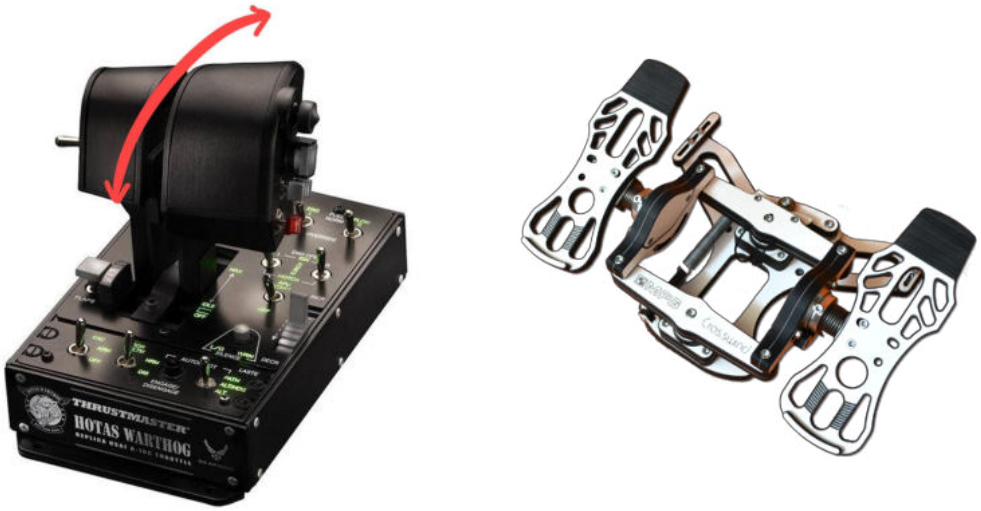


Figure 3.11: Throttle Configuration and MFG Crosswind

3.2.2 Sensor Configuration

AirSim, along with other sensors, supports LiDAR for the multirotor. In order to enable it, the `SensorType` and `Enabled` attributes must be set in the `settings.json` file as follow:

```
"<LidarName>": {
    "SensorType": 6,
    "Enabled" : true,
}
```

Please note that multiple LiDAR can be added to the same vehicle.
The following parameters can be configured via settings JSON in order to design properly the sensor configuration:

Parameter	**Description**
NumberOfChannels	Number of channels/lasers of the LiDAR
Range	Range, in meters
PointsPerSecond	Number of points captured per second
RotationsPerSecond	Rotations per second
HorizontalFOVStart	Horizontal FOV start for the lidar
HorizontalFOVEnd	Horizontal FOV end for the lidar
VerticalFOVUpper	Vertical FOV upper limit for the lidar
VerticalFOVLower	Vertical FOV lower limit for the lidar
X Y Z	Position of the lidar relative to the vehicle
Roll Pitch Yaw	Orientation of the lidar relative to the vehicle
DataFrame	Frame for the points in output

Table 3.1: LiDAR JSON Parameters

From the server side visualization, LiDAR points can be shown in the viewport. In details, due to flag `DrawDebugPoints` set equal to true it is possible to enable the drawing of hit laser points on the display window as shown in the figure 3.12.

Figure 3.12: LiDAR Debug Points Screen Capture

From the client side, instead, the Python API method `getLidarData()` retrieves the LiDAR data.

```
class LidarData(MsgpackMixin):
    point_cloud = 0.0
    time_stamp = np.uint64(0)
    pose = Pose()
    segmentation = 0
```

So the API returns a Point-Cloud as a flat array of floats along with the timestamp of the capture and LiDAR pose, where in the `point_cloud`:

- The floats represent $[x, y, z]$ coordinate for each point **hit** within the range in the last scan
- The frame for the points in the output is configurable using `DataFrame` attribute (see table 3.1):
  - `VehicleInertialFrame`: returned points are in vehicle inertial frame
  - `SensorLocalFrame`: returned points are in LiDAR local frame

The following figure shows how LiDAR data are managed every thick:

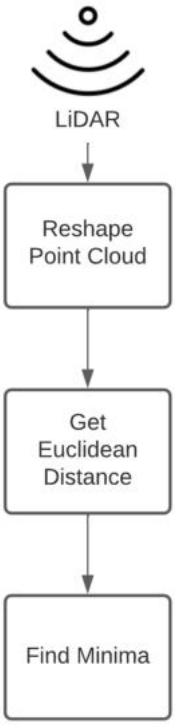


Figure 3.13: LiDAR Data Process Flow Chart

```
def reshape_point_cloud(point_cloud):
        # create a matrix where each row represent
        # the points [x, y, z] that were hit by the LiDAR
        points = np.array(point_cloud, dtype=np.dtype('f4'))
        points = np.reshape(points, (int(points.shape[0] / 3), 3))
        return points
```

Since the Point Cloud is stored in a list, with the static method `reshape_point_cloud` LiDAR data are arranged inside a matrix $\mathbf{M} \in \mathbb{R}^{n \times 4}$.

$$\mathbf{M} = \begin{bmatrix} x_1 & y_1 & z_1 & d_1 \\ \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & z_n & d_n \end{bmatrix} \tag{3.2}$$

Where:

- n is the total number of the scanned points within the chosen range
- the rows represents the related x, y, z with the Euclidean distance d with respect to the vehicle inertial frame

Once the data are properly collected, the Euclidean distance between the sensor frame and the object is calculated. The function returns the minimum distance, if it is below a certain threshold (defined inside the LiDAR class `__init__` method) the obstacle avoidance algorithm starts as described in the next section.

3.3 Force Field

3.3.1 2D Implementation

In order to be confidential with LiDAR data management and the FF implementation, a two-dimensional approach has been implemented. In details, regarding the sensor configuration (see table 3.1), the vertical field of view has been set to zero, while the horizontal field of view ranges between the 0 rad and π rad. Figure 3.16 shows the LiDAR frame, within the distance vector which is pointing to the closest point detected.

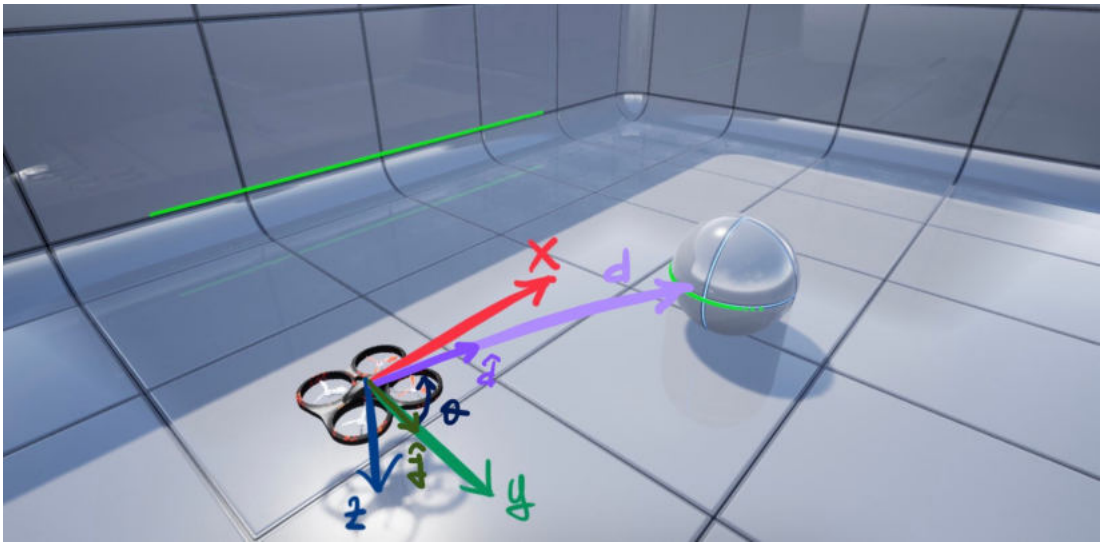


Figure 3.14: Simulation Screen Capture LiDAR 2D

The LiDAR FOV has been subdivided into 5 regions, therefore once you know in which region the obstacle belongs, the drone moves away accordingly.
So, after establishing the AirSim connection, the constructor of the class initialize the following variables:

- `region=['E', 'NE', 'N', 'NO', 'O']`: a list that subdivide the LiDAR FOV as follow:

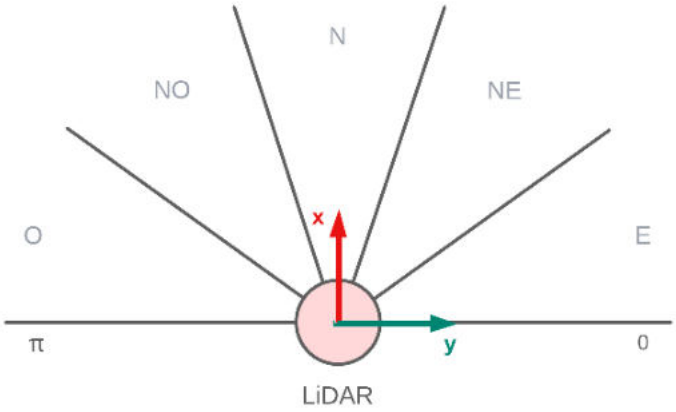


Figure 3.15: LiDAR Regions

- `regions_number = len(self.regions)`: the total number of the region, 5 in this case
- `j = [0, 1]`: represents the $\hat{j}$ versor of the LiDAR frame.

Then if the Euclidean distance is below a certain threshold, the angle between the unit vector $\hat{d} = \frac{d}{|d|}$ and $\hat{j}$ is calculated in order to estimate in which region the obstacle has been detected.

$$\theta = arccos(\hat{d} \cdot \hat{j}) \tag{3.3}$$

Then:

$$\text{region} = \text{floor}(\frac{\theta}{5}) \begin{cases} if \ = \ 0 \rightarrow \ \text{region} = E \\ if \ = \ 1 \rightarrow \ \text{region} = NE \\ if \ = \ 2 \rightarrow \ \text{region} = N \\ if \ = \ 3 \rightarrow \ \text{region} = NO \\ if \ = \ 4 \rightarrow \ \text{region} = O \end{cases} \tag{3.4}$$

So depending on the output of the aforementioned function, the drone is moved away in the desired direction by using the API method `moveByVelocityZBodyFrameAsync`. The desired velocity switches among:

- $V_E = [0, \ -A, \ 0]$ if the obstacle is in the region E
- $V_{NE} = [-\frac{\sqrt{2}A}{2}, \ \frac{\sqrt{2}A}{2}, \ 0]$ if the obstacle is in the region NE

- $V_N = [-A,\ 0,\ 0]$ if the obstacle is in the region N
- $V_{NO} = [-\frac{\sqrt{2}A}{2},\ \frac{\sqrt{2}A}{2},\ 0]$ if the obstacle is in the region N0
- $V_O = [0,\ A,\ 0]$ if the obstacle is in the region O

Where **A** is deeply investigate in section 3.3.2.

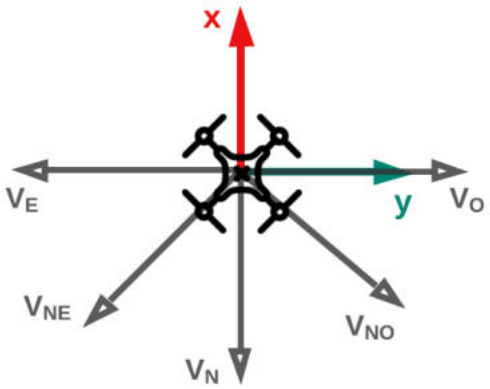


Figure 3.16: Repulsive Velocities

3.3.2 3D Implementation

As mentioned in section 3.2.2, LiDAR class returns the coordinates $[x,\ y,\ z]$, the Euclidean distance of the closest point detected and a flag if the obstacle is below a predefined threshold. Figure 3.17 gives an overview about the execution flow, performed at each tick.

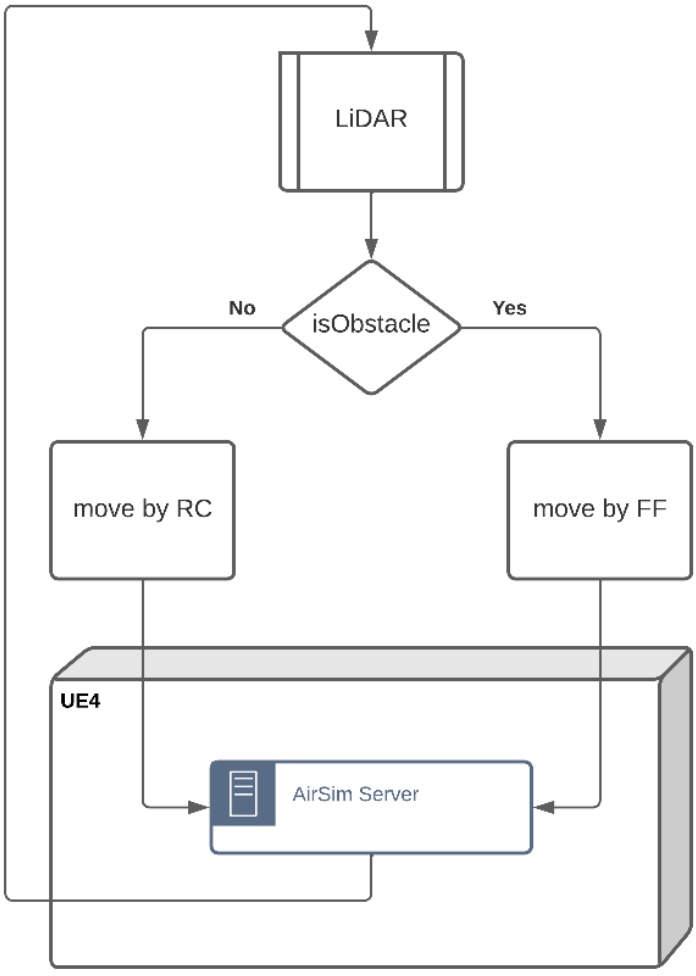


Figure 3.17: FF Overall Architecture

If the flag is false, the drone is moved by the remote control as explained in section 3.2.1. Otherwise, in presence of an obstacle, the FF is activated.
Figure 3.17 shows the FF class behaviour.

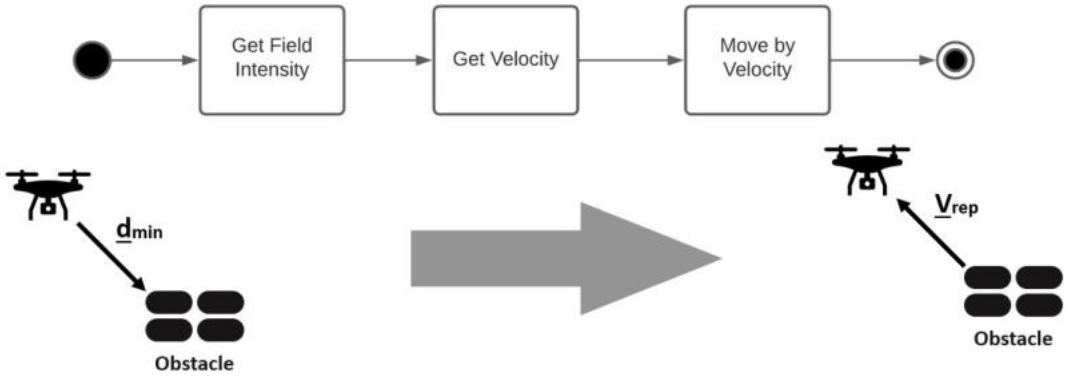


Figure 3.18: FF Flow

The field intensity strictly depends on the current distance between the drone and the obstacle. It is computed as follows:

$$\mathbf{A} = \begin{cases} A_{max}, & \text{if } d \leq d_{min}, \\ A(d), & \text{if } d_{min} < d < d_{max} \\ A_{min}, & \text{if } d \geq d_{max} \end{cases} \tag{3.5}$$

The plot in figure 3.19 explains the Gain A trend. It increases with decreased distance. If d is approximately 0 then A is set equal to the maximum value defined in the `__init__` method of the class.

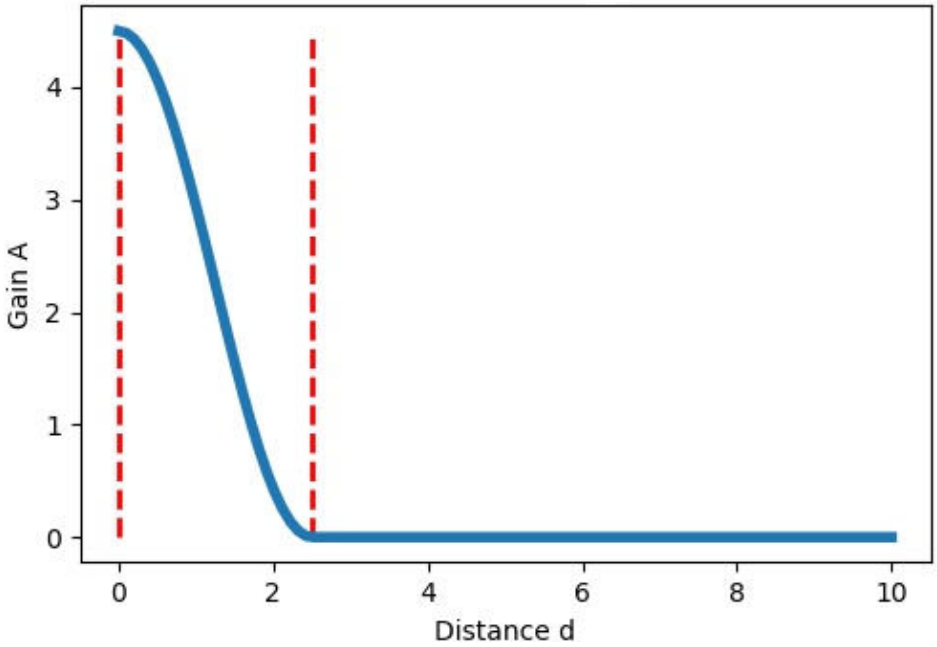


Figure 3.19: Bell Shaped Function

So, given the input points the final velocity will be equal to:

$$\underline{\mathrm{V}}_{rep} = -A \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.6}$$

Before actually calling the API method aimed at moving the drone, the FF classifies the reliability of the repulsive motion. Referring to figure 3.20, supposing that the drone is moving against an obstacle, which is orthogonal to the xy world plane.

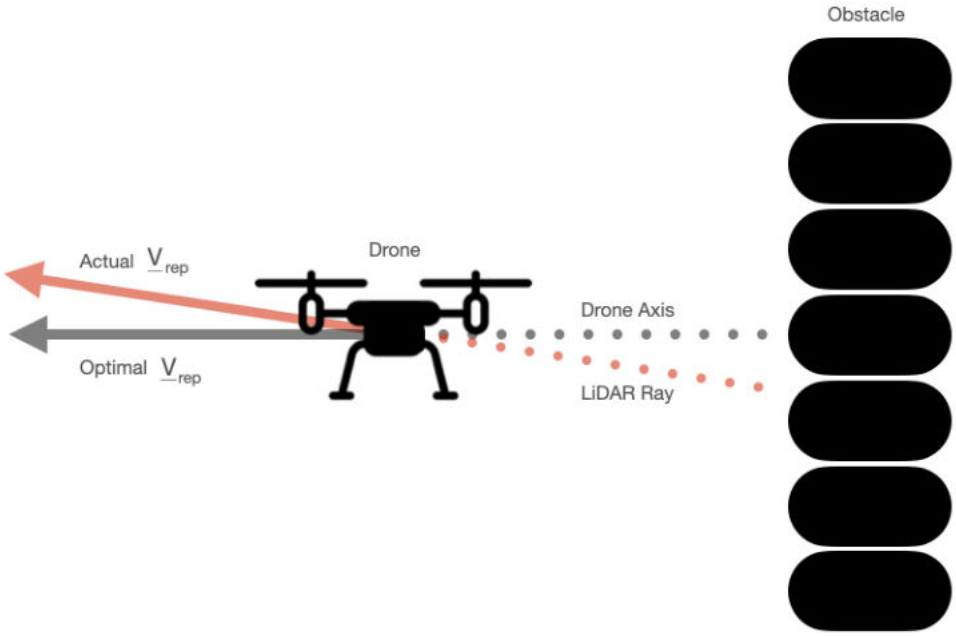


Figure 3.20: FF Scenario Example

If the vector $d = [x,\ y,\ z]$, related to the closest point detected, has one element that is much lower than the others then it is reduced to zero.
Therefore, given the input vector $\underline{d}$:

$$\frac{d_i}{max(\underline{d})} < 0.15 \ \Rightarrow \ d_i = 0 \ for \ i = 1,\ 2,\ 3 \tag{3.7}$$

Moreover, before actually moving the drone, another condition is hold in order to check if the obstacle lies in the xy plane, the drone is forced to move by maintaining the same altitude. This is ensured within the following python method:

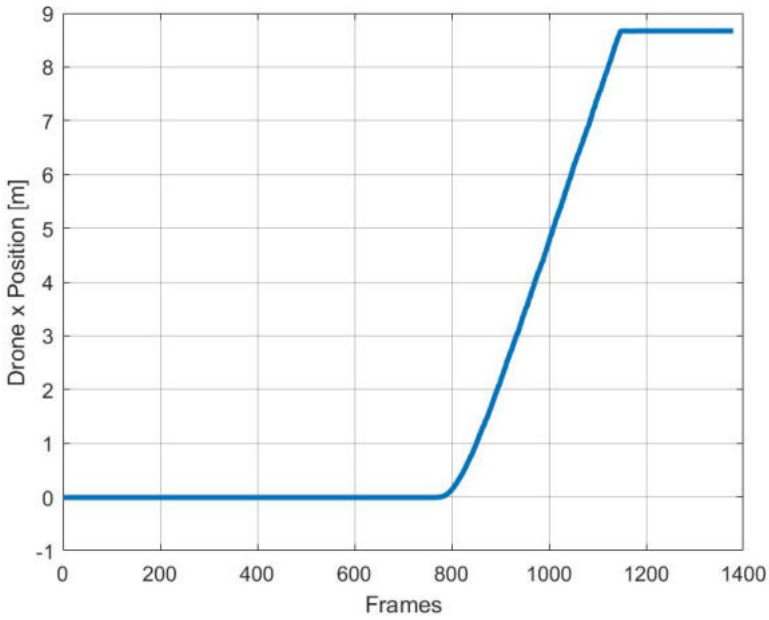
```
self.client.moveByVelocityZBodyFrameAsync(-vel[0].item(),
                                          -vel[1].item(),
                                          current_altitude,
                                          self.duration)
```

where:

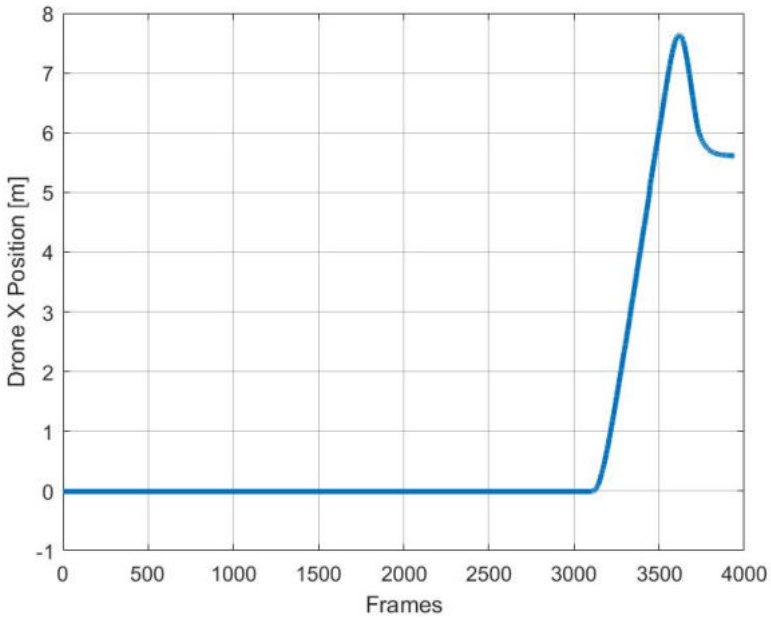
- The vector `vel` refers to equation 3.6
- The argument `current_altitude` is the actual drone altitude

This has been done since from the simulation point of view, this motion is imperceptible, while in the Platform Space this behavior may corrupt the feedback perception. So keeping the same altitude better advertise the pilot about the presence of an obstacle next to him.
If this condition is not satisfied then the method `moveByVelocityBodyFrameAsync` is called and the whole vector computed in 3.6 is passed as argument.

In order to validate the FF algorithm two simple test are conducted, in which the drone moves against the same obstacle, located in front of its (so along the x axis). In can be noticed that once the drone is under the FF effects, the x values decrease, while it's constant when the drone hit the obstacle.



(a) Linear Position x per Frame without FF



(b) Linear Position x per Frame with FF

Figure 3.21: Linear Position x Comparison: without FF and with FF

As explained before, several test are performed therefore the algorithm has been re-adapted in order to send data to the platform. In detail, regarding the **test 2** (where the platforms move only under the FF effect) a flag is send to UE4 in order to open the communication gate (see section 4.5). Concerning the **test 3**, every time an obstacle is detected the ghost drone appears and it is affected by the force field while the user is able to control the main drone. In the previous cases if the obstacle is detected all the user inputs are discarded. In order to spawn properly, the ghost drone inherits the kinematic state of the main one and then it moves under the FF effect. After a certain amount of time, the ghost drone disappears and the API send a flag in order to close the gate and therefore interrupt the platform communication. While for the **test 1** data are sent each tick, hence no flags are needed.

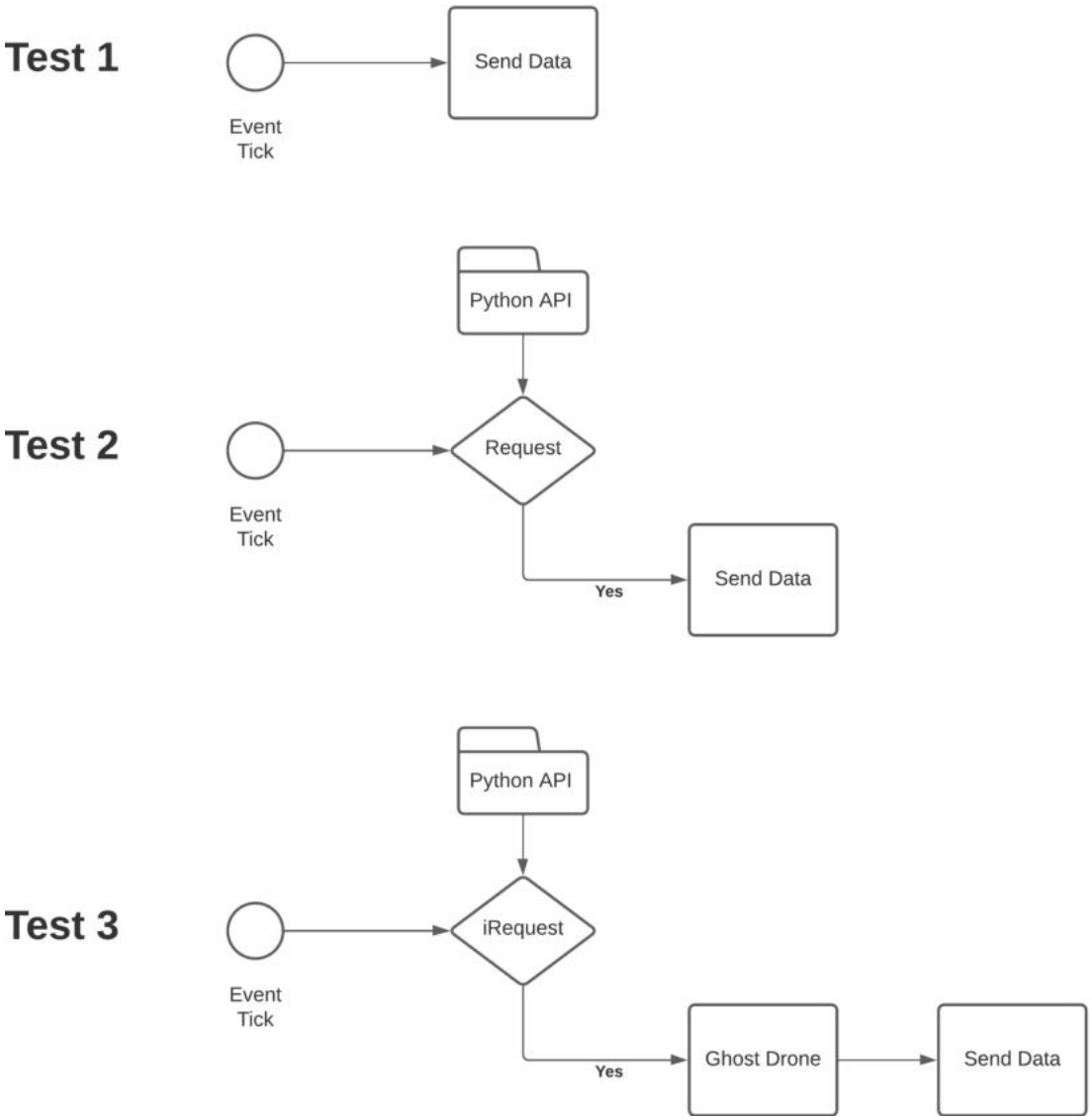


Figure 3.22: Test Flow Overall

Design of Experiments

In order to try out if the motion cue help, or at least add value, to drone piloting three different tests will be performed in addition to two preliminary steps. These allow the user to get familiar with the VR experience. After each session the user remove the HMD and compile the questionnaire. The phases, described in the next sections, are the following:

- Training phase
- Test 1
- Test 2
- Test 3

The following table compares the main tree test. The parameter *visual cue* means a visual correspondence whenever the platform is moving.

NO OBSTACLE			
	Test 1	Test 2	Test 3
Motion Cue	✓	✗	✗
Visual Cue	✓	✗	✗

Table 4.1: Test Comparison in Case of No Obstacle

NO OBSTACLE			
	Test 1	Test 2	Test 3
Motion Cue	✓	✓	✓
Visual Cue	✓	✓	✗

Table 4.2: Test Comparison in Case of Obstacle

Another solution has been developed and it is called *Just Drone*. This mode lets the user navigate inside the scenario, without being affected by motion cue. Even the FF is not activated, the aim of this option is to improve the user's capabilities during the piloting in a more complex scenario, the one used for the tests.

4.1 Training Phase

Explain to the user how to control the drone in a simple learning arena. Here there is no motion cue, instead the user learns how to control the drone through the RC configuration, described in section 3.2.1. So in this scenario the pilot performs easy movements.

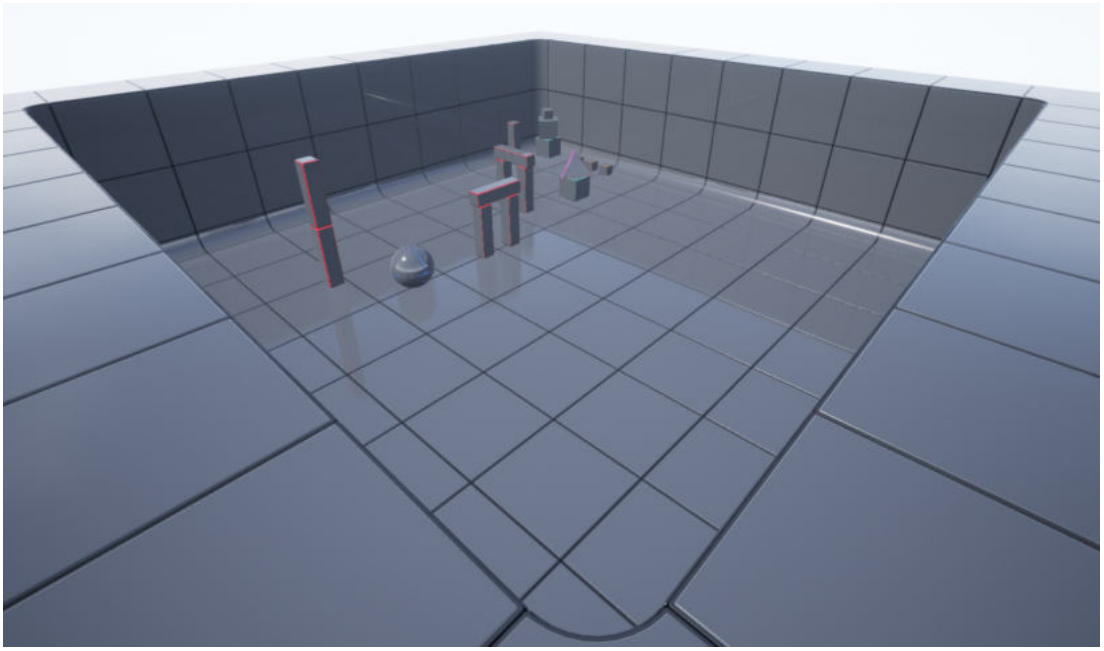


Figure 4.1: Training Scenario

4.2 Test 1

For the next simulation, a new environment has been designed. It is an open space that is bigger than the one described above. The obstacles in the scene (e.g., towers) present convex collision, that guarantee more accurate collision shapes in order to better simulate a real application. During this simulation, the

user can fly over the scene in order to reach the end point. Here the motion feedback is always received.

Figure 4.2: Test Scenario

4.3 Test 2

As in the previous case the drone is affected by the FF and the user input is discarded. However in this case, the platform receives data only when the LiDAR detects obstacle within a certain threshold, so each time the obstacle avoidance algorithm starts. The python API is the one described in section 4.5, where a flag must be send in order to send data from the simulation to the platform.

Figure 4.3: Environment with Obstacle Top View

4.4 Test 3

This is the most challenging scenario. Here only the platform moves under the FF effect and in the meanwhile the user can control the drone. Due to the motion cue, the pilot advertise the presence of a near obstacle and it is own liability to alter manually the course in order to avoid collision. The Python API is the one described in section 3.3. Again, a flag is sent to UE4 in order to start the UDP socket communication. Unlike before data are related to the ghost drone position (see Python API implementation). Its behavior allows to visualize the platform motion in presence of obstacles, therefore it is expected to invite moving away from the obstacle.

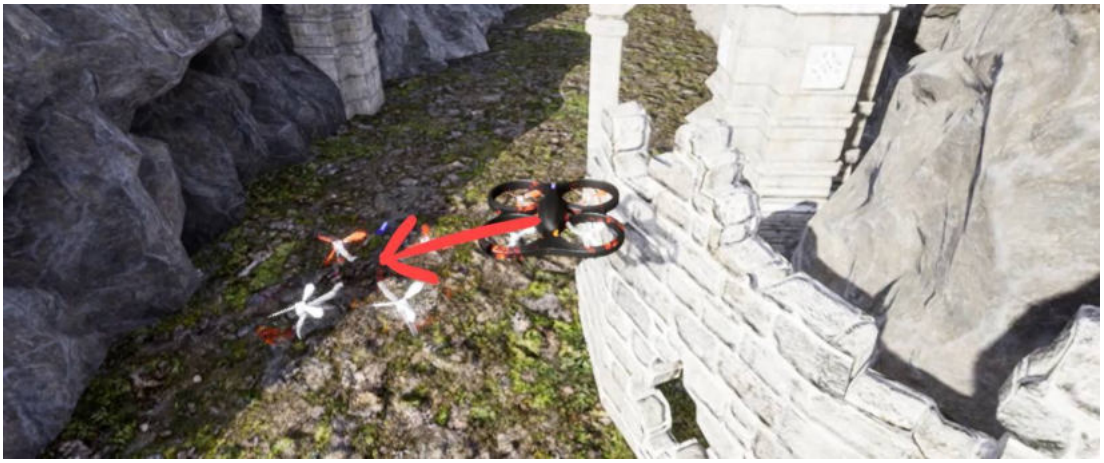


Figure 4.4: Ghost Drone Behaviour Screen Capture

4.5 Experimental Setup Architecture

In this section the simulation flow will be presented, starting from the the launch of the executable to the end of the simulation itself.

4.5.1 Main Menu

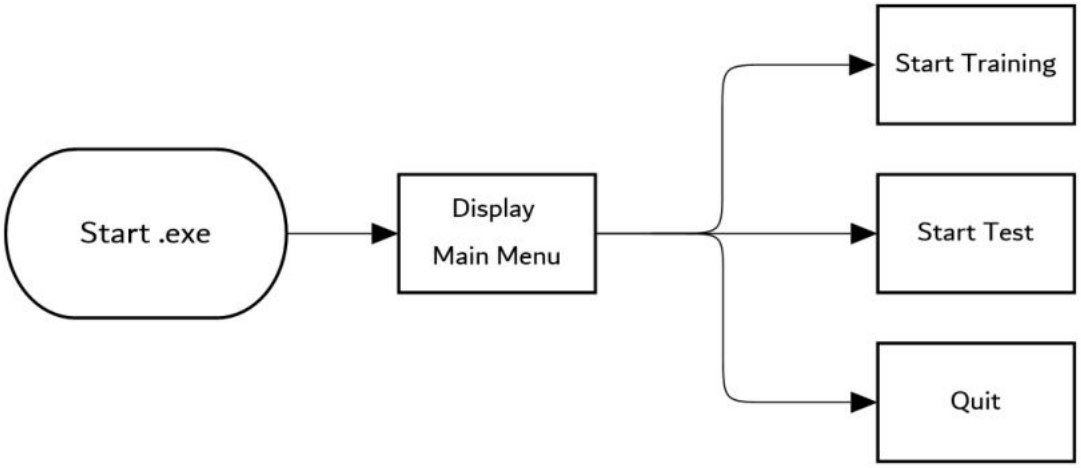


Figure 4.5: Init Flowchart

At the beginning of the simulation, who managed the experiments, is prompt to chose among three initial option:

- Start Training
- Start Test
- Quit

The main menu is a *widget component*, an User Interface (UI) object that allows to interact with the simulation world.

Figure 4.6: Main Menu Screen Capture

Buttons are click-able primitive widget to enable basic interaction, each button, whenever pressed, calls an on clicked-event:

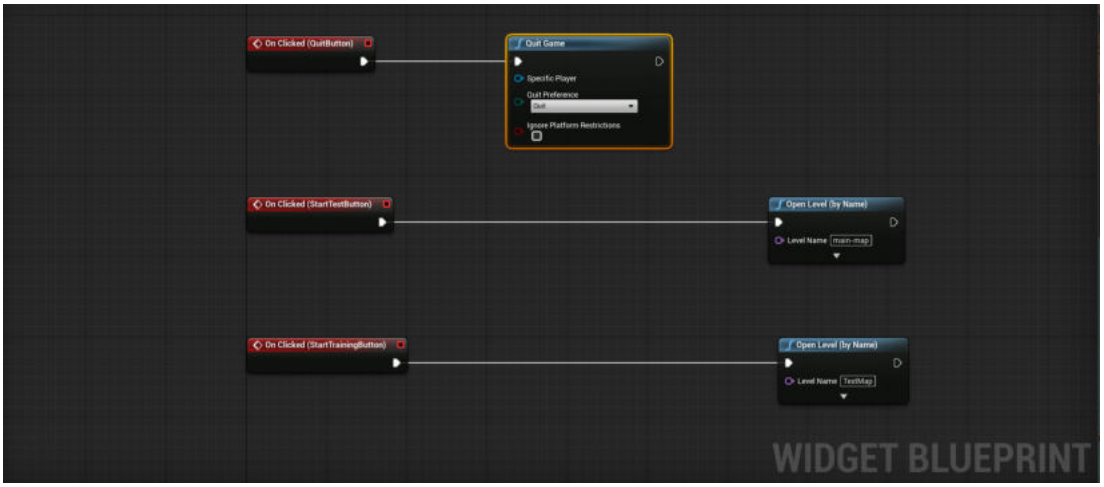


Figure 4.7: Main Menu Widget BP

Depending on the button selected, different action may start:

- **Start Training**: load the environment shown in section 4.1 in order to start the training phase

- **Start Test**: load another menu which allows to select the desired test, as discussed in the next sections.
- **Quit**: exit from the simulation

4.5.2 Start Test

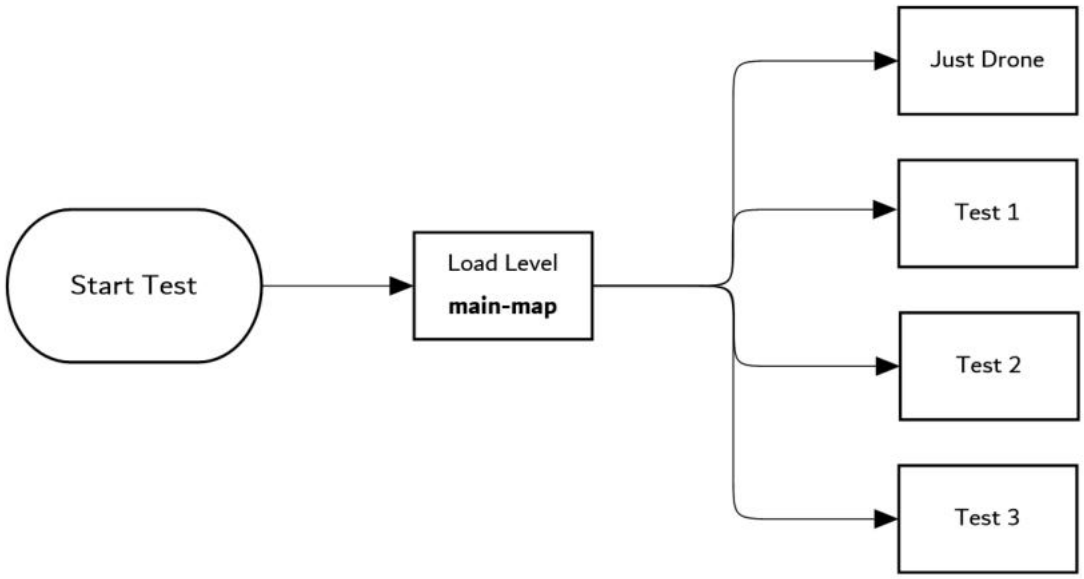


Figure 4.8: Start Test Flowchart

Once user selects the start test button, the Level **main-map** is loaded. In UE4 terms, a Level is made up of a collection of Static Meshes, Volumes, Lights, Blueprints and more all working together to bring the desired simulation experience. Levels in UE4 have the own blueprint that acts as a level-wide global event graph and it can be edited within the Unreal Editor.
The following figure shows the main-map level blueprint implementation:

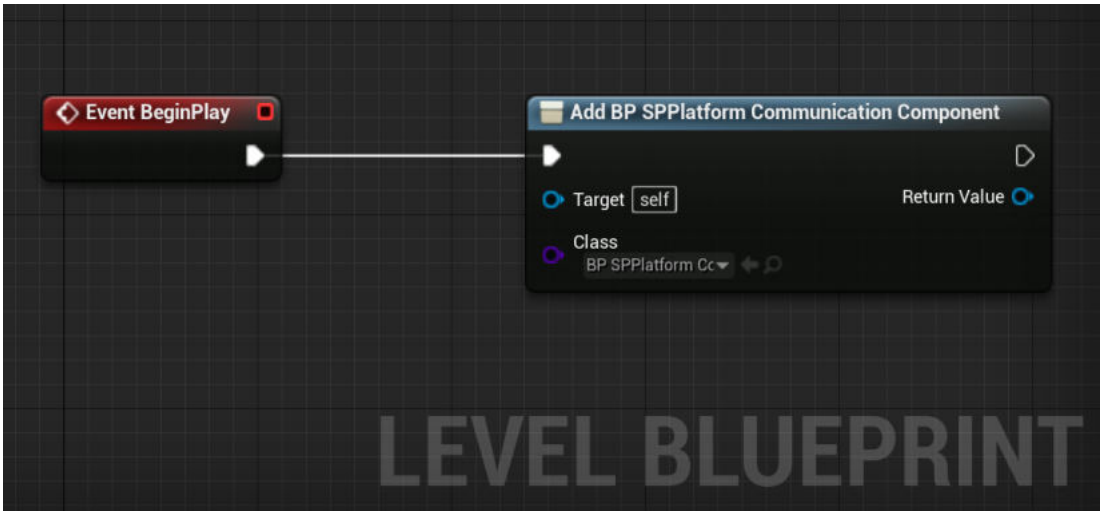


Figure 4.9: Main-Menu Level BP

The event **BeginPlay** is triggered immediately when the level is started, then the add component by class function is called. In UE4, **Components** are a special type of **Object** designed to be used as sub-objects within actors. These are generally used in situations where it is necessary to have easily swappable parts in order to change the behavior or functionality of some particular aspect of the owning Actor. The aforementioned node takes as input the desired object class to be constructed and returns as output the constructed object. In this case is the *SPPlatformCommunicationComponent* which handles the communication between the simulation and the platform.
Once the component is added, it adds to the view-port the following option menu:

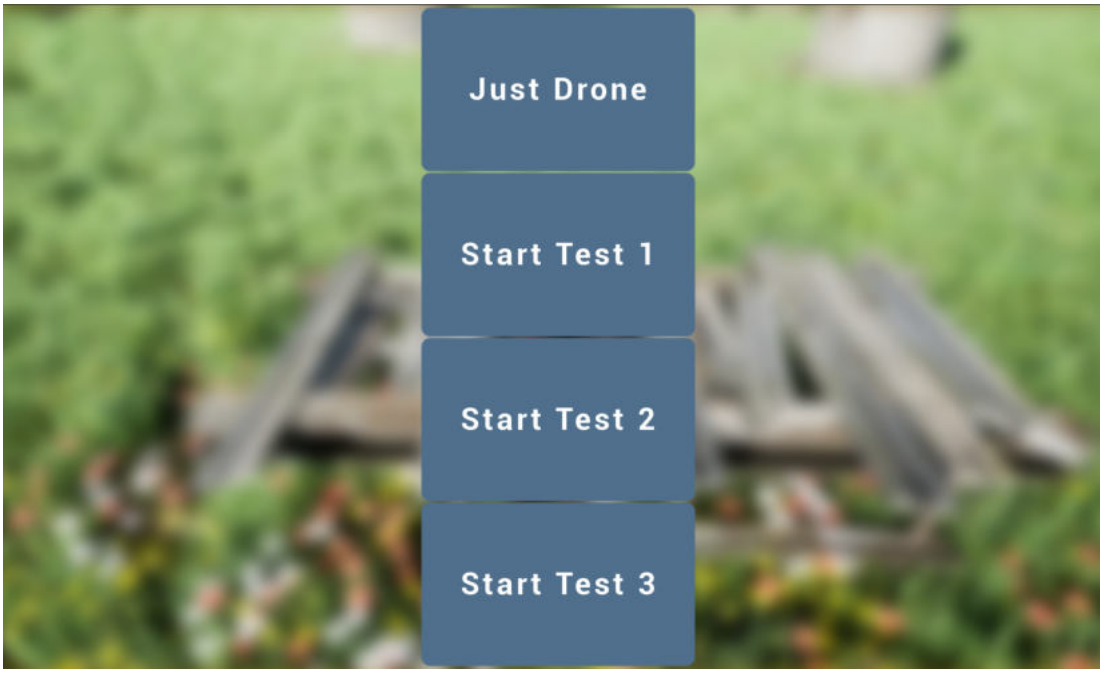


Figure 4.10: Option Menu Screen Capture

Depending on which button is clicked, an **Event Dispatcher** is called.

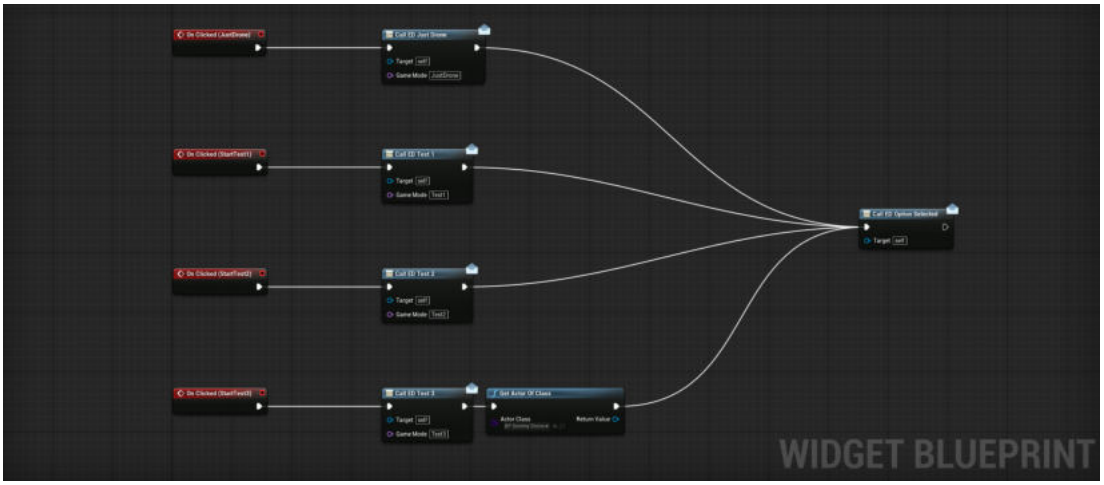


Figure 4.11: Option Menu Widget BP

By binding one or more events to it, the flow can cause all of those events to fire once the Event Dispatcher is called. These events can be bound within a Blueprint Class, as discussed later. Each call has an input variable **GameMode**,

strictly related to the option selected, so it can be set equal to:

- `JustDrone`
- `Test1`
- `Test2`
- `Test3`

The variable's pin type is `name`. These special types of variable provide a very lightweight system for using strings, where a given string is stored only once in a data table, even if it is reused. They are immutable, and cannot be manipulated. The storage system and static nature of names means that it is fast to look up and access `names` with keys.
Inside the *SPPlatformCommunicationComponent* blueprint [BP] the event dispatchers are bind. Calling the Event Dispatcher will have no effect if there are no events bound to it. Consider that each Event Dispatcher has a list of events associated with it. The way to add an event to this list is by using a **Bind Event** node. Once the Bind Event is executed, the related **Custom Event** must be delegated by reference in order to execute the desired processes. These are the ones shown in the figure below.

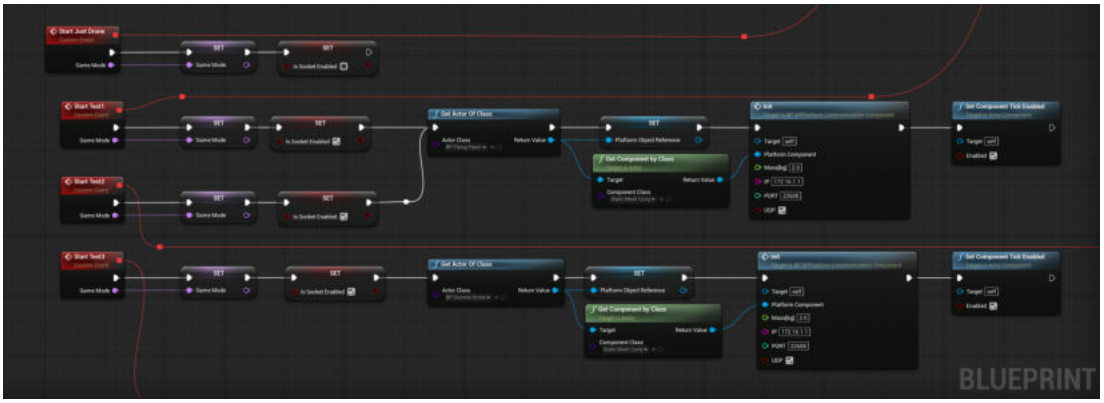


Figure 4.12: Custom Event BP Implementation

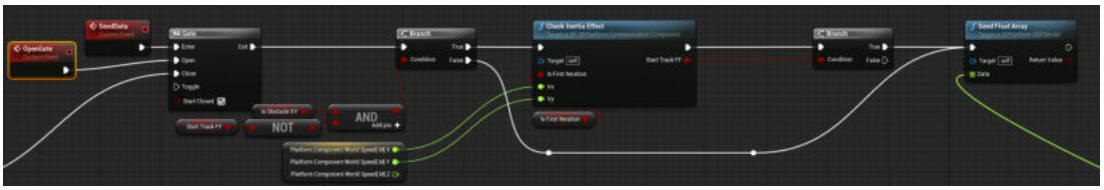
The Custom Event `JustDrone` sets the name variable `GameMode` and it also sets the flag `isSocketEnabled` on. Otherwise, according on the test selected, the *SPPlatformCommunicationComponent* is initialized with different **input**. The

Init node sets up the component for the communication with the platform. It takes as arguments:

- **Platform Component**: *Scene Component Reference*
- **Mass** : *flaot*
- **IP**: *string*
- **Port**: *integer*
- **UDP**: *bool*

The Event Tick is a simple event that is called on every frame of the simulation. The following figure represents the main flow of the component.
Here, if the condition is satisfied, the function `GetPlatformData` is executed, in this way is possible to update the drone data before send them to the platform. Otherwise, if the simulation is the **Just Drone**, the aforementioned step is not required. Before send the data, a *switch* statement is used to control the flow of program execution. A switch node reads in a data input (in this case the `name` variable `GameMode`), and based on the value of that input, sends the execution flow out of the matching execution output. In details, if the current test is the first one, data must be sent every each frame. Otherwise, the **Send Float Array** function is called only when a flag from the python API is received.
For the test 2 and 3, data must be sent only when the drone (the real one or the *ghost* one) moves under the force feedback effect. This behaviour is guaranteed due to the Blueprint node **Gate**. A Gate node is used as a way to open and close a stream of execution. Simply put, the Enter input takes in execution pulses, and the current state of the gate (open or closed) determines whether those pulses pass out of the Exit output or not. So the flag received from the python API opens the Gate. Then the simulation flow is different for each case.

(a) Flag ObsatcleXY Flow



(b) Test 2 Open Gate Flow

Figure 4.13: Test 2 BP Details

Regarding the test 2, before start sending data to the platform some preliminary considerations must be made. As shown in figure 4.13, once the gate is open, the python API also advertises if the obstacle lies on the drone XY plane. Therefore the custom event in figure 4.13 firstly sets the two bool variables `IsObstacleXY` and `IsFirstIteration` equal to `true` in order to initialize properly the `CheckInertiaEffect` input parameter, once the function will be called. Then if the condition are met, the aforementioned function is executed. That is because if the FF is applied the drone keep going again the obstacle for a little amount of time (about 0.2 seconds) due to the inertia effects. From the simulation point of view, this motion is imperceptible. While in the *platform space* this behavior may corrupt the feedback perception.

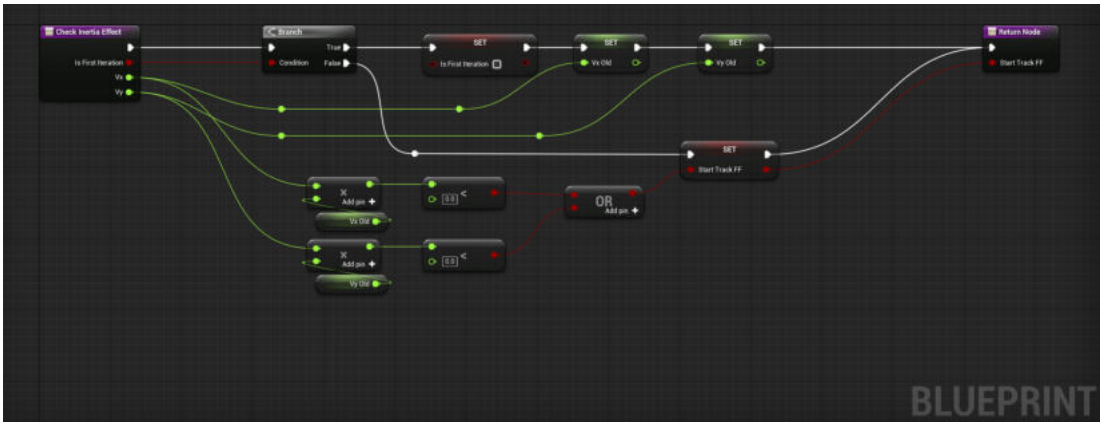


Figure 4.14: Check Inertia Effect Function BP

In the figure above is shown the BP function `CheckInertiaEffect` implementation. It is executed once the obstacle is detected and it checks the following condition:

$$V_{x_{t_0}} \cdot V_{x_{t_i}} < 0 \ \vee V_{y_{t_0}} \cdot V_{y_{t_i}} < 0 \tag{4.1}$$

Where:

- V_{t_0}: is the component velocity at the first iteration
- V_{t_i}: is the component velocity at each tick

Within this control low only the reliable data are sent to the platform, since it returns `true` only when the drone is under the FF effects. Therefore this condition simply ensures the change of direction and as a result data are sent exclusively when the drone is moving away from the obstacle. Once it returns `true`, it is no more executed until the obstacle avoidance algorithm ends the execution. At this point, the python API fires the custom event `StopSending` linked to a `Sequence` Node. The `Sequence` node allows for a single execution pulse to trigger a series of events in order. The node may have any number of outputs, all of which get called as soon as the Sequence node receives an input. They will always get called in order, but **without any delay**. Therefore first it closes the Gate, then it resets the bool variables (`IsObstacleXY` and `IsFirstIteration`) equal to `False`.
Concerning the test 3, the flow is pretty similar to the previous one. Here, since the *Ghost* drone is spawned with the desired kinematic states, the function `CheckInertiaEffect` is no more required. Once the obstacle is detected, the Python API 3.3 opens the gate and it sets a timer, when the time expires a custom event is executed in order to close the Gate. Only in the third case, before opening the **Gate**, the node **DoOnce** is run. As the name suggests, it will fire off an execution pulse just once. From that point forward, it will cease all outgoing execution until a pulse is sent into its *Reset* input. Within this behaviour the motion feedback from the platform is feasible, since as explained in section 3.3 after a few seconds the *Ghost* Drone teleport over the Main Drone position. Once the obstacle is overcame, due to a trigger box, the **DoOnce** node is reset, thus enabling the socket to send data again.

4.6 Data Processing - Offline Phase

Before perform run-time test, during the simulation Drone position and orientation (with respect to the world frame) has been logged for both test 2 and test 3 4.5. Therefore the BP function library `LogDataDrone` has been implemented.

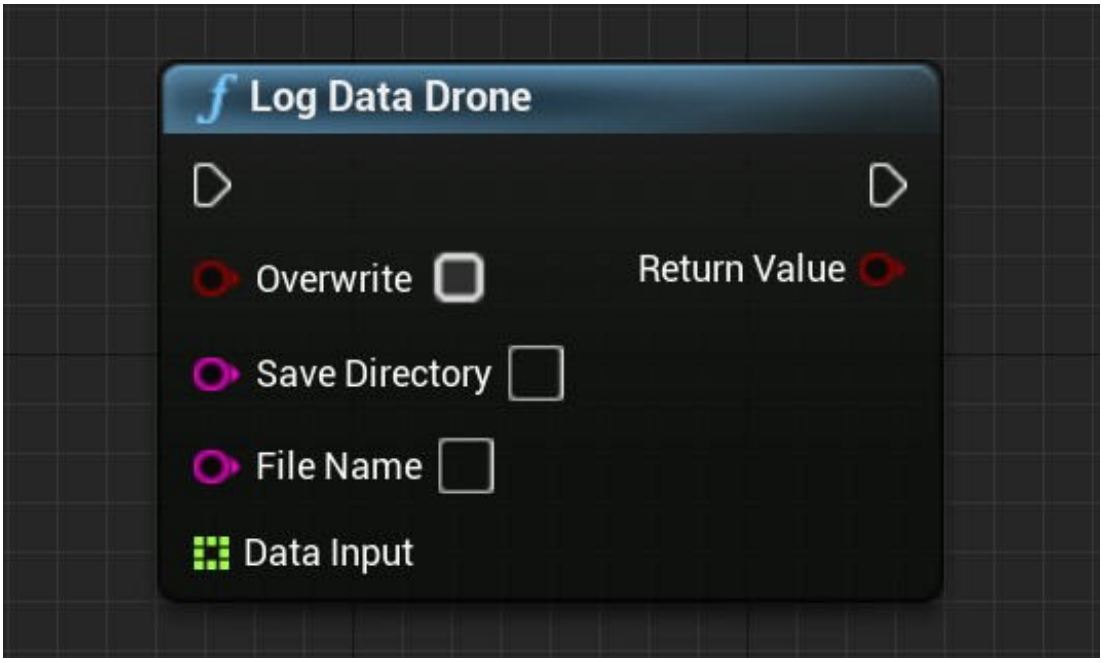


Figure 4.15: LogDataDrone BP Node

Blueprint Function Libraries are a collection of static functions that provide utility functionality not tied to a particular simulation object. These libraries can be grouped into logical function sets or contain utility functions that provide access to many different functional areas.
In details this function takes as input the following arguments:

- Overwrite `bool`: flag that allows the file overwriting
- SaveDierctory `TString`: the directory where the file is saved
- FileName `TString`: the name of the saved file
- DataInput `TArray<float>`: data to log

Please note that UE4 provides its own C++ types, by using container class (e.g. `TString` or `TArray`) UE4 ensures speed, memory efficiency and safety. This preliminary phase guarantees the correct behaviour of the SP7, during the motion feedback.
Due to the input array elements, data are wrote each tick in a *.txt* file with the following shape:

$$\left|\begin{array}{c|c|c|c|c|c|c} t_0 & x_0 & y_0 & z_0 & \theta_{x_0} & \theta_{y_0} & \theta_{z_0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ t_n & x_n & y_n & z_n & \theta_{x_n} & \theta_{y_n} & \theta_{z_n} \end{array}\right|$$

Where:

- t: current timestamp
- x, y, z: position of the drone
- θ_{x_n}, θ_{y_n}, θ_{z_n}: orientation of the drone

The values above are obtained within the BP node `GetGameTimeinSeconds` which returns the current game time is seconds, while `GetWorldRotation`, `GetWorldLocation` nodes returns, respectively, the rotation and the location of the component, in world space. Since the SP7 works with a different reference frame, given the pose vector ${}^{\text{UE4}}\text{C} = [x\ y\ z\ \theta_x\ \ \theta_y\ \theta_z]$ the final vector ${}^{SP7}c$ is given by:

$${}^{\text{SP7}}\text{C} = \begin{bmatrix} \text{R}_p & 0_{3\times3} \\ 0_{3\times3} & \text{R}_p \end{bmatrix} {}^{\text{UE4}}\text{C} \tag{4.2}$$

Where:

$$\text{R}_p = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } \text{R}_o = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{4.3}$$

The following figure shows how data have been post processed thought MATLAB in order to feed properly the platform and therefore to analyze the feedback behaviour.

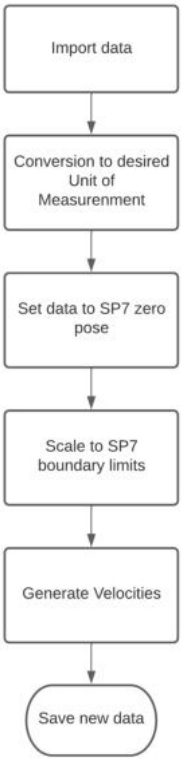


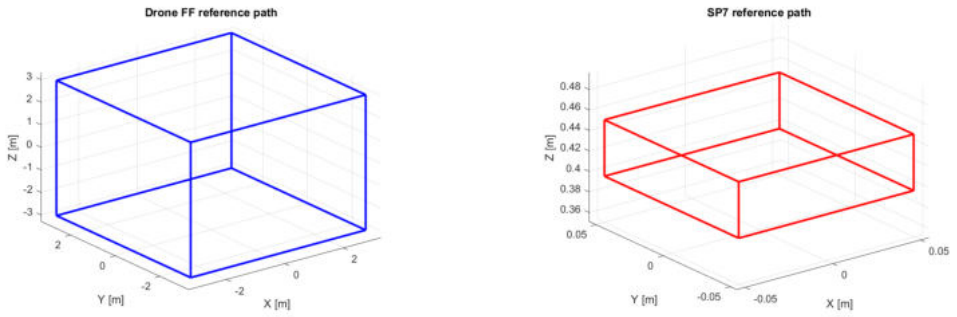
Figure 4.16: Data Post Processing Flow

Below data validation related to the test 3 will be presented.
First of all, since drone data are logged inside the UE4 environment and only when an obstacle has been encountered, the first step involves the conversion to the desired units of measurement (meters for the position and degrees for the orientation) and then the initialization of data in order to set the values starting from zero.
Before feeding the platform, drone data must be scaled since the SP7 work-space has its own boundary limits as shown in the following table.

	x	y	z	θ_x	θ_y	θ_z
Unit	[m]			[deg]		
Min	-0.05	-0.05	0.396	-5	-5	-180
Max	0.05	0.05	0.451	5	5	180

Table 4.3: SP7 Boundary Limits

Regarding the test 3 evaluation, drone data are logged within a certain time frame where the quadrotor is under the force field effect, therefore in figure 4.17a the path space is illustrated.



(a) Drone FF Reference (b) SP7 Reference

Figure 4.17: Work-Space References

This is done due to the MATLAB function:

```
rescale(A, l, u, 'InputMin', inmin, 'InputMax', inmax)
```

where:

- `A`: Input Array
- `InputMin`: Minimum of input range, specified as a scalar. Specifying an input range either expands or shrinks the range of the input data. For instance, `rescale` sets all elements that are less than the specified input minimum to the `InputMin` value before scaling.
- `InputMax`: Maximum of input range, specified as a scalar.
- `l`: Lower bound, specified as a scalar.
- `u`: Upper bound, specified as a scalar.

and the aforementioned function uses the following equation:

$$A' = l + \frac{(A - inmin)(u - l)}{(inmax - inmin)} \tag{4.4}$$

Regarding the `inmin` and the `inmax` values they refer to the figure 4.17a since it is assumed that the drone, under the FF, cannot move more than $\pm$ 3 meters

with respect to the initial position. While the lower and the upper bound refer to the table 4.3.
Before feeding the platform, data has been plotted in order to ensure that the SP7 boundary limits are not exceeded.
As example, figure 4.18 shows the drone which is going against an obstacle. In this case, the *ghost drone* is spawned and pushed away with the repulsive velocity computed by the FF implementation 3.3.

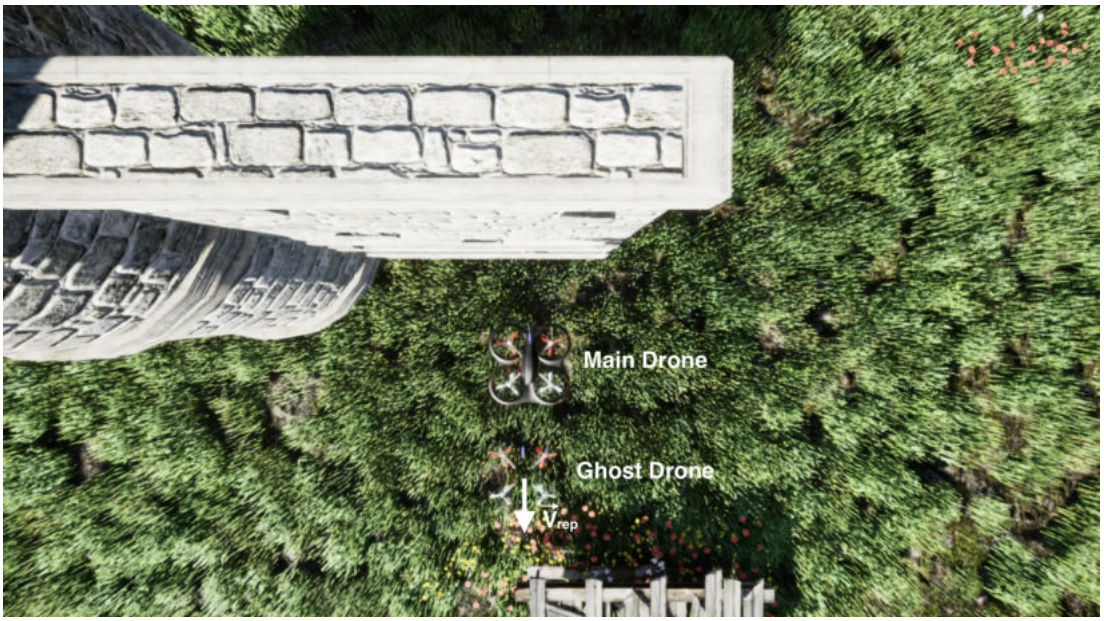
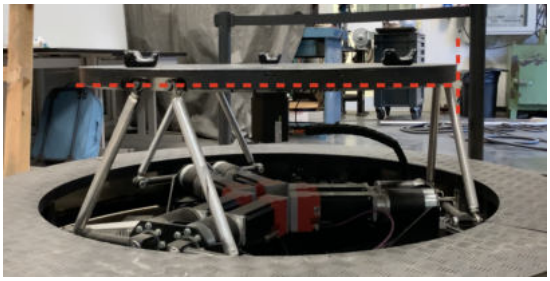


Figure 4.18: Simulation Screen Capture

Since the closest point of the obstacle belongs to the drone x-axis, the ghost drone is moving along the same axis (in the opposite direction), by keeping the same altitude. Figures 5.2 shows the real motion of the platform, once it has been fed with the post-processed data.



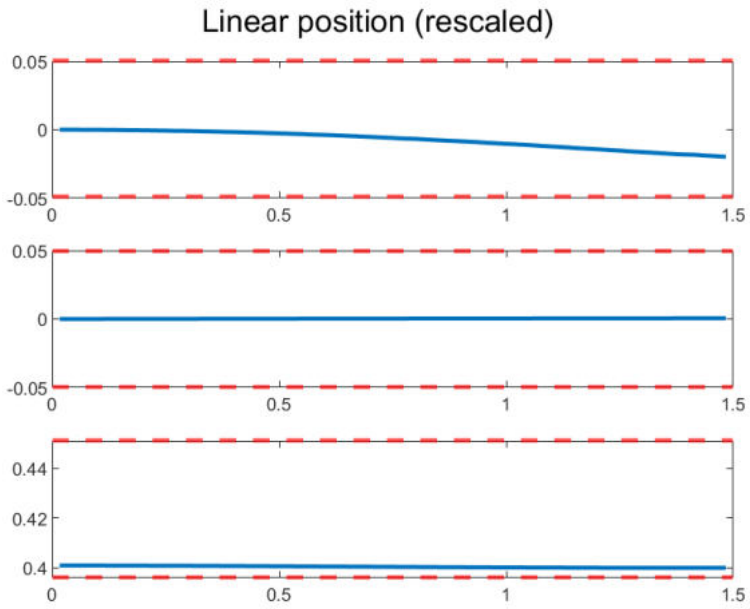
(a) SP7 Initial Position

(b) Start Motion Feedback

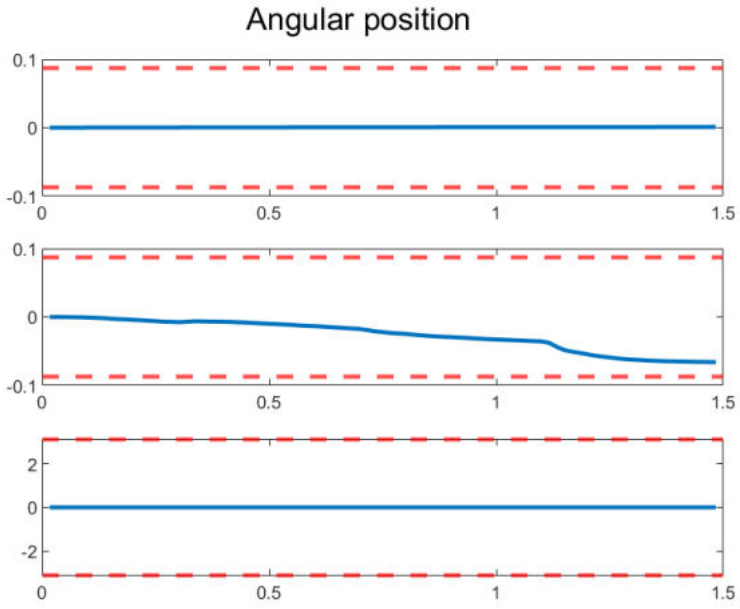
(c) SP7 Final Position

Figure 4.19: SP7 motion feedback

Plots in figures 4.20 show the linear positions, angular positions, linear velocities and the angular velocities of the platform over time. Regarding the translation, accordingly with the simulation, the platform move only along its x-axis (see subfigure 4.20a) and the same applies for the orientation: only θ_y changes over time. This behavior recalls the drone body dynamics that allows desired movement.

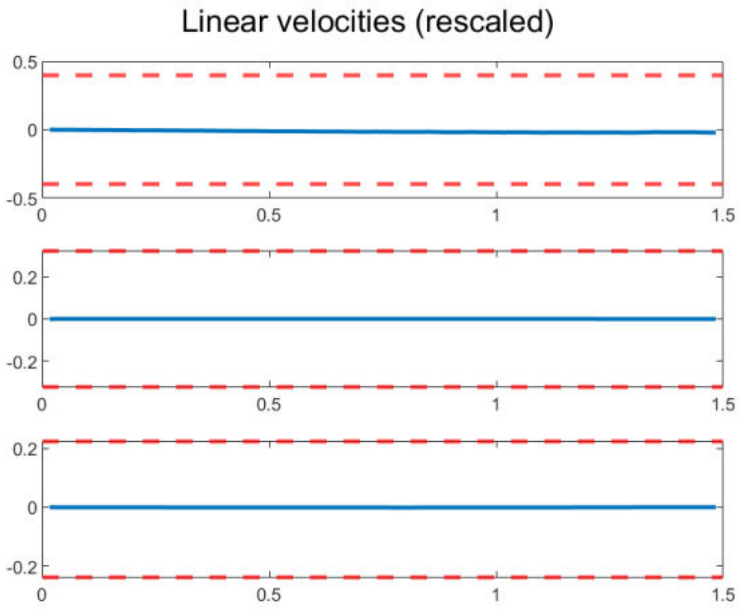


(a) Linear Position $x,\ y,\ z$ [m]

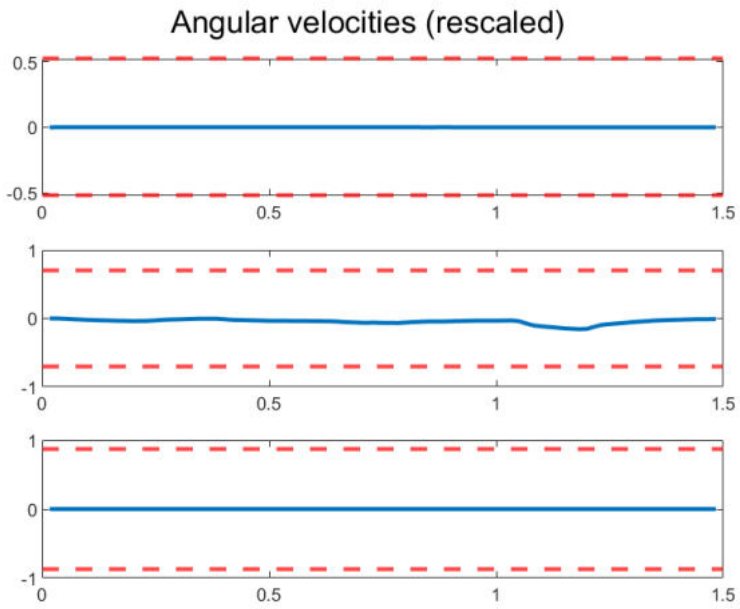


(b) Angular Position $\theta_x,\ \theta_y,\ \theta_z$ [rad]

Figure 4.20: Data Scaled Plot (SP7 Space)



(c) SP7 Linear Velocities $\dot{x},\ \dot{y},\ \dot{z}$ [m/s]



(d) SP7 angular velocities $\dot{\theta}_x,\ \dot{\theta}_y,\ \dot{\theta}_z$ [rad/s]

Figure 4.20: Data Scaled Plot (SP7 Space) (Cont.)

4.7 Questionnaire

Once set up the configuration, a jury of volunteer tried the different tests in order to evaluate the motion feedback perception. Several number of participant took place to the experiments. Pre-test questionnaires have been first given to the subjects, then have been invited to sit on the platform within the HMD. Moreover in order to allow the subject to get accustomed with the experience, a short training session has been provided on motion and visuals on the HMD. A mid-test questionnaire has been given to the participants at the conclusion of each set to provide a feedback. In addition, at the end of the final test, the user is also asked to complete a questionnaire to assess simulator sickness. Depending on how quickly the subject responds to the questionnaire, the overall experimentation duration has been between 20 and 30 minutes.

Figure 4.21: Picture of User During the Test

In summary, the *modus operandi* is the following:

1. Anti Covid rules

2. Brief explanation
3. Agreement signing
4. Pre-Test questionnaire filling
5. Training session
6. Session I
7. Mid-Test questionnaire filling
8. Session II
9. Mid-Test questionnaire filling
10. Session III
11. Mid-Test questionnaire filling
12. Post-Test questionnaire filling
13. Component Sanitation

Moreover, in order to speed up the session, python scripts are launch within a .bat file.

Questionnaire 1

Pre-Test

1. Have you had any previous virtual reality experiences?

   Yes

   No

2. Earlier experience with motion simulators?

   Yes

   No

3. Do you have any medical conditions?

   Yes

   No

   If yes, kindly specify below:

4. When have you eaten or had a drink last time? Kindly specify the approximate time.

5. How do you feel physically?

   On the scale of 0 to 100

   (0 → worst condition ; 100 → Excellent condition)

6. How do you feel mentally?

   On the scale of 0 to 100

   (0 → timid ; 100 → enthusiastic)

Mid-Test

Test 1

1. How well do you perceive the presence of the obstacle?

   On the scale of 0 to 100

   (0 → for nothing ; 100 → very well)

2. How well do you perceive the position of the obstacle?

   On the scale of 0 to 100

   (0 → for nothing ; 100 → very well)

3. How realistic is the experience?

   On the scale of 0 to 100

   (0 → for nothing ; 100 → very real)

4. Does the motion feedback improve the obstacle avoidance?

   On the scale of 0 to 100

   (0 → for nothing ; 100 → absolutely yes)

5. Is the motion feedback faithful to the simulation?

   On the scale of 0 to 100

   (0 → for nothing ; 100 → absolutely yes)

6. Is the feedback disturbing your piloting (*)?

Mid-Test

On the scale of 0 to 100

(0 → absolutely no ; 100 → absolutely yes)

7. Is the feedback confusing about the obstacle perception (*)?

   On the scale of 0 to 100

   (0 → absolutely no ; 100 → absolutely yes)

8. Is the feedback helpful to avoid obstacle?

   On the scale of 0 to 100

   (0 → absolutely no ; 100 → absolutely yes)

Leave a comment:

Mid-Test

Test 2

1. How well do you perceive the presence of the obstacle?

   On the scale of 0 to 100

   (0 → for nothing ; 100 → very well)

2. How well do you perceive the position of the obstacle?

   On the scale of 0 to 100

   (0 → for nothing ; 100 → very well)

3. How realistic is the experience?

Mid-Test

On the scale of 0 to 100

(0 → for nothing ; 100 → very real)

4. Does the motion feedback improve the obstacle avoidance?

   On the scale of 0 to 100

   (0 → for nothing ; 100 → absolutely yes)

5. Is the motion feedback faithful to the simulation?

   On the scale of 0 to 100

   (0 → for nothing ; 100 → absolutely yes)

6. Is the feedback disturbing your piloting (*)?

   On the scale of 0 to 100

   (0 → absolutely no ; 100 → absolutely yes)

7. Is the feedback confusing about the obstacle perception (*)?

   On the scale of 0 to 100

   (0 → absolutely no ; 100 → absolutely yes)

8. Is the feedback helpful to avoid obstacle?

   On the scale of 0 to 100

   (0 → absolutely no ; 100 → absolutely yes)

Leave a comment:

Mid-Test

TEST 3

1. How well do you perceive the presence of the obstacle?

   On the scale of 0 to 100
   (0 → for nothing ; 100 → very well)

2. How well do you perceive the position of the obstacle?

   On the scale of 0 to 100
   (0 → for nothing ; 100 → very well)

3. How realistic is the experience?

   On the scale of 0 to 100
   (0 → for nothing ; 100 → very real)

4. Does the motion feedback improve the obstacle avoidance?

   On the scale of 0 to 100
   (0 → for nothing ; 100 → absolutely yes)

5. Is the motion feedback faithful to the simulation?

   On the scale of 0 to 100
   (0 → for nothing ; 100 → absolutely yes)

6. Is the feedback disturbing your piloting (*)?

   On the scale of 0 to 100
   (0 → absolutely no ; 100 → absolutely yes)

Mid-Test

7. Is the feedback confusing about the obstacle perception (*)?

   On the scale of 0 to 100

   (0 → absolutely no ; 100 → absolutely yes)

8. Is the feedback helpful to avoid obstacle?

   On the scale of 0 to 100

   (0 → absolutely no ; 100 → absolutely yes)

Leave a comment:

Questionnaire 3

Post-Test

1. Did you feel nausea?

   Yes

   No

2. Did you feel dizzy?

   Yes

   No

3. Did you feel eye strain?

   Yes

   No

Post-Test

4. Did you have any other eye trouble?

   Yes

   No

5. Did you have headache?

   Yes

   No

6. Did you feel mental pressure?

   Yes

   No

7. Were you tired?

   Yes

   No

8. Did you feel anxiety (uneasiness)?

   Yes

   No

9. Did you fear?

   Yes

   No

   If yes, kindly specify at which point:

10. Did you bored?

    Yes

Post-Test

No

If yes, kindly specify at which point:

If you have any comments about your overall experience, kindly mention below:

Experimental Results

This chapter focuses on the experimental evaluation of the integrated system explained in section 4. Figure 5.1 presents the final setup configuration, used to perform the final tests. The HTC Vive base station (30) are used to track the HMD, within the recommended physical space ($5m^2$) while the other components functionalities have been deeper investigated in the previous chapters.

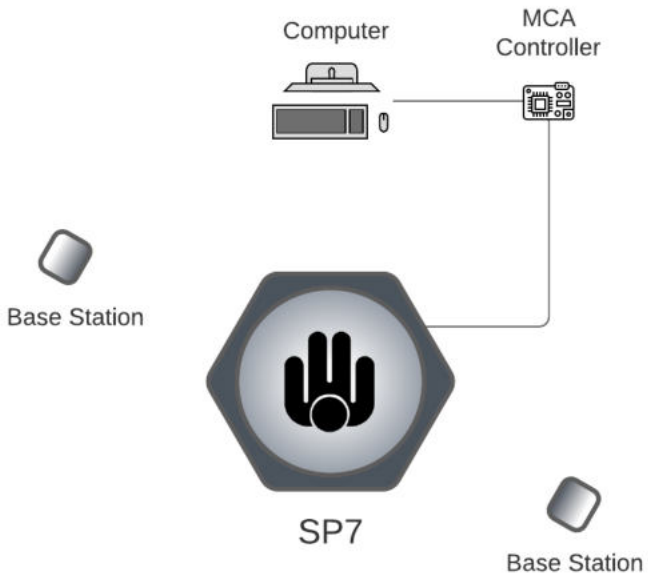
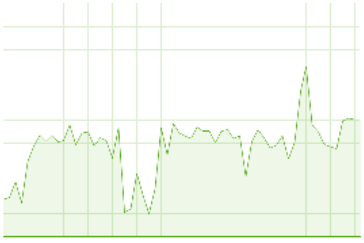


Figure 5.1: Setup Layout (Top View)

During the extensive testing of the platform, it has been clear how complex

the tuning process of the MCA was and how important it was to achieve a smooth motion simulation experience. This is a delicate process since the motion feedback is supposed to advertise the pilot about the obstacle position, wrong information may lead to collision. On the other hands, sudden movement or strong vibration may annoy the user experience.
Once the simulation architecture has been completed, the UE4 project has been packaged. Aside high-performance hardware component (GPU, RAM, etc...) AirSim also recommends to run the application within a solid state drive [SSD], since bad performance interfaces with the drone control loop and if it does not run at high frequency (300/500 HZ) then the drone do not fly as expected (1). Another suggestion related to reduce the hard-drive Input/Output is to package the final project. Code and content are packaged to ensure they are up to date and in the right format for the target platform. Several steps are involved in this process. First, the source code will be compiled, then, upon compilation, all required content will be converted into a format that the target platform can use and the executable will be available.



(a) Normal Editor Hard-Drive Usage

(b) Cooked Application Hard-Drive Usage

Figure 5.2: Hard-Drive Usage Comparison

5.1 Questionnaire Evaluation

In order to validate the motion feedback assistance, the questionnaire scores have been evaluated as follow.
First of all, questions were grouped into three categories:

- **Avoid** [Q4, Q8]: representing how the motion feedback help to avoid obstacle
- **Perceive** [Q3, Q5, Q6*]: representing how much fine the pilot perceive the obstacle thanks to the motion feedback
- **Response Control** [Q1, Q2, Q7*]: representing how much well the pilot is able to react after the motion feedback

Therefore for each user, scores has been processed as follow within each category:

$$s_j = \frac{\sum q_{i,j}}{\mathrm{TOT}_j} \text{ for } i = 1, ..., n \text{ and } j = 1, 2, 3 \tag{5.1}$$

Where:

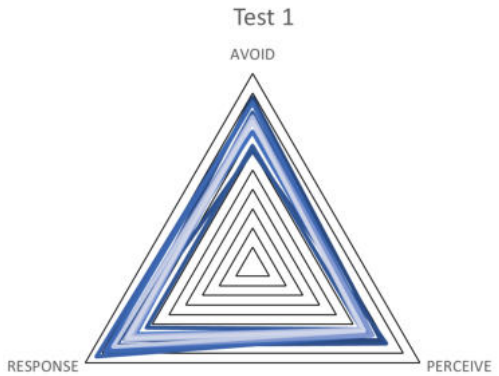
- j: represent the category (e.g., Avoid, Perceive, Response Control)
- i: the question belonging to the category
- TOT_j: the maximum score for each category

Please note that for the question marked by the asterisk the final values is evaluated as: $100 - \text{score}$ since in these case higher values mean negative ratings.
Figure 5.3 show the actual results.

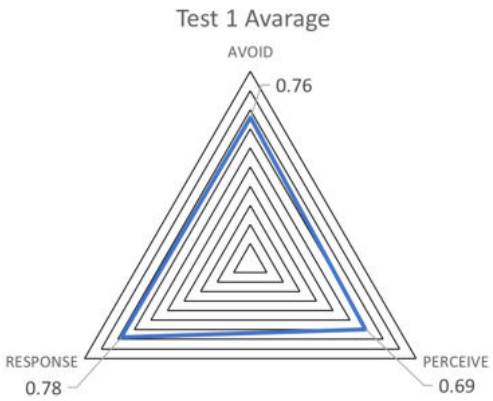
#	T1			T2			T3		
	AVOID	PERCEIVE	RESPONSE	AVOID	PERCEIVE	RESPONSE	AVOID	PERCEIVE	RESPONSE
1	0.88	0.77	0.77	0.90	0.83	0.87	0.93	0.90	0.92
2	0.60	0.80	0.82	0.95	0.92	0.97	1.00	1.00	1.00
3	0.63	0.65	0.63	0.90	0.67	0.93	0.90	0.77	0.93
4	0.80	0.70	0.73	0.90	0.80	0.83	0.90	0.90	0.90
5	0.85	0.72	0.93	0.83	0.73	0.73	1.00	1.00	0.98
6	0.85	0.72	0.88	0.83	0.78	0.82	0.93	0.88	0.95
7	0.70	0.70	0.85	0.83	0.73	0.85	0.90	0.85	0.95
8	0.78	0.65	0.70	0.80	0.75	0.90	0.93	0.78	0.95
9	0.75	0.70	0.73	0.88	0.65	0.73	0.98	0.87	0.95
10	0.78	0.72	0.83	0.83	0.78	0.88	0.95	0.87	0.95
11	0.78	0.65	0.72	0.85	0.75	0.87	0.93	0.93	0.93
12	0.75	0.67	0.77	0.85	0.73	0.73	0.95	0.82	0.95
13	0.78	0.65	0.73	0.80	0.65	0.77	0.90	0.83	0.95
14	0.75	0.67	0.83	0.83	0.68	0.78	0.95	0.87	0.93
15	0.75	0.67	0.75	0.83	0.77	0.83	0.95	0.83	0.93
Avarage	0.76	0.69	0.78	0.85	0.75	0.83	0.94	0.87	0.95

Figure 5.3: Classified Scores

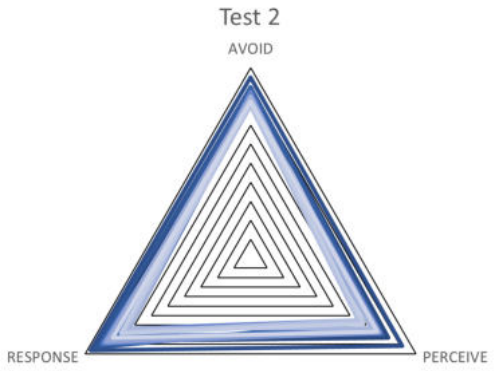
While figures below present radar plots in order to compare the average values for each test.



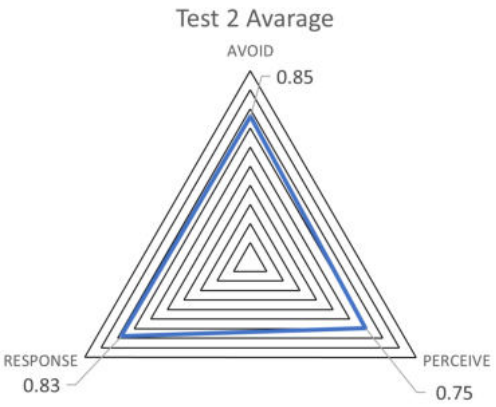
(a) Test 1 Radar Plot [0 1]



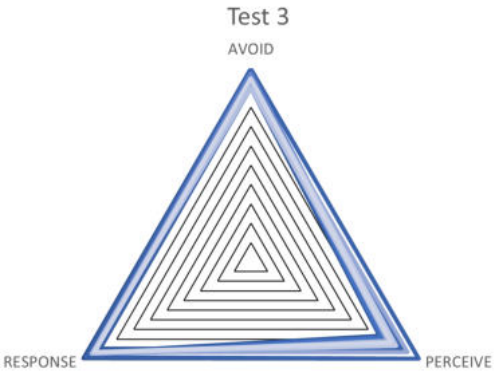
(b) Test 1 Average Values Radar Plot [0 1]



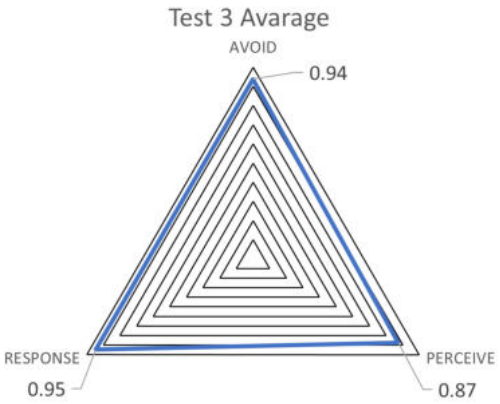
(c) Test 2 Radar Plot [0 1]



(d) Test 2 Average Values Radar Plot [0 1]



(e) Test 3 Radar Plot [0 1]



(f) Test 3 Average Values Radar Plot [0 1]

Since during the test 1 the user always receives the motion feedback, the scores are lower than the other. That is because in this case continuous cue perceptions lead to desensitisation hence resulting counterproductive for the obstacle perception. Moreover, motion sickness reduces piloting performance so the response control is less quick.

Even if also in the test 2 the user input is discarded during the FF execution, at this instance scores are better. Since there is no motion feedback during the normal piloting, the user learns that a cue implies the presence of an obstacle,

therefore this improves the perception and hence the response control.
As mentioned in the sections before, the main difference in the last test is that the user input is not discarded during the force feedback. As well as the test before, the platform rests until sensors detect an obstacle. So the user quickly reacts to the cue, perceiving properly the obstacle position and therefore he avoids better obstacle.
As final consideration, the motion feedback improves the obstacle perception and the response control, two essential features to avoid collision.

Conclusions and Future Works

For this work, an environment has been built within the UE4 in order simulate a drone piloting. The plugin AirSim allows to performs reliable simulation, integrating sensors data and providing APIs for control the drone. Therefore a control setup allows to remotely control the drone, while the pilot sits over the SP7. Visual cues from the HMD guarantee a very immersive experience. A FF algorithm, implemented via Python APIs, ensures responses in presence of obstacles. Hence several test have been designed to navigate in a predefined scenario and performances are evaluated with the purpose of assessing how the user response to the motion cues.
Questionnaire results validate the initial hypothesis, that the motion feedback improves obstacle avoidance, perception and control response while piloting.
Based on the actual architecture, any algorithm can be used in order to improve drone response in case of obstacle since it leads to the corresponding platform movement. In addition, a platform with a greater workspace enhances longer cues in case of obstacle. Hence this better advertise the user about the obstacle position.
Starting from this work is possible to develop a real semi-autonomous teleoperation, where the motion cue provides a safer navigation, integrating also a dynamic obstacle avoidance algorithm.

Appendix A: setting.json

```
{
  {
  "SeeDocsAt":
      "https://github.com/Microsoft/AirSim/blob/master/docs/settings.md",
  "SettingsVersion": 1.2,
  "SimMode": "Multirotor",
  "ClockSpeed": 1,
  "EngineSound": true,
  "LogMessagesVisible": false,
  "PawnPaths": {
    "MainDrone": { "PawnBP":
        "Class'/AirSim/Blueprints/BP_FlyingPawn.BP_FlyingPawn_C'" },
    "DummyDrone": { "PawnBP":
        "Class'/Game/DroneBase/DummyDrone/BP_DummyDrone.BP_DummyDrone_C'"
        }
  },
  "Vehicles": {
    "MainDrone": {
      "VehicleType": "simpleflight",
       "RC": {
        "RemoteControlID": -1
      },
      "DefaultVehicleState": "Armed",
      "PawnPath": "MainDrone",
      "AutoCreate": true,
      "IsFpvVehicle": true,
      "CameraDefaults": {
```

```
  "CaptureSettings": [
    {
      "ImageType": 0,
      "Width": 256,
      "Height": 144,
      "FOV_Degrees": 90,
      "AutoExposureSpeed": 100,
      "AutoExposureBias": 0,
      "AutoExposureMaxBrightness": 0.64,
      "AutoExposureMinBrightness": 0.03,
      "MotionBlurAmount": 0,
      "TargetGamma": 1.0,
      "ProjectionMode": "",
      "OrthoWidth": 5.12
    }
  ]
},
"Sensors": {
  "DistanceFront": {
    "SensorType": 5,
    "Enabled": false,
    "MaxDistance": 10,
    "X": 0,
    "Y": 0,
    "Z": -1,
    "Yaw": 0,
    "Pitch": 0,
    "Roll": 0,
    "DrawDebugPoints": false
  },
  "DistanceRight": {
    "SensorType": 5,
    "Enabled": false,
    "MaxDistance": 7,
    "X": 0,
    "Y": 0,
    "Z": -1,
    "Yaw": -90,
```

```
      "Pitch": 0,
      "Roll": 0,
      "DrawDebugPoints": false
    },
    "DistanceLeft": {
      "SensorType": 5,
      "Enabled": false,
      "MaxDistance": 10,
      "X": 0,
      "Y": 0,
      "Z": -1,
      "Yaw": 90,
      "Pitch": 0,
      "Roll": 0,
      "DrawDebugPoints": false
    },
    "LidarSensor": {
      "SensorType": 6,
      "Enabled": true,
      "NumberOfChannels": 512,
      "RotationsPerSecond": 120,
      "PointsPerSecond": 200000,
      "Range": 10,
      "X": 0,
      "Y": 0,
      "Z": 0,
      "Roll": 0,
      "Pitch": 0,
      "Yaw": 0,
      "VerticalFOVUpper": 180,
      "VerticalFOVLower": -180,
      "HorizontalFOVStart": -180,
      "HorizontalFOVEnd": 180,
      "DrawDebugPoints": false,
      "DataFrame": "SensorLocalFrame"
    }
  }
},
```

```
    "DummyDrone": {
      "VehicleType": "simpleflight",
      "RC": {
        "RemoteControlID": -1
      },
      "AutoCreate": true,
      "DefaultVehicleState": "Armed",
      "AllowAPIAlways": true,
      "PawnPath": "DummyDrone",
      "EnableCollisionPassthrogh": false,
      "EnableCollisions": false
    }
  }
}
```

Appendix B: Compiled Questionnaire

Experimentation session for objective evaluation of motion feedback in motion simulators

UNIVERSITÀ DEGLI STUDI DI GENOVA

Date	15/03/2022
Time	18:20
Session number	1
Gender	M
Age	29

Questionnaire

March 2022

Pre-Test

Mid-Test

1. Have you had any previous virtual reality experiences?

   Yes ✓

   No

2. Earlier experience with motion simulators?

   Yes

   No ✓

3. Do you have any medical conditions?

   Yes

   No ✓

   If yes, kindly specify below:

4. When have you eaten or had a drink last time? Kindly specify the approximate time.

   14:30

5. How do you feel physically? 100

   On the scale of 0 to 100
   (0 → worst condition ; 100 → Excellent condition)

6. How do you feel mentally? 100

   On the scale of 0 to 100
   (0 → timid ; 100 → enthusiastic)

TEST 1

Pre-Test

1. How well do you perceive the presence of the obstacle?

   85

   On the scale of 0 to 100
   (0 → for nothing ; 100 → very well)

2. How well do you perceive the position of the obstacle?

   70

   On the scale of 0 to 100
   (0 → for nothing ; 100 → very well)

3. How realistic is the experience?

   90

   On the scale of 0 to 100
   (0 → for nothing ; 100 → very real)

4. Does the motion feedback improve the obstacle avoidance?

   80

   On the scale of 0 to 100
   (0 → for nothing ; 100 → absolutely yes)

5. Is the motion feedback faithful to the simulation?

   90

   On the scale of 0 to 100
   (0 → for nothing ; 100 → absolutely yes)

6. Is the feedback disturbing your piloting (*)?

   40

   On the scale of 0 to 100
   (0 → absolutely no ; 100 → absolutely yes)

7. Is the feedback confusing about the obstacle perception (*)?

   10

   On the scale of 0 to 100
   (0 → absolutely no ; 100 → absolutely yes)

8. Is the feedback helpful to avoid obstacle?

40

On the scale of 0 to 100
(0 → absolutely no ; 100 → absolutely yes)

Leave a comment:

I HAD TROUBLES FROM THE FORCE FEED BACK GIVEN FROM OBSTACLES BEHIND ME

TEST 2

1. How well do you perceive the presence of the obstacle?

90

On the scale of 0 to 100
(0 → for nothing ; 100 → very well)

2. How well do you perceive the position of the obstacle?

100

On the scale of 0 to 100
(0 → for nothing ; 100 → very well)

3. How realistic is the experience?

90

On the scale of 0 to 100
(0 → for nothing ; 100 → very real)

4. Does the motion feedback improve the obstacle avoidance?

100

On the scale of 0 to 100
(0 → for nothing ; 100 → absolutely yes)

5. Is the motion feedback faithful to the simulation?

85

On the scale of 0 to 100
(0 → for nothing ; 100 → absolutely yes)

Pre-Test

Pre-Test

4. Does the motion feedback improve the obstacle avoidance?

   100

   On the scale of 0 to 100
   (0 → for nothing ; 100 → absolutely yes)

5. Is the motion feedback faithful to the simulation?

   100

   On the scale of 0 to 100
   (0 → for nothing ; 100 → absolutely yes)

6. Is the feedback disturbing your piloting (*)?

   0

   On the scale of 0 to 100
   (0 → absolutely no ; 100 → absolutely yes)

7. Is the feedback confusing about the obstacle perception (*)?

   0

   On the scale of 0 to 100
   (0 → absolutely no ; 100 → absolutely yes)

8. Is the feedback helpful to avoid obstacle?

   100

   On the scale of 0 to 100
   (0 → absolutely no ; 100 → absolutely yes)

Leave a comment:

Post-Test

Post-Test

1. Did you feel nausea?
   Yes
   No ✓
2. Did you feel dizzy?
   Yes
   No ✓
3. Did you feel eye strain?
   Yes
   No ✓
4. Did you have any other eye trouble?
   Yes
   No ✓
5. Did you have headache?
   Yes
   No ✓
6. Did you feel mental pressure?
   Yes
   No ✓
7. Were you tired?
   Yes
   No ✓
8. Did you feel anxiety (uneasiness)?
   Yes
   No ✓
9. Did you fear?

Post-Test

Yes

No ✓

If yes, kindly specify at which point:

10. Did you bored?

Yes

No ✓

If yes, kindly specify at which point:

If you have any comments about your overall experience, kindly mention below:

Bibliography

[1] "Microsoft/AirSim," Microsoft, Jun. 2021. iv, 8, 9, 10, 11, 13, 29, 82

[2] A. Sharma, M. S. Ikbal, D. T. Cuong, and M. Zoppi, "A sliding mode-based approach to motion cueing for virtual reality gaming using motion simulators," *Virtual Reality*, vol. 25, no. 1, pp. 95–106, 2021. iv, 16, 17

[3] L. Binet and T. Rakotomamonjy, "Using haptic feedbacks for obstacle avoidance in helicopter flight," *Progress in Flight Dynamics, Guidance, Navigation, and Control–Volume 10*, vol. 10, pp. 47–70, 2018. iv, 18, 19

[4] A. Sharma, M. S. Ikbal, and M. Zoppi, "Acausal Approach to Motion Cueing," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 1013–1020, Apr. 2019. iv, 19, 20, 21

[5] "Unreal Engine — The most powerful real-time 3D creation platform," https://www.unrealengine.com/en-US/. iv, 27, 28

[6] "Drones and Augmented Reality -Powerful Tools when Disaster Strikes," Oct. 2016. 2

[7] S. Spiss, Y. Kim, S. Haller, and M. Harders, "Comparison of Tactile Signals for Collision Avoidance on Unmanned Aerial Vehicles," in *Haptic Interaction*, ser. Lecture Notes in Electrical Engineering, S. Hasegawa, M. Konyo, K.-U. Kyung, T. Nojima, and H. Kajimoto, Eds. Singapore: Springer, 2018, pp. 393–399. 3

[8] T. M. Lam, M. Mulder, and M. M. van Paassen, "Haptic Feedback for UAV Tele-operation - Force offset and spring load modification," in *2006 IEEE International Conference on Systems, Man and Cybernetics*, vol. 2, Oct. 2006, pp. 1618–1623. 3

[9] DroneDeploy, "Commercial Drone Industry Trends," https://medium.com/aerial-acuity/commercial-drone-industry-trends-aae2010ff349, Mar. 2017. 8

[10] J. Courbon, Y. Mezouar, and P. Martinet, "Autonomous navigation of vehicles from a visual memory using a generic camera model," *IEEE Transactions on Intelligent Transportation Systems*, vol. 10, no. 3, pp. 392–402, 2009. 8

[11] C. Goodin, M. Doude, C. R. Hudson, and D. W. Carruth, "Enabling Off-Road Autonomous Navigation-Simulation of LIDAR in Dense Vegetation," *Electronics*, vol. 7, no. 9, p. 154, Sep. 2018. 8

[12] "AirSim-W — Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies," https://dl.acm.org/doi/abs/10.1145/3209811.3209880. 9

[13] M. H. McCrink and J. W. Gregory, "Blade Element Momentum Modeling of Low-Reynolds Electric Propulsion Systems," *Journal of Aircraft*, vol. 54, no. 1, pp. 163–176, Jan. 2017. 11

[14] L.-W. Tsai, *Robot Analysis: The Mechanics of Serial and Parallel Manipulators*. John Wiley & Sons, 1999. 14

[15] B. Dasgupta and T. S. Mruthyunjaya, "Closed-Form Dynamic Equations of the General Stewart Platform through the Newton–Euler Approach," *Mechanism and Machine Theory*, vol. 33, no. 7, pp. 993–1012, Oct. 1998. 14

[16] M. Li, T. Huang, J. Mei, X. Zhao, D. G. Chetwynd, and S. J. Hu, "Dynamic formulation and performance comparison of the 3-DOF modules of two reconfigurable PKM—the tricept and the trivariant," 2005. 14, 15

[17] D. Stewart, "A Platform with Six Degrees of Freedom," *Proceedings of the Institution of Mechanical Engineers*, vol. 180, no. 1, pp. 371–386, Jun. 1965. 14, 15

[18] M. Baarspul, "A review of flight simulation techniques," *Progress in Aerospace Sciences*, vol. 27, no. 1, pp. 1–120, Jan. 1990. 14

[19] Y. D. Patel and P. M. George, "Parallel Manipulators Applications—A Survey," *Modern Mechanical Engineering*, vol. 02, no. 03, pp. 57–64, 2012. 14

[20] J. Wang, C. Wu, and X.-J. Liu, "Performance evaluation of parallel manipulators: Motion/force transmissibility and its index," *Mechanism and Machine Theory*, vol. 45, no. 10, pp. 1462–1476, Oct. 2010. 14

[21] M. Carricato and V. Parenti-Castelli, "A Family of 3-DOF Translational Parallel Manipulators1," *Journal of Mechanical Design*, vol. 125, no. 2, pp. 302–307, Jun. 2003. 14

[22] J. C. F. de Winter, D. Dodou, and M. Mulder, "Training Effectiveness of Whole Body Flight Simulator Motion: A Comprehensive Meta-Analysis," *The International Journal of Aviation Psychology*, vol. 22, no. 2, pp. 164–183, Apr. 2012. 14

[23] K. Valentino, K. Christian, and E. Joelianto, "Virtual Reality Flight Simulator," /paper/Virtual-Reality-Flight-Simulator-Valentino-Christian/678707d1bbd2a274ae6644b7b9426812f687a15b, 2017. 14

[24] "A New Real-Time Flight Simulator for Military Training Using Mechatronics and Cyber-Physical System Methods — IntechOpen," https://www.intechopen.com/books/military-engineering/a-new-real-time-flight-simulator-for-military-training-using-mechatronics-and-cyber-physical-system-. 14

[25] K. Herbuś and P. Ociepka, "Integration of the virtual model of a Stewart platform with the avatar of a vehicle in a virtual reality," *IOP Conference Series: Materials Science and Engineering*, vol. 145, p. 042018, Aug. 2016. 14

[26] C.-T. Pan, P.-Y. Sun, H.-J. Li, C.-H. Hsieh, Z.-Y. Hoe, and Y.-L. Shiue, "Development of Multi-Axis Crank Linkage Motion System for Synchronized Flight Simulation with VR Immersion," *Applied Sciences*, vol. 11, no. 8, p. 3596, Jan. 2021. 14

[27] A. Kemeny and F. Panerai, "Evaluating perception in driving simulation experiments," *Trends in cognitive sciences*, vol. 7, no. 1, pp. 31–37, 2003. 14

[28] J. J. Gibson, "The perception of the visual world." 1950. 14

[29] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. Devitt, M. Hansen, D. Redfern, K. Rickard *et al.*, *Maple v Programming Guide: For Release 5*. Springer Science & Business Media, 2012. 16

[30] "VIVE Tracker (3.0) — VIVE United States," https://www.vive.com/us/accessory/tracker3/. 16, 81

[31] P. Stegagno, M. Basile, H. H. Bülthoff, and A. Franchi, "A semi-autonomous UAV platform for indoor remote operation with visual and haptic feedback," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, pp. 3862–3869. 18

[32] S. Lee, G. S. Sukhatme, G. J. Kim, and C.-M. Park, "Haptic control of a mobile robot: A user study," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3. IEEE, 2002, pp. 2867–2874. 18

[33] D. Repperger, "Active force reflection devices in teleoperation," *IEEE Control Systems Magazine*, vol. 11, no. 1, pp. 52–56, 1991. 18

[34] S.-G. Hong, J.-J. Lee, and S. Kim, "Generating artificial force for feedback control of teleoperated mobile robots," in *Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No. 99CH36289)*, vol. 3. IEEE, 1999, pp. 1721–1726. 18

[35] J. Borenstein and Y. Koren, "Real-time obstacle avoidance for fast mobile robots," *IEEE Transactions on systems, Man, and Cybernetics*, vol. 19, no. 5, pp. 1179–1187, 1989. 18

[36] B. Krogh, "A generalized potential field approach to obstacle avoidance control," in *Proc. SME Conf. on Robotics Research: The next Five Years and beyond, Bethlehem, PA, 1984*, 1984, pp. 11–22. 18

[37] E. Rimon, "Exact robot navigation using artificial potential functions," Ph.D. dissertation, Yale University, 1990. 18

[38] D. H. Kim and S. Shin, "Local path planning using a new artificial potential function composition and its analytical design guidelines," *Advanced Robotics*, vol. 20, no. 1, pp. 115–135, 2006. 18

[39] L. D. Reid and M. A. Nahon, "Flight simulation motion-base drive algorithms: Part 1. Developing and testing equations," *UTIAS Report, No. 296*, 1985. 21

[40] "Unreal Engine 4 Documentation," https://docs.unrealengine.com/4.26/en-US/. 24

[41] "Pygame/pygame," pygame, Mar. 2022. 31