University of South-Eastern Norway

www.usn.no

FMH606 Master's Thesis 2023

MSc Industrial IT and Automation

Sea Search and Rescue by Autonomous Drones in High Fidelity Visual and Physical Simulation

Rajeev Poudel

Faculty of Technology, Natural Sciences and Maritime Sciences

Campus Porsgrunn

University of South-Eastern Norway

www.usn.no

Course: FMH606 Master's Thesis, 2023

Title: Sea Search and Rescue by Autonomous Drones in High Fidelity Visual and Physical Simulation

Number of pages: 90

Keywords: Unreal Engine 4.27, Unreal Engine 5.1, Microsoft AirSim, ROS, SAR, Autonomous, UAV, UAS, YOLOv7, YOLOv8, Path Planning, Archimedean Spiral, Boustrophedon Path, Blueprints

Student: Rajeev Poudel

Supervisor: Fabio Andrade

External partner: N/A

Summary:

Unpredictable ship accidents still claim a lot of human life every year even with so many technological advancements. Maritime Search and Rescue missions during such hazards are mostly carried out with costly equipment and manpower that have some inherent estimation biases in many physical quantities. With small-size, lower operational cost, flexible aerial maneuverability, wireless communication, and mathematical computation ability, drones can be useful to minimize the costs and speed up the SAR operations without physical intrusion in dangerous post disaster scenarios. And due to the risky nature of the problem, simulation was the rational path initially.

But there was a shortage of previous literature that tried to especially solve this problem in proper simulation platform. Therefore, in the beginning a high fidelity dynamic marine simulation environment was created using Unreal Engine 4.27, Microsoft AirSim, and ROS which contained a Post Disaster Ship, other many debris, and human victims floating. Then, an autonomous SAR mission was planned and implemented for the drone with various pretrained YOLOv7 models that achieved high accuracy of victim detection. This work was published in IEEE/CVF WACV Conference, 2023. After that another iteration of autonomous simulation for tracking both treading and swimming victims with YOLOv8 pretrained models was carried out in custom environment in Unreal Engine 5.1 which also had satisfactory results. Furthermore, the ID and detected location in latitude and longitude of the tracked victim was made easily accessible for use in concerned places. Finally, the possibility for the cooperation and control of multiple drones working together for SAR missions was thoroughly discussed in the end.

Preface

This master's thesis is done as a requirement for the 30 credit thesis course in the MSc Industrial IT and Automation programme at USN. This thesis is closely related with USN responsibilities for European Union 's Project Valkyries that USN is part of with various other stakeholders whose primary aim was to create a common framework in Europe for the use of Autonomous Systems in post disaster conditions for saving lives of victims and mitigate any other financial and environment losses. This is a very noble cause to safeguard the lives of people making the world a safer and better place to live.

So, I am very much thankful to USN for providing me the opportunity through this thesis to work on this noble goal of increasing the well-being of the human-kind.

I would like express my deepest gratitude to my supervisor, Fabio Augusto de Alcantara Andrade for his relentless technical as well as emotional support to complete this work. I would not have been able to complete this work without his support. Also, I would like to thank PhD student Luciano Lima for his valuable support and motivation to write and publish the paper.

Furthermore, I am very much grateful to the coordinator of this programme MSc Industrial IT and Automation, Hakon Viumdal for his motivation and emotional support throughout my study period in USN.

Finally, I would like to thank my family, and friends for the constant love and support.

May 15, 2023

Rajeev Poudel

Contents

List of Figures

List of Tables

Nomenclature

AUV Autonomous Underwater Vehicle
API Application Programming Interface
BP Blueprint
CVF Computer Vision Foundation
FOV Field of View
GPS Global Positioning System
GPU Graphics Processing Unit
HITL Hardware-in-the-Loop
IEEE Institute of Electrical and Electronics Engineers
NED North East Down
PID Proportional Integral Derivative
ROS Robot Operating System
SAR Search and Rescue
SITL Software-in-the-loop
UAV Unmanned Aerial Vehicle
UAS Unmanned Aerial System
USV Unmanned Surface Vehicle
UE Unreal Engine
YOLO You Only Look Once

1 Introduction

With the everlasting curiosity and dissatisfaction in human beings to evolve further, it has been an ultimate goal of humanity to manifest intelligence into an immortal machine making it a fully autonomous entity. Furthermore, the need for survival has been the major driving factor for innovations to homo sapiens justifying the significance of the interest to develop an effective and robust network of smart systems that can independently operate to mitigate the damage inflicted after an unforeseen disaster. Consequently, the European Union also decided to initiate a project named "VALKYRIES" that works in Harmonization and Pre-Standardization of Equipment, Training and Tactical Coordinated procedures for First Aid Vehicles deployment on European multi-victim Disasters [1]. The University of South-Eastern Norway (USN) is one of the participants of this project with allocation of several responsibilities which will be the basis for this Master thesis.

This chapter will firstly impart the introduction to the thesis with the discussion of the necessary background and context of the overall study, succeeded by the formulation of research problem, questions, and objectives, and finally, the limitations to the study that are acknowledged upfront.

1.1 Context and Motivation

Initially, the thesis topic with the preliminary background and minimum expectations in a broad horizon along with the primary tools and software to be used, including chiefly Unreal Engine 4, Microsoft AirSim, and ROS, was decided, and allotted by USN. On further investigation, the topic was in line with the current work being carried out by the Autonomous Research Group, USN, in the prestigious VALKYRIES project. Therefore, with an aim to contribute to the ongoing work at the University as well as to augment the relevancy of the study, it was decided to direct the thesis in the corresponding direction.

Now diving into the specific topic, the aim of all major technological inventions is to make human life easier and better accompanied by economic prosperity with minimal harm to the environment. But there is always a persistent risk of uncontrollable and unpredictable occurrence of natural calamities due to imbalance of forces in Mother Earth that jeopardizes the main essence of mankind to live a happy and prosperous life. Likewise, the continual possibility of disasters in ships, buses, airplanes, and other mechanical systems also instigates peril to the survival of person itself followed by the costly economic and environmental deterioration. Hence, the proper management of natural catastrophes and other disasters has captivated a lot of attention since the start of human civilization because it is related to safeguarding the core subconscious instinct of any living species to survive. It resulted in all the novel technologies developed in the process of evolution being implemented, at their respective times, for addressing this issue.

Moreover, the enhancement of technology from steam powered rotatory mechanical machines in the first industrial revolution, to on off transistor logic powered second revolution, to microcontrollers, made with combination of transistors, propelled third revolution, and finally to internet driven fourth industrial revolution (Industry 4.0) has enabled the metamorphosis of machines to highly sophisticated, and self-governing entities, commonly known as autonomous systems, that are independently able to sense, perceive, plan, and act according to the

surrounding environment with proper training. It is, therefore, intriguing for people to deploy autonomous systems for amelioration of any post disaster condition which can reduce the associated risk, cost, and delay along with the increase of efficiency to salvage more lives, and curb down the financial and ecological loss.

In congruence, the European Union decided to commence the VALKYRIES project in 2020 with 17 main participants, including various universities and research institutes, hailing from several countries inside Europe [1]. The primary goal of the project, in simple words, is the creation of a uniform multi-country framework of autonomous post-disaster response system that can work together independently for multi-faceted operations like search and rescue, first aid, health, and safety of the victims, in the event of catastrophes encompassing various nations. Every country has some unique subtle and inconspicuous legal, socio-economic, and security fabric which substantiates the pertinency of this project. As a result, common concordance between the states is vital for cooperation and coordination in case of detrimental hazards.

Finally, USN is one of the member universities of the VALKYRIES project who has been allotted various duties. The elementary task relevant to the thesis topic is described in Task 4.1 titled as "First aid vehicles and supportive autonomous units" [1]. Basically, beginning with scrutinizing the cutting-edge technical possibilities for the reaction of unaided and sovereign artificial agents in first aid, USN needs to replenish the inconsistencies in their institutionalization and standardization to put forward a guideline for their embracement by the EU first aid responders. Furthermore, to showcase the application, USN has been designated an explicit use case involving post-disaster scenario of an oil tanker ship in the region of North Sea between Norway, Denmark, and Netherlands [1]. The fundamental aspects of the use case demonstration are the search and rescue of the victims with emergency care, oil spill detection, and salvage cargo with collaboration between the different responsible governmental authorities in many fronts. For that, the principal anticipated self-governing systems to be deployed are Unmanned Surface Vehicles (USVs) that need to replicate the tasks otherwise carried out by human first aid responders by sovereignly infiltrating the affected region, and Unmanned Aerial Vehicles (UAVs), especially drones, that can gather the essential information from higher elevation with minimum penetration into the potentially risky and treacherous areas. Moreover, it is also expected that the information gathered should be transmitted to a common framework, named as SIGRUN, developed of cloud-based database with linkage to web and mobile based applications.

Additionally, it is stressed that the implementations should be able to cope with the damage to the conventional communication framework when employed without the precise outlook of the cross-frontier and cross-sectorial BLOS (Beyond Line of Sight) missions [1].

1.2 Research Problem

Due to the inherent terrain intrusive nature of USVs with limited range of visibility compared to the flexible aerial maneuverability of drones with higher spectrum of perceptibility at a secure altitude, it is preferable to utilize drones for the initial surveillance of the potentially fatal vicinity of the disaster struck ship that forms the foundation for the deployment of USVs. Moreover, the elementary aim of establishment of any emergency management systems is to strengthen the probability of detecting and emancipating any threat from the survival of human life.

Based on these postulations, the research problem for this master thesis is formulated as follows: Develop a virtual reality simulation environment of a post-disaster scenario of an oil-tanker ship positioned in North Sea between Norway, Denmark, and Netherlands using Unreal Engine, and implement a network of multiple drones in ROS with interface to Microsoft AirSim that autonomously carry out reconnaissance missions with the focus on search and rescue of victims.

1.3 Research Objectives

Based on the context and the research problem, the following are the paramount objectives of the study:

a. To design and construct a sea simulation environment using Unreal Engine, Microsoft AirSim, and ROS.

b. To conceptualize as well as actualize the various strategies for cooperation between several self-governing drones to effectively inspect the locality of the wrecked ship prioritizing the detection of victims.

c. To convey useful information to concerned authorities from hazardous territory.

1.4 Research Questions

The following fundamental questions were triggered with the research objectives that guided the overall thesis study:

a. How to develop a high-fidelity sea simulation environment with a post disaster ship where multiple drones can be spawned and controlled?

b. How to distribute responsibilities among the individual drones?

c. How to locate the victims within the vicinity of the ship?

d. How to handle the dynamic sea environment where the victims and objects keep on moving?

e. How to plan the time and energy efficient path for the drones ensuring full coverage of the solicited area?

f. How to make the drones carry out the missions autonomously collaborating with each other?

g. How to communicate between the multiple agents in real-time?

h. How to transmit the information gathered by the drones to the concerned authorities remotely?

1.5 Limitations

The following are the major limitations of the study recognized upfront:

a. There will always be some bias and discrepancies in the simulation from the real world.

b. The downward facing camera in the drone even with the gimbal might not be completely stable, which might cause error in the georeferencing process.

c. The Odometry NED values are prone to errors because they are estimated values calculated based on other motion sensors.

2 Theory

This chapter introduces the major tools and the theory behind some of the methods used in this Master thesis.

2.1 Unreal Engine

Unreal Engine is an incredibly powerful and popular game development tool which is created and maintained by Epic Games [2]. It is a game engine especially popular for the creation of real-time 3D games, but it also supports creators across various industries to develop cutting-edge real-time 3D content, interactive experiences, and immersive virtual worlds. Therefore, lots of industries and academia use Unreal Engine which has a large user base around the world with a solid support framework. The basic introduction to installation and use of Unreal Engine with explanation of various features is available in structured form in [3].

2.2 Microsoft AirSim

AirSim is an open-source simulator platform built on Unreal Engine that is developed by the Microsoft Research Team with the primary goal to narrow the gap between simulation and reality to facilitate the development of autonomous vehicles, with elementary focus on aerial systems, by providing physically and visually realistic simulations [4]. It can offer real-time hardware-in-the-loop (HITL) simulations, with support for popular lighweight messaging protocols for drones like MavLink [5] working on popular hardware platforms like Pixhawk, by the help of a physics engine able to operate at a high frequency. In addition, it also supports software-in-the-loop (SITL) simulations with the availability of built-in default flight controller called *simple_flight*, which is used in this Master thesis, with also the support for PX4 and Ardupilot as external flight controllers [6].

The overall architecture of the AirSim system is shown in Figure that illustrates the core components and the interactions between them.

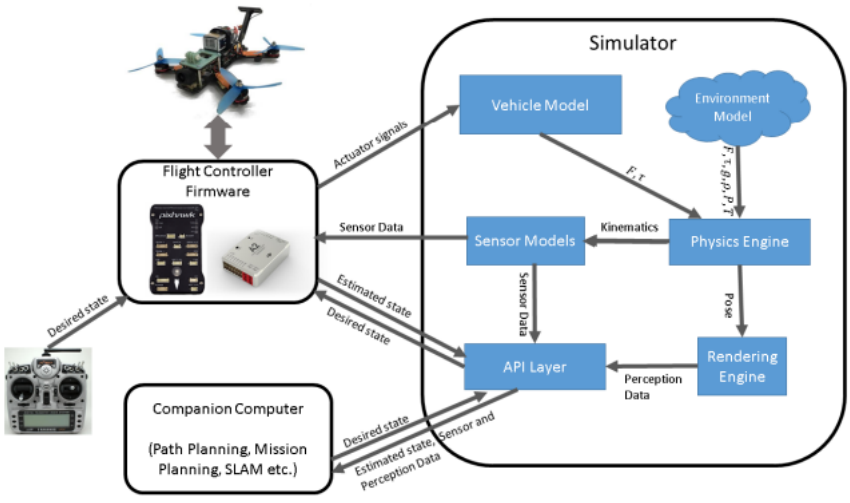

Figure 2.1. Architecture of the AirSim simulator with core components and interactions between them [4]

The Figure portrays the core components with modular design that includes simulator part with environment model, vehicle model, physics engine, sensor models, rendering interface, public API layer, and an interface layer for vehicle firmware or the companion computer [4]. This research thesis focuses on SITL simulation, and hence does not use physical firmware. So, the focus will be on the simulation in the local computer as if it is the companion computer of the drone which sends the desired state wish to the *Simulator* through the *API layer* and gets back the current estimated state as well as sensor, and perception data required for the autonomous search and rescue mission planning from the *API layer* of the simulator.

Basically, the built-in flight controller inside the simulator obtains desired state input from the companion computer, and sensor data from *Sensor models* and perception data from the *Rendering engine* which is Unreal Engine, then calculates the current state estimate and outputs the actuator control signals to the *Vehicle Model* to achieve the desired state. The *Vehicle Model* computes the forces, including forces generated from drag, friction and gravity simulated by various models for them, and torques generated by the simulated actuators to send to the *Physics Engine* that calculates the next kinematic state, expressed in term of 6 quantities as position, orientation, linear velocity, linear acceleration, angular velocity, and angular acceleration. The *Physics Engine* also considers the *Environment models* for gravity, air density, air pressure, magnetic field, and geographic location which together with kinematics forms the ground truth for the simulated *Sensor Models*. Also, the *Physics Engine* sends the current calculated pose of the drone to the *Rendering Engine* for the display, and the loop continues as seen in Figure. All the models used for simulating physical properties in [4] are proven dynamic physical models which justifies the high fidelity visual and physical simulation.

Furthermore, even though Microsoft has officially shutdown the further development of AirSim from 2022 and archived the official AirSim repository [7] to launch their new platform called Project AirSim [8], Codex laboratories LLC have forked the official AirSim repository, and continued the development of AirSim with a new name Colosseum [9] working with Unreal Engine 5 which was used for the solution in Chapter 4 of this Master thesis.

2.3 Robot Operating System (ROS)

ROS is an popular open-source robotics middleware framework running mainly on Unix-based platforms such as Ubuntu and Mac OS X systems [10]. It is not an operating system but provides similar functionalities such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Basically, the main goal of ROS is to provide a common, scalable, flexible, and language independent platform for robotics developers to share and reuse the code. The detailed explanation of all the concepts, installation procedures, tutorials, and other information about ROS can be found in an systematic form in its official documentation [10]. Furthermore, AirSim has a built-in wrapper for ROS that helps to interface the AirSim API as shown in Figure with ROS directly whose detailed explanation is given in [11]. This is extensively used throughout the Master thesis.

2.4 Archimedean Spiral

The Archimedean spiral (also known as the arithmetic spiral) is a spiral named after the 3rd-century BC Greek mathematician Archimedes [12]. It is the locus comprising of the locations of a point moving away from a fixed center point over time with a constant speed along a line that rotates with constant angular velocity as shown in Figure.

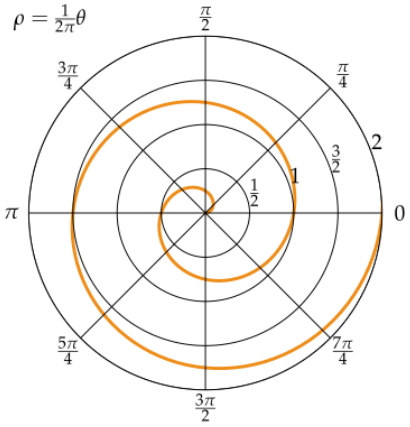

Figure 2.2. Archimedean spiral represented on a polar graph [12]

In polar coordinates (r, θ) it can be represented by the {equation}.

$$r = a + \frac{b}{2\pi} \cdot \theta$$

where *a* and *b* are real numbers.

The parameter *a* controls the position of the center point of the spiral. If *a* is positive then the center is shifted outward towards θ = 0, and if a is negative then the center of the spiral moves outward from the origin towards θ = π. Whereas parameter *b* controls the distance between the loops, which is equal for all the loops. This property of Archimedean spiral makes it suitable for full coverage of the unknown desired region with low or no overlap in search and rescue missions if the distance *b* between the loops is selected according to the Field of View (FOV) and height of the camera from the ground as shown in Figure which is discussed also by the authors in [13], [14], and [15].

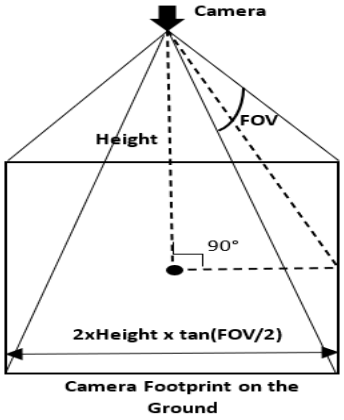

Figure 2.3. Effect of Camera FOV and Height from the surface on the Camera footprint

3 Published Paper

This chapter presents the paper [16] published by the author of this thesis, PhD Student Luciano Lima, and the supervisor of this thesis Fabio Andrade in Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) Workshops, 2023 during the duration of this Master thesis, and is an integral part of this Master thesis.

3.1 Abstract

This work presents a novel framework providing the ability to control an Unmanned Aerial System (UAS) while detecting objects in real-time with visible detections, containing class names, bounding boxes, and confidence scores, in a changeable high-fidelity sea simulation environment, where the major attributes like the number of human victims and debris floating, ocean waves and shades, weather conditions such as rain, snow, and fog, sun brightness and intensity, camera exposure and brightness can easily be manipulated. Developed using Unreal Engine, Microsoft AirSim, and Robot Operating System (ROS), the framework was firstly used to find the best possible configuration of the UAS flight altitude, and camera brightness with high average prediction confidence of human victim detection, and then only autonomous real-time test missions were carried out to calculate the accuracies of two pretrained You Only Look Once Version 7 (YOLOv7) models: YOLOv7 retrained on SeaDronesSee Dataset (YOLOv7-SDS) and YOLOv7 originally trained on Microsoft COCO Dataset (YOLOv7-COCO), which resulted in high values of 97.8% and 93.79%, respectively. Furthermore, it is proposed that the framework developed in this study can be reverse engineered for autonomous real-time training with automatic ground-truth labeling of the images from the gaming engine that already has all the details of all objects placed in the environment for rendering them onto the screen. This is required to be done to avoid the cumbersome and time-consuming manual labeling of large amount of synthetic data that can be extracted using this framework which could be a groundbreaking achievement in the field of maritime computer vision.

3.2 Introduction

Unforeseeable in nature, disasters involving ships at sea not only inflict costly economic and environmental damage, but also jeopardize the invaluable life of crew and passengers onboard. According to [17], there were a total of 892 shipping losses worldwide between 2012 to 2021 with 54 total mishaps alone in 2021. Even though the total number of global vessel hazards declined by around 57% over the decade, it is still a substantial amount with each case necessitating prompt and costly deployment of Search and Rescue (SAR) teams to rapidly curb down the resulting harm. And, naturally, the primary focus of all rescue missions is to first scour the inhospitable post-disaster region for victims and safeguard their lives. All this substantiates the research interest to effectively and efficiently utilize the existing cutting-edge scientific innovations to alleviate the threat on human life emanating from unpredictable maritime accidents.

However, the abundance of all the applicable contemporary technologies introduces perplexity in deciding the perfect combination between them for optimum performance. In general, almost

all major research conundrums are resolved with the thorough comprehension of the problem domain and taking inspirations from the phenomenon already occurring in nature. On breakdown of present real-life search and rescue operations, intuitively most of the associated expense including time and money is attributed to the transportation of human first responders in boats, helicopters, and aircrafts [18]. In addition, the involvement of humans, pursuant to [19], brings upon various errors due to estimation biases of different physical quantities such as under-estimation of horizontal distance, over-estimation of height when looking down and under-estimation when looking up. These drawbacks can be overcome using Unmanned Aerial Systems (UAS) that have small-size, lower operational cost, flexible aerial maneuverability, wireless communication, and mathematical computation ability. UAS equipped with simple RGB and/or thermal cameras and either onboard or cloud-based processing capability which facilitates the use of deep convolutional neural networks (CNN) based object detection models, as discussed by the authors in [20], [21], [22], [23], [24] and [25], can best mimic the action of rescue personnel flying in helicopters or aircrafts for finding the victims in hazardous territories, making the rescue process more efficient. Furthermore, among different modern deep learning based object detection models [26], the state-of-the-art YOLOv7 that transcends all other recognized object detectors in speed and accuracy [27] is here considered the most suitable one because in critical real-time SAR missions both response time and accuracy are equally important for saving human life. Therefore, the starting scientific dilemma is now narrowed down to the paramount research question that forms the main basis for this work which is: How to find the best possible configurations of the UAS and state-of-the-art object detection models for working together in real-time with optimal accuracy of victim detection at an erratic post-disaster ship scenario?

With this question in mind, simulation seems to be the only plausible path forward initially because of the risk, price, time, and effort involved to set up the physical test environment at sea with real persons and UAS with cameras, not to mention the absurd complications in the re-enactment of the alternating scenario in the aftermath of an actual ship accident. Moreover, the general prerequisites of the simulation platform to be used can also be deduced from the research question as: (1) It should be able to produce detailed reproduction of a disaster-struck ship surroundings with high quality of graphics; (2) It should allow the replica of UAS with various sensors to be spawned and controlled in the fabricated environment; (3) It should have an interface to a mechanism capable to control as well as read and process sensor data from a real UAS, and execute object detection models, enabling transferability to real-world applications; and (4) It should have the ability to pass a continuous image stream from the replicated UAS that can be fed as input to object detection models for real-time processing.

Unreal Engine 4 [28] with the integration of AirSim [29], and Robot Operating System (ROS) [30], on the basis of [31], [32], [33], [34] and [35], has the potential to fulfill all the requirements of the simulation platform for this work as mentioned above. But when the requirements are actually materialized with the combination of Unreal Engine, AirSim, ROS, and Object Detection Models, a novel framework originates that answers the research question.

Therefore, this work follows the steps according to the requirements to firstly develop the framework. Then, using this framework, the object detection models are evaluated to find the finest configurations for achieving high accuracy of victim detection in real-time.

Hence, the main contributions of this paper are summarized as:

- The creation of a high-fidelity changeable sea simulation environment, where the deep-rooted challenges in the maritime computer vision such as the different light conditions, altitudes, sea colors, buoyancy, objects movement, camera exposures, brightness, weather, size of the objects, among many others, can be easily controlled. This also allows to inexhaustibly generate synthetic data for training new models.

- The development of a framework with the constructed simulation environment to evaluate the performance of the cutting-edge object detection models with the input images from the UAS in real-time autonomous SAR missions, which can directly be transferred to real-world UAS applications.

- The proposal to reverse engineer the created framework for autonomous real-time training of object detection models with the automatic ground-truth labeling of the desired objects in the images from the UAS which could be a breakthrough in maritime computer vision.

3.3 Development of the Novel Framework

This section describes the overall steps carried out based on the requirements of the simulation platform mentioned in the previous section.

3.3.1 Virtual Environment

In this section, all the steps to build the simulated environment will be presented.

The simulation environment is composed of a oil tanker, objects and people in water, and a small boat where the drones are deployed from.

In Figure 3.1, the environment is presented, highlighting the oil tanker. Another angle of the environment, highlighting the objects and people can be seen in Figure 3.2.

Figure 3.1. Simulated environment from oil tanker side.

Figure 3.2. Simulated environment from objects side.

3.3.1.1 Environment Project

The Environment Project [36] is an open source environment simulation project for Unreal Engine 4. It is the continuation of the Ocean Project, and has many features, such as ocean simulation, sky simulation, buoyancy, time, and fish plugins. In this work, the simulation environment was built on top of an existing example world that is made available by the Environment Project.

Two important configurations that are only present when building sea environments are the color of the ocean and the waves. It is possible to choose a darker or brighter ocean or more blue or green, for example. Regarding the waves, it is possible to choose the height, direction, among others, to make a more stormy or calm sea. In the Environment Project world, these configurations are in the Blueprint "BP_Ocean". Additionally, the various environmental aspects like sunlight intensity, brightness, atmospheric light, fog, and others were present in the blueprint "BP_Sky".

In addition, it is possible to configure weather parameters such as wind, rain, among others, which are also present in any world of Unreal Engine 4 but have their own plugin in the Environment Project.

3.3.1.2 Post-Disaster Oil Tanker

The first element that was added to the environment was a post disaster ship.

Figure 3.3. Post Disaster Oil Tanker.

Unreal Engine 4 Marketplace has much content available for download, both free and paid. The content that was chosen for this work is called "Post-Apocalyptic Oil Tanker" and was made available for purchase in 2017 by the content creator "mikkotahtinen". An illustration of the ship can be seen in Figure 3.3. It is important to note that the content that is downloaded is composed by many separate blueprints (Figure 3.4). The creator of the world needed to build the oil tanker with the desired content. One advantage was that in the content there were many other interesting objects such as containers, that were added in the environment developed by this work.

Figure 3.4. Blueprints available in the "Post-Apocalyptic Oil Tanker" product.

3.3.1.3 People

As the goal of this proposed framework was to provide a realistic environment, it was required to populate it with people. This work focused on including people treading water to simulate

victims in a sea disaster. However, it is also possible to include people walking in the ship or swimming. Many characters and animations can be downloaded for free at Mixamo [37] by Adobe.

In this work, around six different characters were used, all of them with the animation of treading water.

After downloading the animation, the physics aspects must be properly configured. The two configurations that allow the person to properly tread water and be affected by the water movement are to enable collision and choose the "SK_Mannequin_PhysicsAsset" as the "Physics Asset Override". This was implemented with the proper understanding of similarity in the bone structure and hierarchy of the "SK_Mannequin" Asset which is the default third person character of Unreal Engine, and the Mixamo character. This also allows the manual control of the Mixamo characters using the physics control capability of the "SK_Mannequin".

3.3.1.4 Buoyancy Configuration

One of the main aspects of this work is to have objects which are affected by the stream and waves of the environment. Therefore, the buoyancy must be correctly configured, otherwise, the objects would just be with a static position, frozen in the 3D space, without following the water movement.

To configure the buoyancy, first the "Buoyant Force" component was added to the Blueprint, then, the buoyancy points were decided with the assistance of the arrow tool as shown in Figure 3.5. Therefore, it is possible to know the exact position to add the buoyancy in the "Test Points" configuration element. For the swimmer, three buoyancy points were added. This varies for different objects.

Figure 3.5. Buoyancy points configuration.

Finally, Figure 3.6 presents the three test points which were included for the swimmer blueprint.

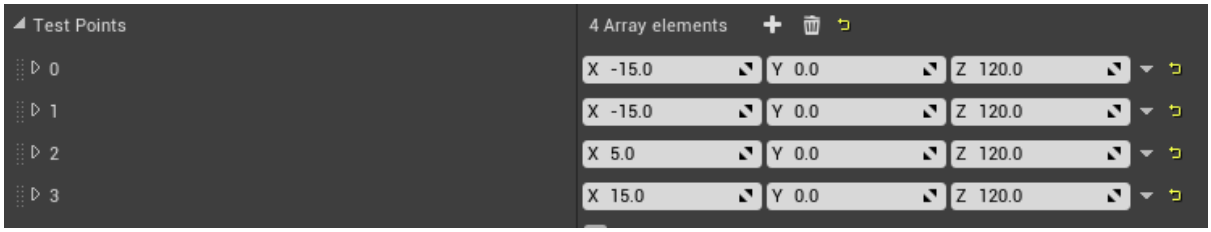

Figure 3.6. Buoyancy points placement.

It is important to note that the same procedure must be performed for all objects placed on the sea, such as the oil tanker, containers, oil barrels, buoys, among others. Nevertheless, the buoyancy points should be added to only one blueprint of any object, and then the same object can be easily replicated with the same settings.

3.3.1.5 Other Aspects

In addition, buoys, and other objects, such as containers and oil barrels with buoyancy added following the same procedure as people, were placed as seen in Figure 3.7.

Figure 3.7. Top view with objects.

3.3.2 Initial Setup of the UAS

Firstly, the AirSim plugin was integrated into the custom Unreal environment following the procedures as explained in the AirSim documentation [38]. All settings, except for the camera, were kept as default. A single multirotor UAS named "Drone_1" was spawned in the environment with "PlayerStart" placed on top of a rescue fishing boat as shown in Figure 3.8.

Figure 3.8. Initial UAS Setup in the Virtual Environment.

The camera settings were modified facilitating the UAS to have a single camera of resolution 640x640, which is the YOLOv7 model standard image resolution, field of view (FOV) of 90 degrees, and gimbal enabled with perfect stabilization of 1 and pitch of -90 degrees making the camera face vertically downward. In addition, the sensors like IMU, Magnetometer, GPS, and Barometer were also enabled automatically if the settings were left unchanged for the Multirotor sim mode as mentioned in the AirSim documentation [38].

Furthermore, complying with the directives specified, AirSim ROS wrapper was setup for Noetic version of ROS inside Windows Sub-system for Linux (WSL) 2 with Ubuntu 20.04 as Linux distribution on a Windows 10 computer having NVIDIA GeForce RTX 2080 Ti Graphical Processing Unit (GPU). It primarily contained two nodes among which the mostly used first node named "airsim_node" was a wrapper over AirSim's multirotor C++ client library that was comprised of various publishers, subscribers, services, and parameters.

3.3.3 Manual Control of the UAS

Next, using the "Twist" ROS message type, the velocity command subscriber topic from the wrapper allowed the movement of the UAS in all directions with the input of both linear and angular velocities in x, y, and z coordinates. For utilizing this feature to manually move the UAS in a desired way in the simulation environment, a ROS package named "AS_RoS_Teleop" was used that linked the different keyboard keys with separate control commands to publish velocity twist messages in the chosen topic.

3.3.4 Implementation of YOLOv7 in ROS

Subsequently, the effort of implementing YOLOv7 in ROS was eased with the ready-made ROS package titled "yolov7_ros" which was a ROS wrapper built over the original framework by the official developers of YOLOv7 [27]. After that, the weights of the chosen pertained YOLOv7 models were downloaded, and the class names for the respective models in the required txt file format were saved in separate folders. Then, the path to the model weights, class names, and the image topic were specified accordingly in the launch file to initiate the node for the real-time detection and visualization of the detections along with the bounding boxes, class names, and confidence scores using the desired YOLOv7 model one at a time.

3.4 Evaluation of Object Detection Models

This section explains the different procedures adopted to evaluate the performance of the object detection models for real-time detection of human victims in autonomous UAS missions.

3.4.1 Selection of Pretrained YOLOv7 Models on Different Datasets for Evaluation

As this study was in its early phase, it was decided to utilize the ready-to-use YOLOv7 models that were already trained on datasets containing people because the focus of this study was to detect human victims with high accuracy in the post-disaster scenarios.

The first obvious choice was the originally trained YOLOv7 model on Microsoft COCO (Common Objects in Context) [39] which was a large-scale dataset developed for object detection, classification and segmentation with 91 labeled objects constituting also people designated as "person" class. Due to the core nature of any Deep CNN based models including YOLOv7 to learn patterns in the training image using shifting convolution operations, it was important to assess the type of human images in this dataset. So on further scrutiny, it was found that the majority of the images were taken in canonical perspective [40] with different viewing angles.

Secondly, in search of datasets specially concentrating on the marine environment and aerial images, SeaDronesSee [41] was found, which was also a large-scale dataset from different aerial perspectives developed with focus on SAR operations on the sea using UAS. This was completely relevant for this work. In addition, the SeaDronesSee team had also trained YOLOv7 in their own dataset, and made the model freely available in project GitHub [42]. The output labels in this model were swimmer (people floating with stretched hands and legs), boat, jet ski, buoy, and lifesaving appliance (life jacket/lifebelt).

3.4.2 Experimentation with Various Configurations

The main beauty of the developed framework was that it enabled numerous experiments with minimal efforts which otherwise would have been either impossible or extremely difficult in real-life.

However, to make the study more focused in accordance with all other experimental studies, the variables to be considered in this work were also reduced from the plethora of the manipulable variables. Thus, keeping constant the environmental factors such as dark blue ocean shade, low wave amplitude and velocity, normal level of atmospheric and other lights, only the UAS position, especially height, and camera brightness was manipulated. The camera brightness was altered by changing the post-process settings present inside the camera component of the main parent blueprint of AirSim Camera named "BP_PIPCAMERA". Also, to further reduce the variables involved in this study, the camera brightness was changed as very low, low, normal, high, and very high. When the environment is executed in AirSim Game Mode, the images rendered on the screen are from the external camera which is also a child of the parent AirSim Camera. Hence, when the brightness of the camera was changed, it affected the image displayed on the viewport as seen in Figure 3.10.

Therefore, the starting experiment was carried out by freely traversing the UAS in the environment with different camera brightness and YOLOv7 models. On doing so, some interesting phenomenon of human victim detection were observed for both the models.

With the YOLOv7-SDS model selected, all the objects were detected as "boat" class in low or normal camera brightness for all heights of the UAS. But when the brightness was high, the model started to detect floating people with hands and legs moving as "swimmers" whereas other objects were still as "boat". Meanwhile, with the YOLOv7-COCO model chosen, the human buoyant victims were correctly classified as "person" class mostly in low heights with low or normal brightness.

For concretizing these observations, a separate test area with just the imported six characters was created as shown in Figure 3.9.

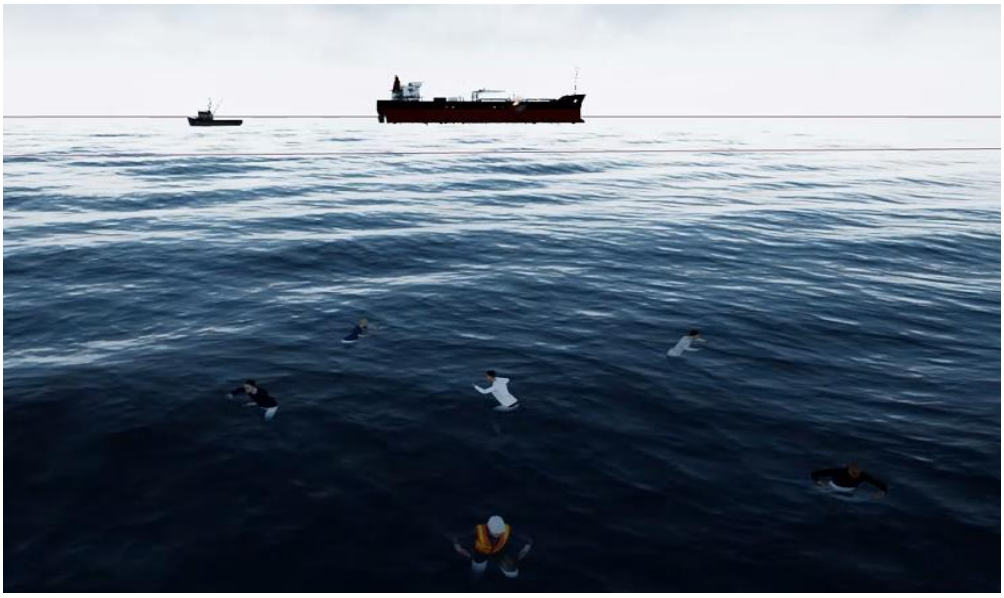

Figure 3.9. Isolated test region with just humans.

After that, the UAS was manually flown to the center of the testing region, and slowly only the altitude of the UAS was elevated from low to high and vice versa with different camera brightness each time for both YOLOv7-SDS and YOLOv7-COCO models. Concurrently, the detections with bounding boxes and confidence scores, the average prediction confidences and the altitude were closely monitored as shown in Figure 3.10.

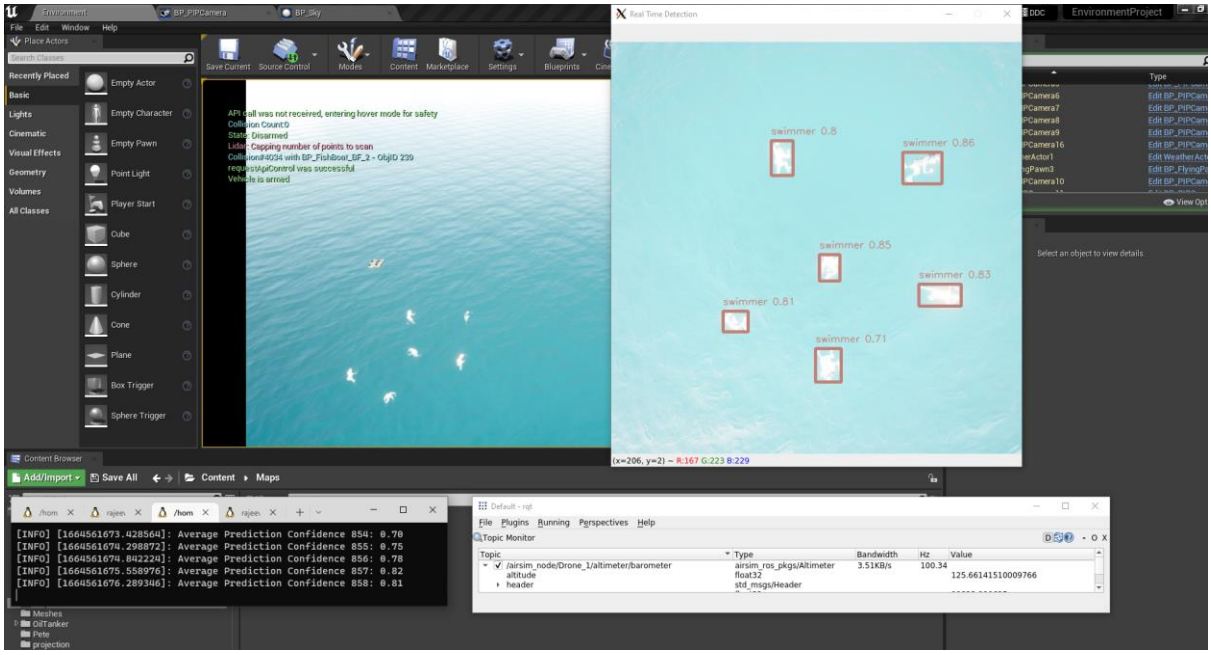

Figure 3.10. Illustration of the experimental procedures followed in the testing region.

Finally, after exhaustive trials it was found that the YOLOv7-SDS model had the highest average prediction confidence of detecting human victims as "swimmers" class at 8 meters from the sea level with a very high camera brightness, whereas the YOLOv7-COCO model had the highest average prediction confidence of detecting human victims as "person" class at 2 meters from the sea level with low camera brightness.

3.4.3 Path Planning for the Autonomous Mission

As the primary objective of this study was to evaluate the performance of the models for victim detection by skipping the arduous process of deploying the UAS in actual post-disaster scenarios with a simulated one, there was a need to replicate the mission that would have been employed in real-life, which could be used to gather the test images after detection by the models for empirical accuracy calculation.

Moreover, the predetermination of the specific height and camera settings of the UAS also laid the foundation for the autonomous surveillance mission. Using the distributed node processing capability of the ROS framework, the responsibilities of taking the UAS to the appropriate location in the environment, and then covering the desired locality fully were assigned to separate nodes. The point-to-point transfer of the UAS was implemented by modifying the second node present in the AirSim ROS wrapper named "Simple PID Controller Node" from service node into an action server node waiting for the position goal asynchronously where the

controller parameters proportional gain (K_p) and derivative gain (K_d) were set after heuristic tuning to 0.5, and 2 respectively.

For full coverage of the desired post-disaster region by the UAS, the boustrophedon path [43], as shown in Figure 3.11, was deemed to be the most straightforward and effective option for this work, where the width in each step was selected to be:

$$width = 2 \times Z_{UAS} \times \tan\left(\frac{FOV}{2}\right)$$

where Z_{UAS} is the altitude of the UAS and FOV is the field of view of the camera.

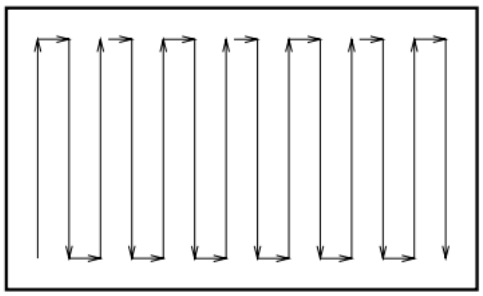

Figure 3.11. Boustrophedon Path.

Here, the boustrophedon path was implemented in a node where the odometry sensor topic was subscribed for current UAS position and the required velocities were published to the velocity command topic in world frame.

Furthermore, an additional path planner node was created for the autonomous systematic execution of both the nodes.

3.4.4 Final Mission Execution

Lastly, the autonomous reconnaissance operations were carried out, where the drone independently takes off, goes to the specified starting point of the desired area, covers the area for predefined mission time, and returns back to land in the initial position. All these actions were executed by the collaboration between the different nodes discussed in the previous sections as shown in Figure 3.12.

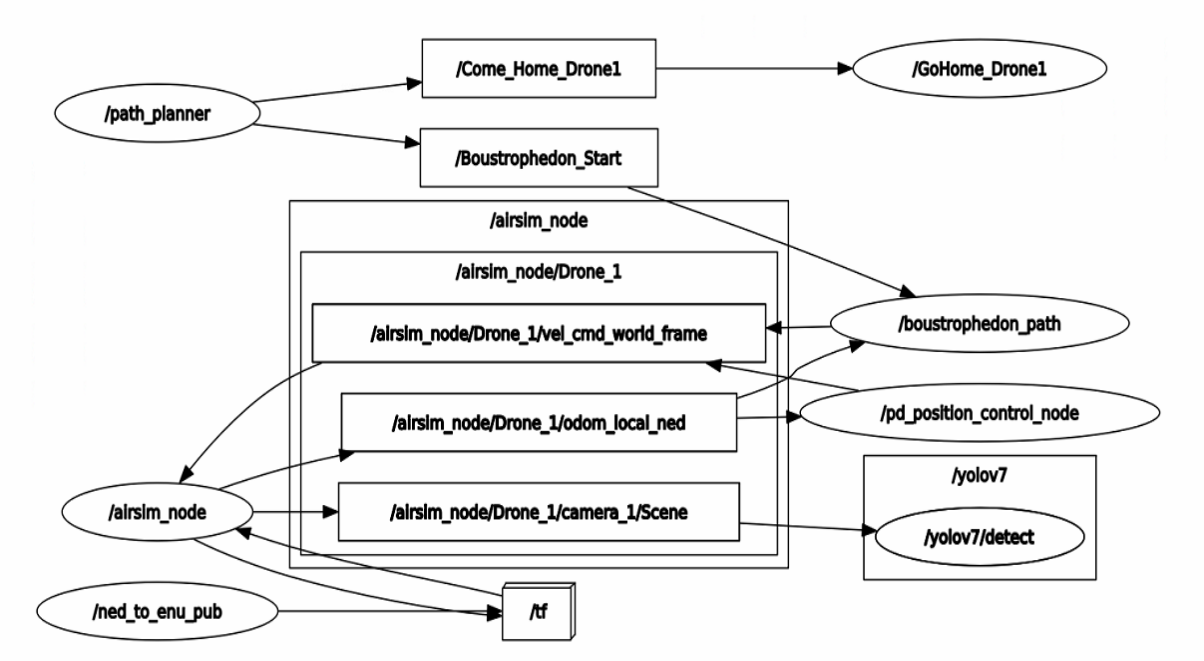

Figure 3.12. Collaboration between the nodes during the mission obtained using "rqt_graph".

Also, using the "image_view" package in ROS, the real-time images with detections published during the mission by the "yolov7_ros" node in the visualization topic were observed and some chosen images at strategic locations containing people were saved by simple right-click of mouse-button for both the models which ensured proper representative sampling for statistical calculation. This was mainly done to limit the number of images gathered for numerical testing due to high frame rate without losing substantial features from the data.

3.5 Results and Discussion

This section presents the results from the statistical metrics calculation of the selected output test images from the real-time detection models along with the relevant discussions.

Table 1 illustrates the flight altitude and camera brightness of the UAS for each model in addition to the number of test images gathered for evaluation. The number of images sampled was lower when using the YOLOv7-SDS because the flight elevation was higher, and as each time the altitude is doubled, the area covered is quadrupled, so less images were needed for the representative analysis of the same area.

Table 1. Overview of the datasets collected and assessed.

Model	Altitude	Brightness	Images
YOLOv7-COCO	2 m	Low	130
YOLOv7-SDS	8 m	Very High	26

The confusion matrix of the detections in the YOLOv7-COCO model can be seen in Table 2. There were 129 objects detected. Of these 129 objects, 110 were correctly detected as "person", 2 were incorrectly classified as another object and 6 people were not detected at all. Therefore, the number of false negatives is equal to 8. That gives an accuracy of 93.79%. Among the correctly detected people, the average confidence level was 84%, the minimum was 27% and the maximum 96%.

Table 2. Confusion Matrix of the detections in the collected images for the Yolov7 model trained on the COCO2017 dataset (YOLOv7-COCO).

		Predicted	
		Person	Not Person
Actual	Person	110	8
	Not Person	0	11

The confusion matrix of the detections in the YOLO-SDS model can be seen in Table 3. There were 142 objects detected. Of these 142 objects, 36 were correctly detected as swimmers, there was 1 incorrectly classified as another object and 2 swimmers were not detected at all. Therefore, the number of false negatives is equal to 3. That gives an accuracy of 97.8%. Among the correctly detected swimmers, the average confidence level was 69%, the minimum was 38% and the maximum 82%.

Table 3. Confusion Matrix of the detections in the collected images for the Yolov7 model trained on the SeaDronesSee dataset (YOLOv7-SDS).

		Predicted	
		Swimmer	Not Swimmer
Actual	Swimmer	36	3
	Not Swimmer	0	103

Now, the first point to be discussed is the very fact that the models trained on real images were able to detect the synthetic objects with high accuracies provides a strong proof-of-concept for the interchangeability of real and virtual SAR missions, justifying the importance of this work.

Secondly, although the accuracies achieved by both models were high, there was a huge difference in the nature of the input images fed into the models. This inspired further contemplation on the working of the deep CNN itself. As, in deep learning the patterns in pixel level from the input images are encoded into the model, so the behavior of the model is dependent on the normalized pixel intensities in the three RGB color channels which is actually the numerical input into the model. With this comprehension, the results from the YOLOv7-model, as seen in Figure 3.14 made complete sense because the patterns in training images of

MS COCO Dataset [39] matched with the pixel-level patterns in the input images due to the adjusted flight altitude of the UAS. But the result from the YOLOv7-SDS, as observed in Figure 3.13, was a surprising discovery which was only possible due to the rigorous experiments with various configurations of altitude and camera settings allowed by this framework. Comparing this result to the images with people in the SeaDronesSee Dataset [41], it could only be hypothesized that the matching of the pixel-level pattern of a human floating in water with stretched hands and legs along with high uniform intensities of the bright pixels triggered the neural network to output "swimmer" class. This intriguing phenomenon needs more research and can be particularly interesting for training with high altitude or satellite images.

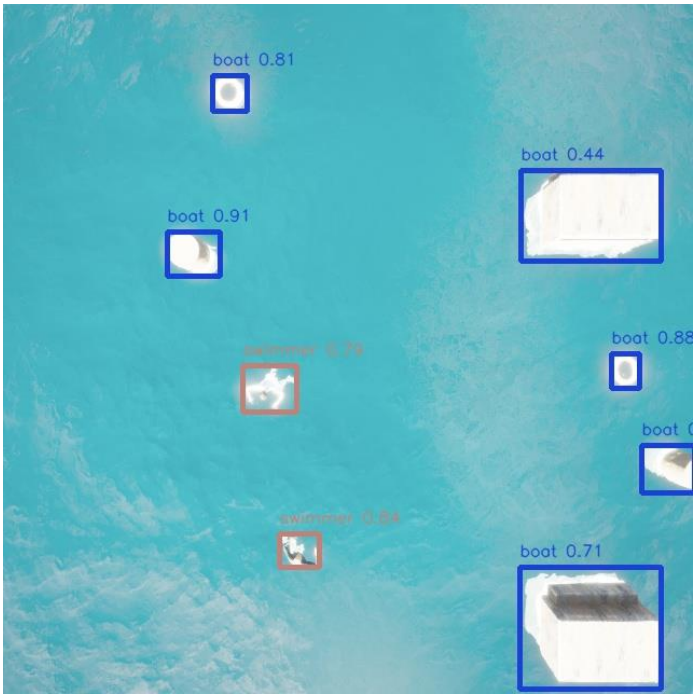

Figure 3.13. Detection with YOLOv7-SDS in mission.

Lastly, another captivating observation highlighting the significance of real-time detection in SAR is that with a high frame rate of input images, the footprint underneath the UAS is always overlapping with the UAS moving forward only small distance in each iteration resulting in multiple chances of detection. So, even if a victim is not detected in one image, there is still a high probability that it might have been detected in the earlier or will be detected in the subsequent image which is clearly exemplified by Figure 3.14.This also results in the actual false omission rate (FOR) being lower than what was observed with the test images.

Figure 3.14. Detections with YOLOv7-COCO in mission.

3.6 Conclusion

In this work, a framework was developed using Unreal Engine, Microsoft AirSim, and ROS that enabled the control of the UAS in a desired way based on velocity commands while detecting objects in an alternating post-disaster ship simulation environment. After that, two pretrained YOLOv7 models were selected: YOLOv7-SDS and YOLOv7-COCO. Then, using the created framework, extensive manual testing was implemented by changing the UAS altitude along with the camera brightness to discover the best possible combination with high average prediction confidence of detecting human victims naturally floating in water with moving hands and legs. This allowed to plan and implement autonomous UAS missions using the same framework yielding high accuracies of real-time victim detection calculated as 97.8% for YOLOv7-SDS when the UAS was deployed in an altitude of 8m with high camera brightness, and 93.79% for YOLOv7-COCO when the UAS was employed at a height of 2m from the sea-level with lower camera brightness.

Furthermore, the developed framework has immense potential for further work. Due to the limitation of time and difficulty of accommodating all the things in a single paper, only a few experiments were carried out in this study. But extensive experimentations will be performed in the near future with various object detection models in different configurations accompanied by the implementation of the findings physically with a real UAS. Additionally, other researchers are also encouraged to utilize the detailed steps of reproducing the framework fabricated in this work to carry out experiments according to their respective needs.

Moreover, a small modification in the research question proposed in this work to "How to make *any* configuration of the UAS and object detection models to work together in real-time with optimal accuracy of detecting *any* desired object in a dynamic marine environment?" suddenly demystifies the true potential of this framework. By reverse engineering the images in the UAS camera to have the precise automatic ground truth labeling from the gaming engine, the same framework has the capability of training new object detection models in real-time in different

imaginable configurations for any thinkable objects overcoming all the traditional challenges in maritime computer vision.

Acknowledgment:

This paper is supported by European Union's Horizon 2020 research and innovation programme under grant agreement No. 101020676 for the project named as VALKYRIES (Harmonization and Pre-Standardization of Equipment, Training and Tactical Coordinated procedures for First Aid Vehicles deployment on European multi-victim Disasters).

4 Additional work on the paper

This chapter describes the additional work carried out in accordance with the paper with a primary objective to answer the research question of how to handle the dynamic marine environment where people and objects cannot remain in the same place due to the nature of open sea water.

4.1 Overall Process Diagram

The block diagram, as shown in Figure 4.1, briefly illustrates the overall process flow for this chapter which will be explained in detail in the later sections.

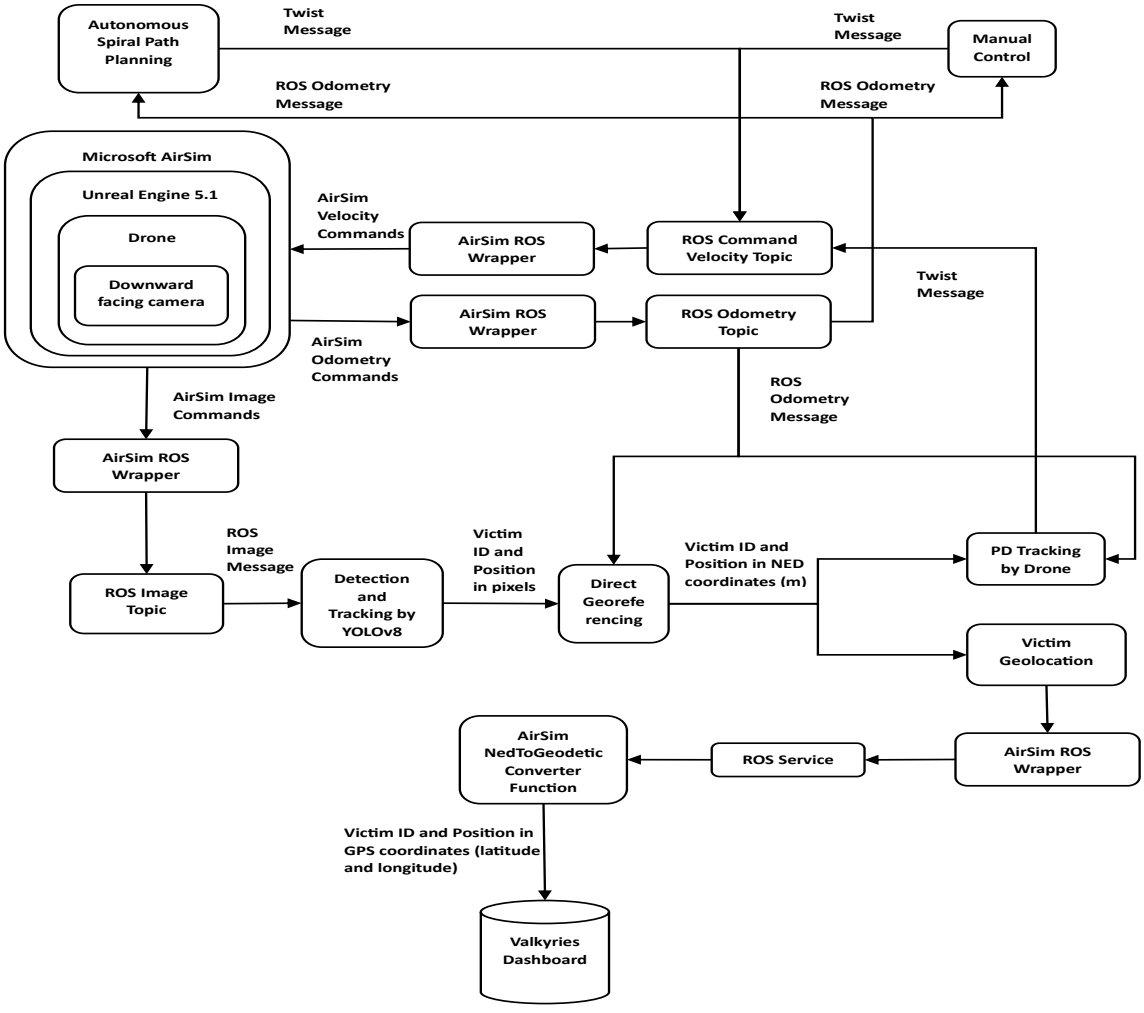

Figure 4.1. Block Diagram of the Overall Process Flow

4.2 Custom Virtual Environment in Unreal Engine 5.1

This section describes the steps followed for the creation of a new custom Sea Simulation Environment from scratch using the currently latest fully released version 5.1 of Unreal Engine (UE) and Microsoft AirSim. This was carried out to further explore the potential of the built-in features of UE5.1 as a knowledge building step.

4.2.1 Setting up the initial Sea Environment

Here, the primary steps involved in setting up a natural looking simulation environment with sky and sea are explained. Starting with a new *blank blueprint project* with *starter content* selected in the initial user interface of UE5.1, firstly, a new *empty level* was created, which was completely empty and dark, and was saved as *Environment* in *Maps* folder. Then, from the *Place Actors* panel, firstly, *Sky Atmosphere* component was placed into the *Environment* from the *Visual effects* category, followed by the *Directional Light* component from *Light* category, which was renamed as *Sunlight*. Next, again from the *Visual effects* category, two components *Volumetric Cloud* and *Exponential Height Fog* were added into the *Environment*, ending with *Skylight* component from the *Light* category with *Real Time Capture* property checked for better lighting. Consequently, the *Environment* level contained a realistic looking sky atmosphere with blue sky, movable sun, clouds, and a negligible amount of fog, whose settings could be modified according to the desire of the developer.

Furthermore, for the placement of sea in the *Environment*, in the beginning two built-in plugins named *Water* and *Landmass* were enabled. Then, changing the viewing mode from *Perspective* to *Landmass*, the size of the landmass was selected sufficiently large in *Quads* with *Enable Edit Layers* property checked, which was necessary for *Water* plugin to work because the water body was built over the landmass, and leaving other properties unchanged. After that, again switching to *Perspective* viewing mode, from the *Place Actors* panel, *Water Body Ocean* component was added into the *Environment*, which spawned the sea into the *Environment*, and with it selected, the *spline points* in white color were made adequately small to cover up the default formation of a small island in the *Environment*. Finally, after the completion of all these steps, a simulation environment was developed with sky and empty sea.

4.2.2 Transfer of Assets

The different useful assets like *Oil Tanker*, *Buoys*, and *six people* from the previously developed environment in UE4.27 as in Section 3.3.1, were migrated to this project in UE5.1. The *Ship* blueprint developed earlier by combining different static meshes parts, was directly placed in the *Environment*, and *Particle Effects* of *Fire* from *Starter Content* were added to simulate its accident. However, some modification had to be made for different other assets that required buoyancy for floating like containers, oil barrels, buoys, and people because the *Environment Project* as in Section 3.3.1.4 had its own buoyancy component named *Buoyant Force*, but for the custom environment the unreal engine had its own default *Buoyancy* component. Nevertheless, the differences were subtle with primarily just the change in name

for the points where buoyancy was applied from *Test Points* to *Pontoons* as shown in Figure 4.2 for oil barrel, and similarly for other static meshes.

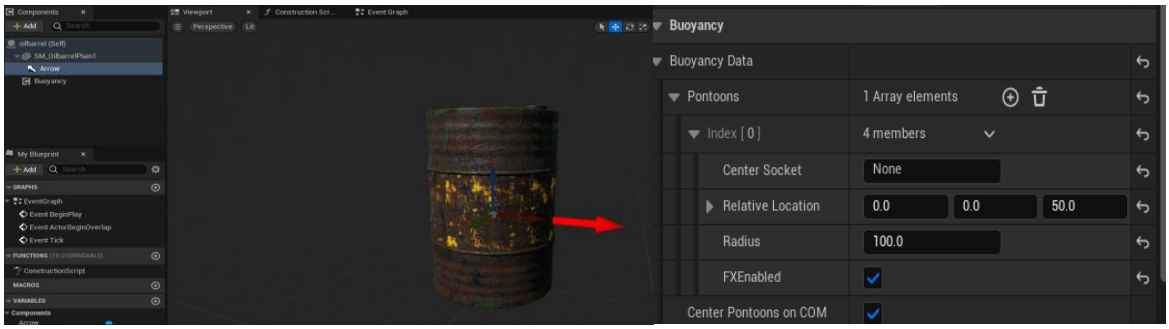

Figure 4.2. Blueprint for Oil Barrel with *Buoyancy* component

Likewise, some adjustments were also made to simulate treading people from previous work in Section 3.3.1, by wrapping the character into a blueprint class that is a child of *Character* class, which gives additional components like *capsule collision cylinder*, *skeletal mesh*, *character movement* by default because it was later found in further study that this was the standard practice in Game Development used for controlling character as shown in Figure 4.3.

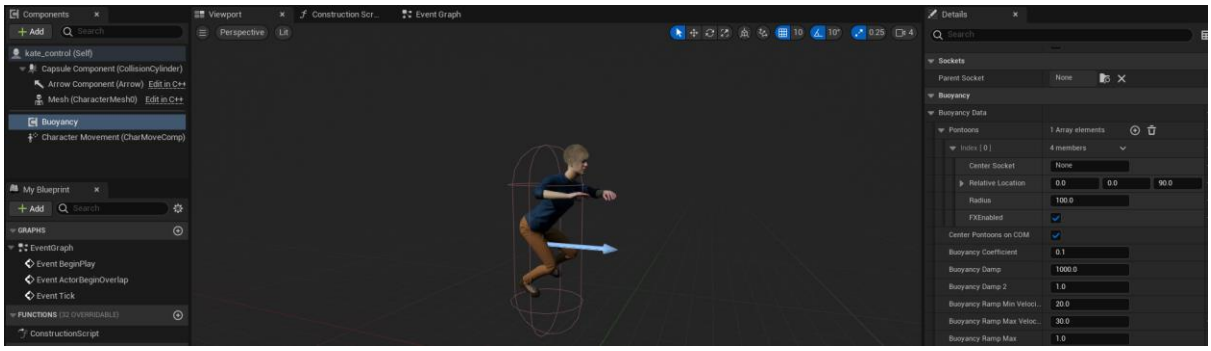

Figure 4.3. Character Blueprint with Buoyancy added.

The desired skeletal mesh of the person, and the existing animation asset of *Treading* was selected in the *Mesh* component. Previously, the physics was enabled directly for the skeletal mesh itself, by also overriding its physics asset to *SK_Mannequin_PhysicsAsset* as in Section 3.3.1.4, which was found after extensive trial and error to make the character fall normally without any unusual behavior. But here, the physics was enabled for the *capsule component* by checking the *Simulate Physics* and *Enable gravity* property which made the character experience gravity without directly changing any settings of the skeletal mesh as it was a child of the capsule component. Also, only a single *Pontoon* point was added in *Buoyancy* component with other properties like *Radius* left to default values as shown in Figure 4.3. As, it was enough to make the person naturally float on water with treading action, which was the elementary concern for this study as shown in Figure 4.4.

Figure 4.4. Character treading in Sea with buoyancy

4.2.3 Swimming People

As the people normally floating in water with moving hands and legs i.e., treading, had already been implemented in the paper, the next step was to implement swimming victims in water. For that, firstly, swimming animation for the character was downloaded from Mixamo [37] and imported into the project. But people cannot directly start swimming or continuously only swim forever. Taking this into account, animation blendspace was used that helps for transition of animation from treading to swimming based on the value of arbitrarily chosen parameter. In this case, the speed of the swimmer is chosen as that parameter, which is a natural phenomenon also. One of six characters, named *Leonard*, was chosen initially to update it into a swimmer, and a *1D Animation Blendspace* component from *Animation* option when right clicked in the content browser of UE5.1 was created as shown in Figure 4.5.

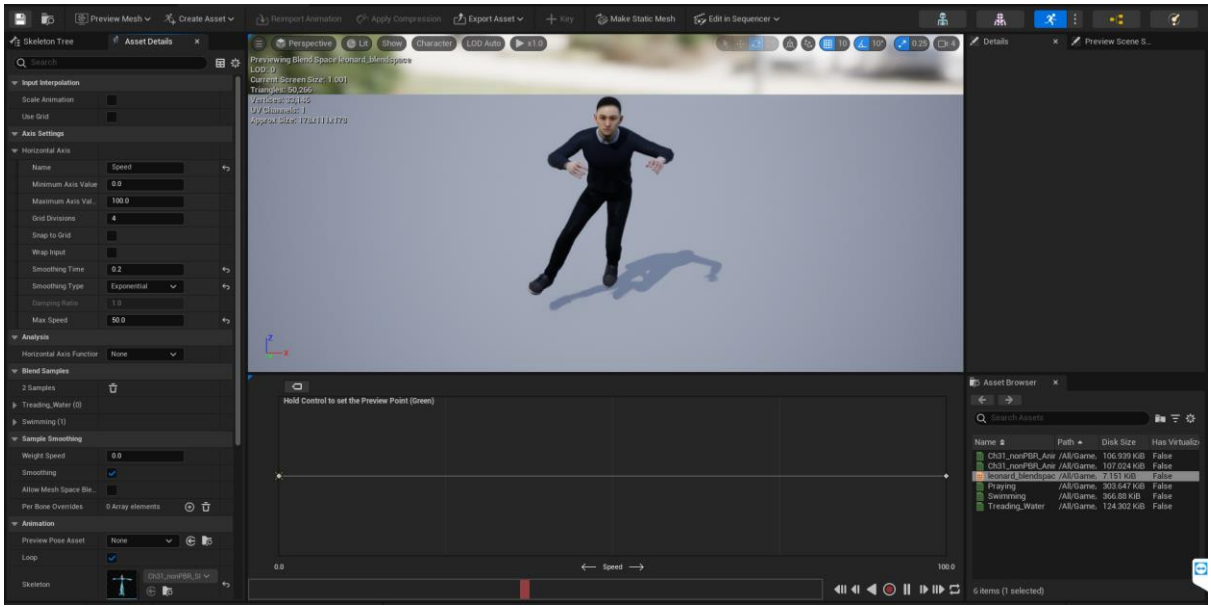

Figure 4.5. 1D Animation Blendspace with Treading Animation

As seen in *Axis Settings* in Figure 4.5, *Speed* parameter was kept in the *Horizontal Axis* with *Minimum Axis Value* as 0.0 and *Maximum Axis Value* as 100.0. Then, the *Treading Animation*

was dragged from the *Asset Browser* in the lower right section to the leftmost part of the timeline with symbol of “x”, and *Swimming Animation* was dragged to the rightmost part as shown in Figure 4.6.

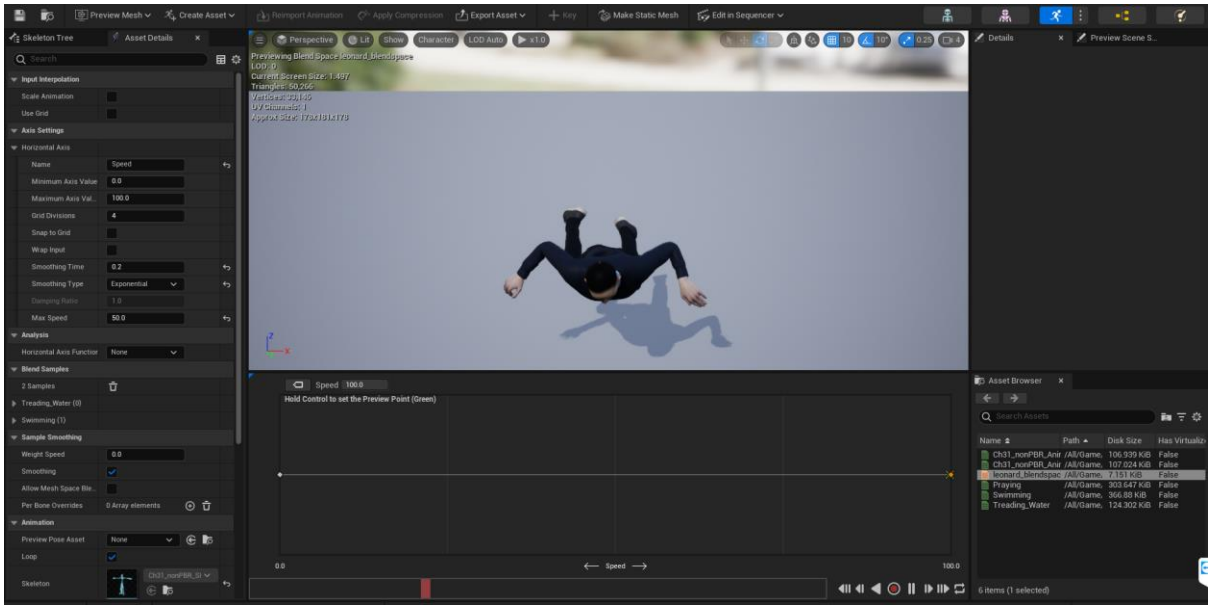

Figure 4.6. 1D Animation Blendspace with Swimming Animation

Hence, the animation slowly changes with the increasing input value of *Speed* from 0.0 to 100.0. The developed animation blendspace was then used to create an *Animation Blueprint*, which could be used inside the blueprint of the character. In the *Event Graph* of the *Animation blueprint*, the current speed of the character was taken from the *Get Component Velocity* function and set to a float variable named *Speed* as depicted in Figure 4.7.

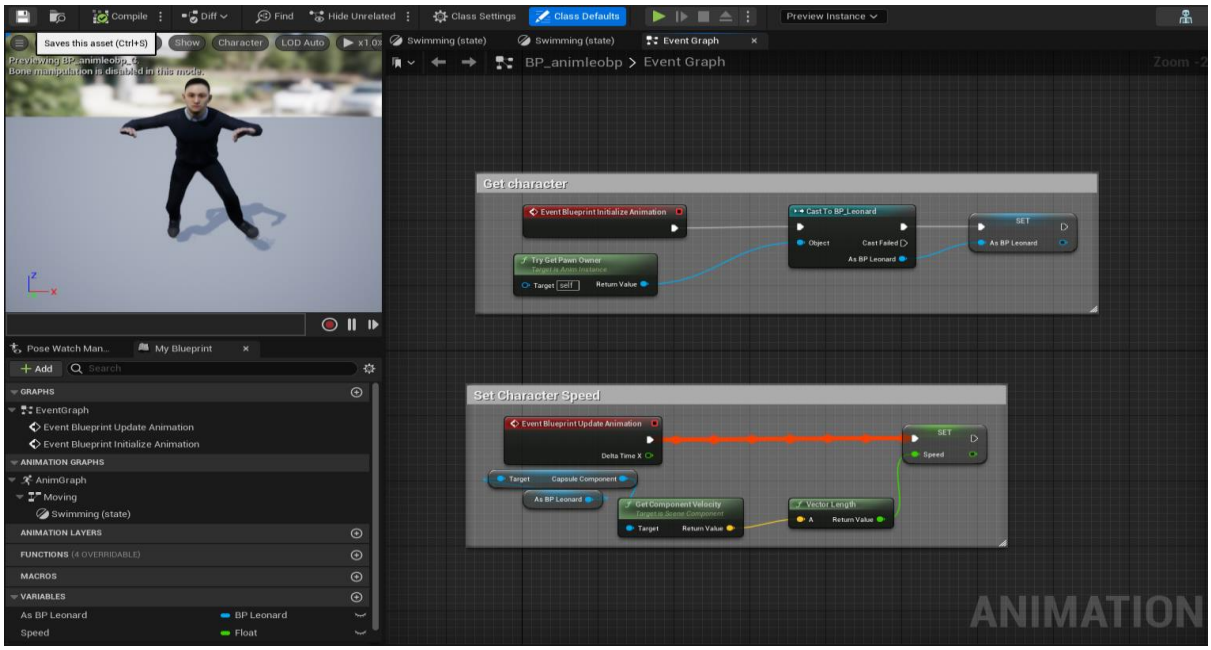

Figure 4.7. Animation Blueprint Event Graph

Then, that *Speed* variable was used as input to the blendspace, and the output of the blendspace was connected to *Output Animation Pose* node of the animation blueprint that directs the current animation for the character as illustrated in Figure 4.8.

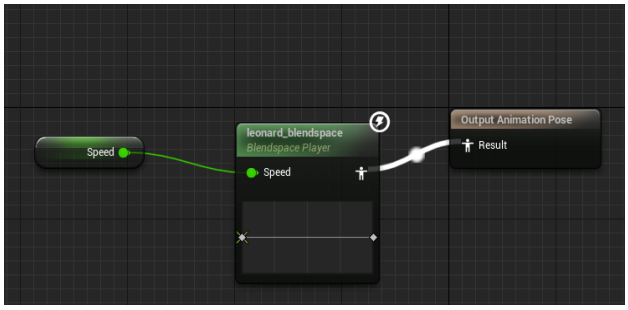

Figure 4.8. Animation Blendspace to Output Animation Pose in Animation Blueprint

Now, for the actual swimming action of the character, firstly, an idea was brainstormed to make the character swim continuously in a square pattern for the easiness of the testing purposes later. For that, a custom event *Trigger swim* was created inside the character blueprint to trigger the start of the movement as shown in Figure 4.9.

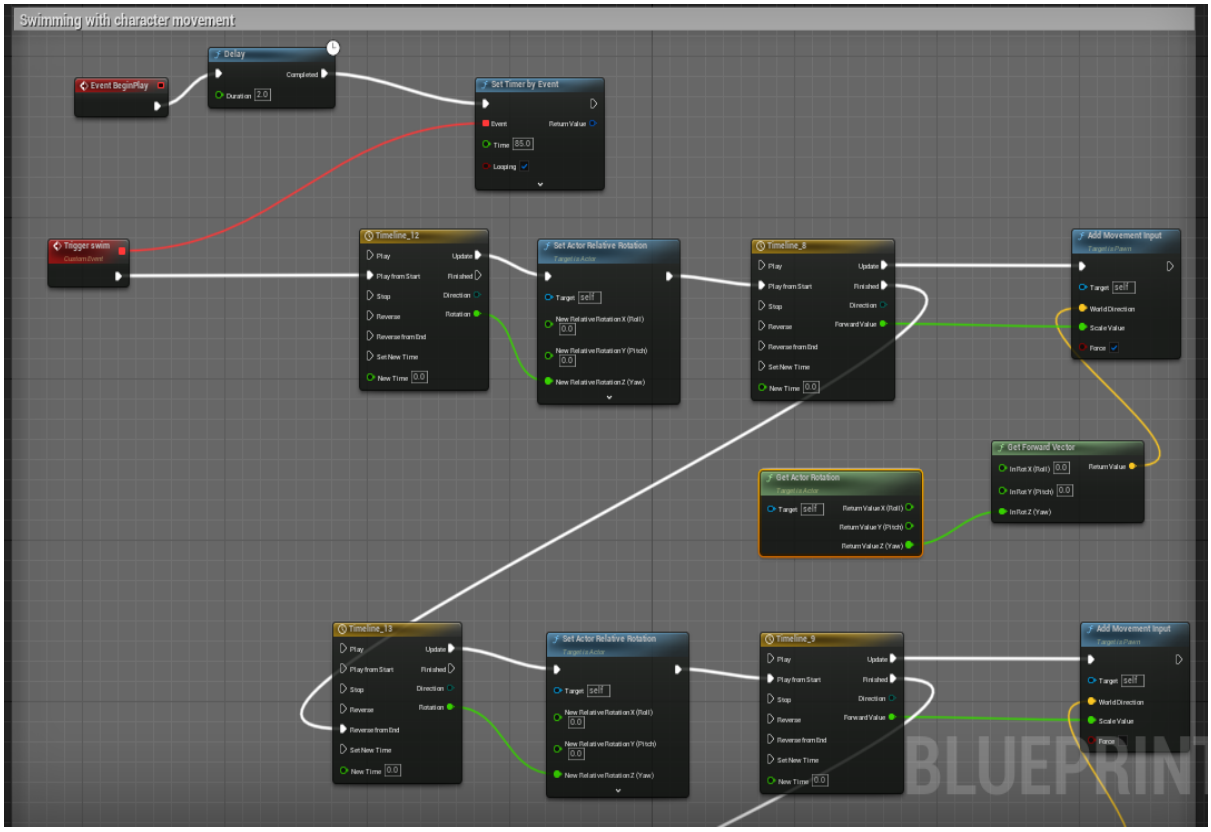

Figure 4.9. Event Graph for Character Leonard Swimming in a square pattern

For the control of time and value, different *Timeline* nodes were used. The first two will be explained as an example, with a similar idea for the rest. The first timeline node *Timeline_12*, as shown in Figure 4.10, was of short duration of 2 seconds, that was used to output smooth continuous value from 0.0 to 90.0 to *Set Actor Relative Rotation* function node which rotates the character to the right by 90°.

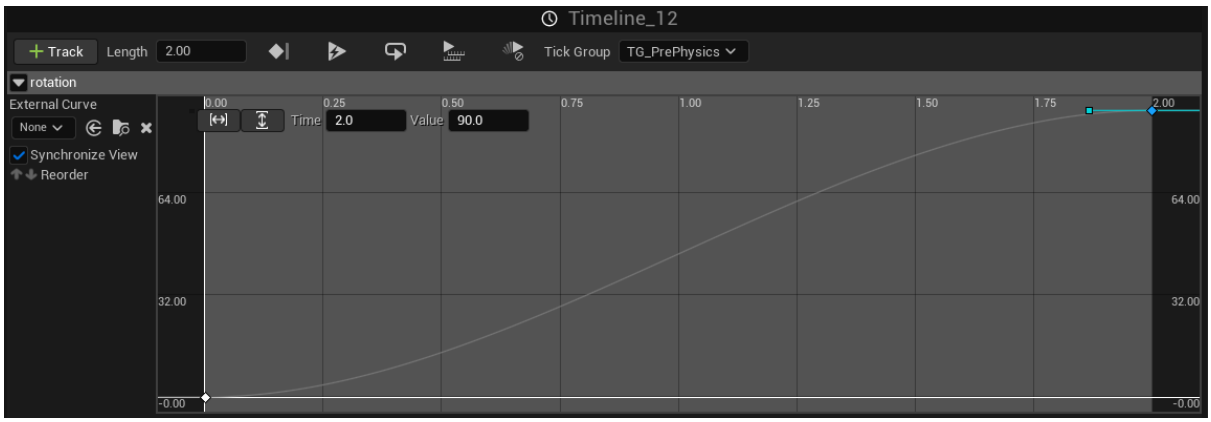

Figure 4.10. Timeline Node outputting values from 0.0 to 90.0 in 2 seconds

Then, the second timeline node *Timeline_8*, as illustrated in Figure 4.11, was used for producing values between 0.0 and 0.9 between a duration of 20 seconds, which was fed into the *Add Movement Input* function node for making the character slowly swim forward in the current relative z or yaw direction that was found by extracting the forward vector from the z rotation value given by *Get Actor Rotation* function node as portrayed in Figure 4.9.

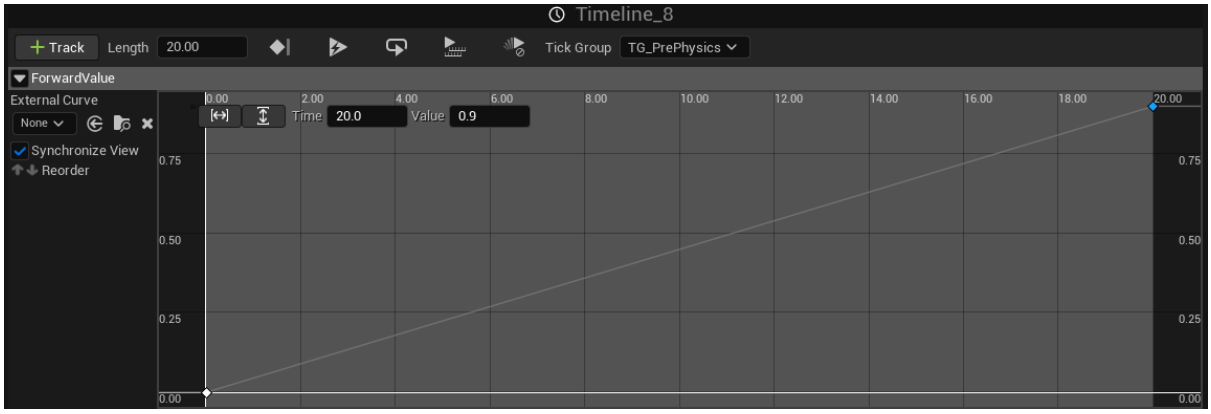

Figure 4.11. Timeline Node outputting values from 0.0 to 0.9 over the duration of 20 seconds

Similarly, when the character swam for 20 seconds, then the relative rotation was adjusted to follow the square pattern, and again the character swam for another 20 seconds. Hence, a total of 4 timeline nodes of 2 second duration were used for setting the relative rotation smoothly in each phase, and another 4 timeline nodes of 20 second duration for the swimming movement input. Then, as illustrated in Figure 4.9, *Set Timer by Event Handle* node was used to again trigger the event Trigger swim automatically after the end for making the character continuously swim in a square pattern.

Another important point to note is that before the character starts swimming, the *Physics Volume* of the *Movement Component* was set as *Water Volume*. The *Maximum Swimming Speed* property of the Character was set to 100 cm/s in the *Character Movement* component in accordance with the *Speed* parameter set in the *Animation blendspace* as in Figure 4.5. And the maximum value to be produced by the timeline node was set to 0.9 to not make the character swim in a fully horizontal way inside water, but with a small angle to the horizontal.

Likewise, another idea was formulated that a continuously forward swimming person would also be required during the intermediate testing of the tracking by drone. Therefore, another character named *Pete* was chosen to be a continuous swimmer. The event graph of the continuous swimmer is shown in Figure 4.12. Here, a single timeline node was used for a duration of 5 seconds and maximum value of 0.9 with Loop option turned on.

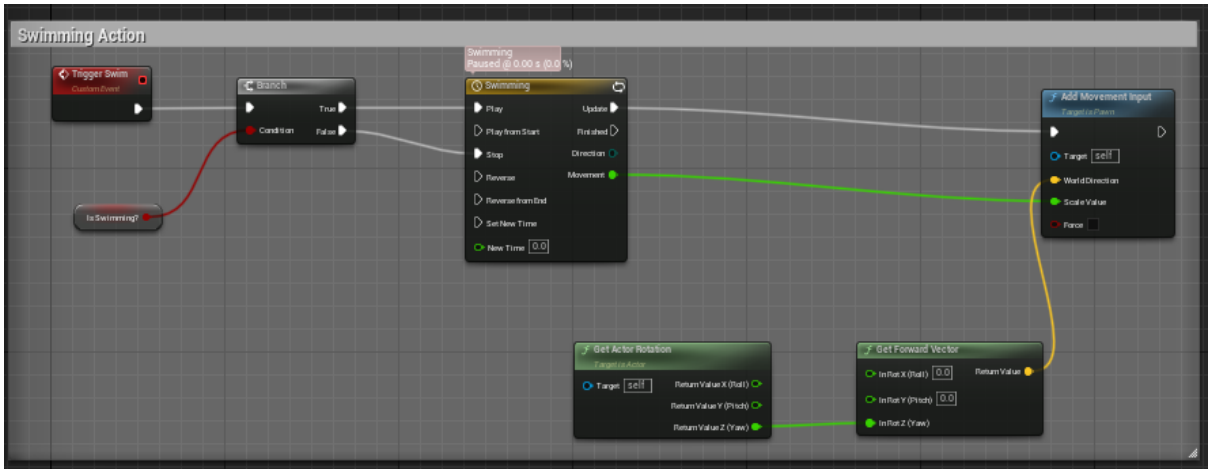

Figure 4.12. Event Graph for Character Pete swimming forward continuously

4.2.4 Controllable Speed Boat

In the previous work Section 3.3.2, the boat where drone was placed and launched was available in the *Environment Project* as in Section 3.3.1.1 and was just floating on water. So, it was decided to try to move the boat also autonomously to the desired location in this *Environment* i.e., victims in the current context. For this, a new asset named *Speed Boat Packs* was bought from the *Unreal Engine Marketplace* which was supported for UE version 5.1. It contained many boat types among which a *Speed Boat* as shown in Figure 4.13 was chosen for this study. Firstly, a new Blueprint Class named *BP_SpeedBoatControlled* was created as a child of default *Pawn* Class. Then, all the static meshes available in the *Speed Boat* asset were put together as children of the main boat body static mesh in the blueprint as illustrated in Figure 4.14. After that, six *sockets* were added at different locations in the static mesh of the boat where the *Pontoons* could act to provide buoyancy as depicted in Figure 4.13.

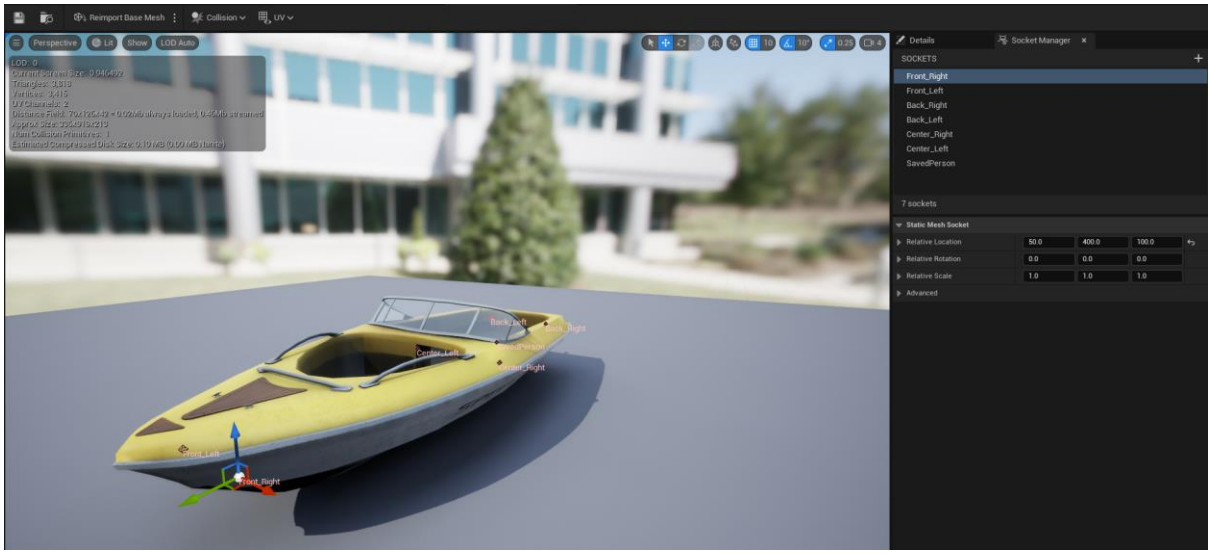

Figure 4.13. Sockets added to the static mesh of the chosen Speed Boat

Next, the *Buoyancy* component was added to the blueprint and the *Pontoons* were attached to the *sockets* by the *Center Socket* property of the *Pontoon* as shown in Figure 4.14.

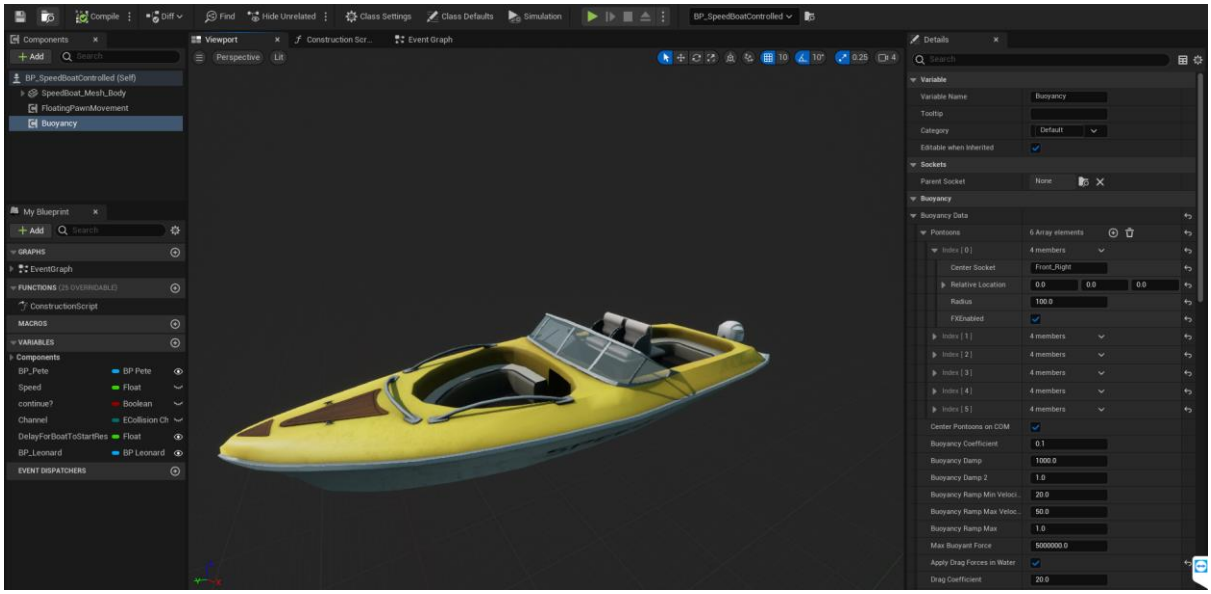

Figure 4.14. Blueprint for the Speed Boat with Buoyancy component

For controllable autonomous driving of the boat, a custom event *Move To* was created in the *Event Graph* as illustrated in Figure 4.15. The Figure 4.15 contains two contiguous parts on top of one another to accommodate inside the page and maintain the readability of the blueprint code.

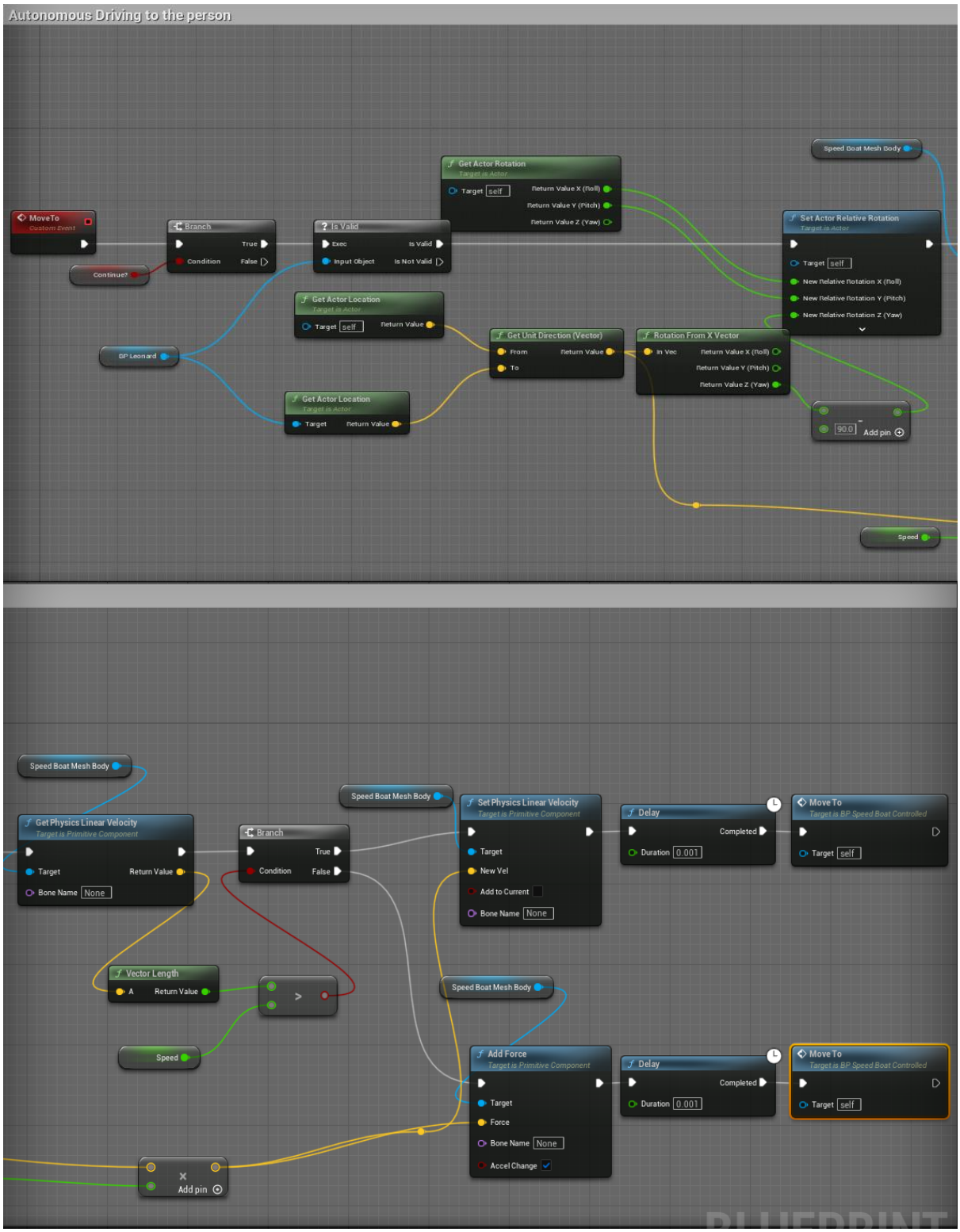

Figure 4.15. Custom *Move to* Event inside *Speed Boat* to autonomously drive to selected person.

Here, firstly, the location of the blueprint character *BP_Leonard* was obtained by *Get Actor Location* function node, then the unit direction vector between the boat and the character was calculated using *Get Unit Direction Vector* function node. As, the direction was calculated from the center of the boat, 90° was subtracted in yaw direction of the output of *Rotation from X Vector* function node, which gave the rotation difference from x-direction, to make the boat point straight in the direction of the person by setting the new rotation in z or yaw axis using *Set Actor Relative Rotation* function node. Also, the unit direction vector was multiplied by the user specified *Speed* variable with *Instance editable* property checked that enabled editing it in the *Level Editor*, to input it into the *Add Force* function node. Moreover, to limit the maximum speed of the boat, the current velocity of the boat obtained from *Get Physics Linear Velocity* function node was checked whether it was higher or lower than the *Speed* set by the user, and only if it was lower the input to *Add Force* function node was allowed, otherwise the current velocity of the boat was clamped to the *Speed* set by using *Set Physics Linear Velocity* function node. Then, finally in the end again the *Move To* custom event was triggered after a small *Delay* to loop continuously until the destination was reached as shown in Figure 4.15.

Also, another custom event titled *TriggerMoveTo* was fabricated to trigger the *Move To* event after some user specified delay through a float variable named *Delay for Boat to Start Rescue* whose Instance Editable property was turned on for the ease of changing the value from the *Level Editor* as portrayed in Figure 4.16.

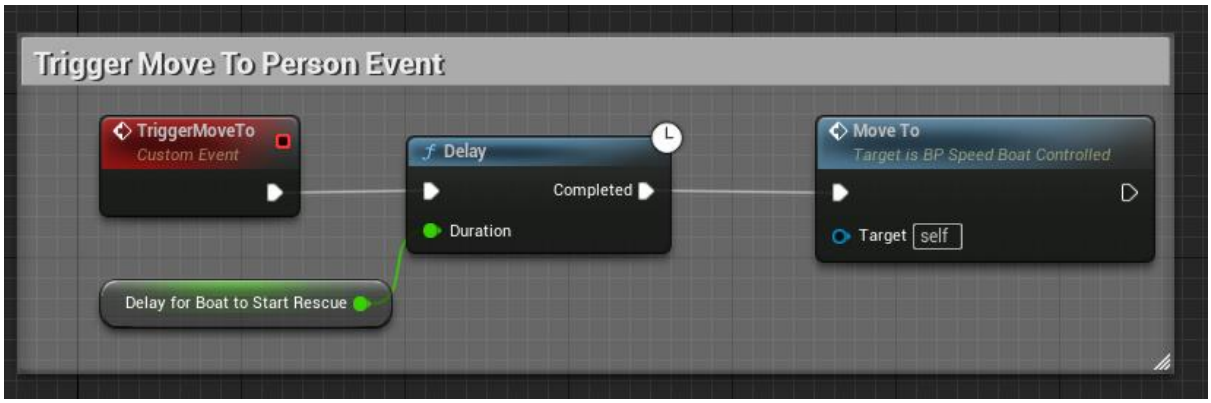

Figure 4.16. *TriggerMoveTo* custom event inside *BP_SpeedBoatControlled*

Furthermore, an additional custom event named *OnBoardPerson* was also created to attach the person onto the boat as shown in Figure 4.17. For that, initially an extra *socket* titled *SavedPerson* was added onto the static mesh of the boat in the appropriate empty location in the boat as seen in Figure 4.13. Then, a *Skeletal Mesh* component named *SavedPerson* was added as a child of the main body static mesh of the boat. Moreover, a new animation asset called *Praying* was also downloaded from Mixamo [37] and imported into the engine to simulate the action of the saved victim thanking for the rescue. The *OnBoardPerson* event was made to take as input the desired *Skeletal Mesh* component which was set into the *SavedPerson* component using *Set Skeletal Mesh Asset* function node. Then, using the *Set Animation Mode* and *Set Animation* function nodes, the animation asset of the *SavedPerson* component was set to *Praying*. And, finally using the *Attach Component to Component* function node, as shown in Figure 4.17, the *SavedPerson* component could be attached with the body mesh of the speed boat in the location of *SavedPerson socket*.

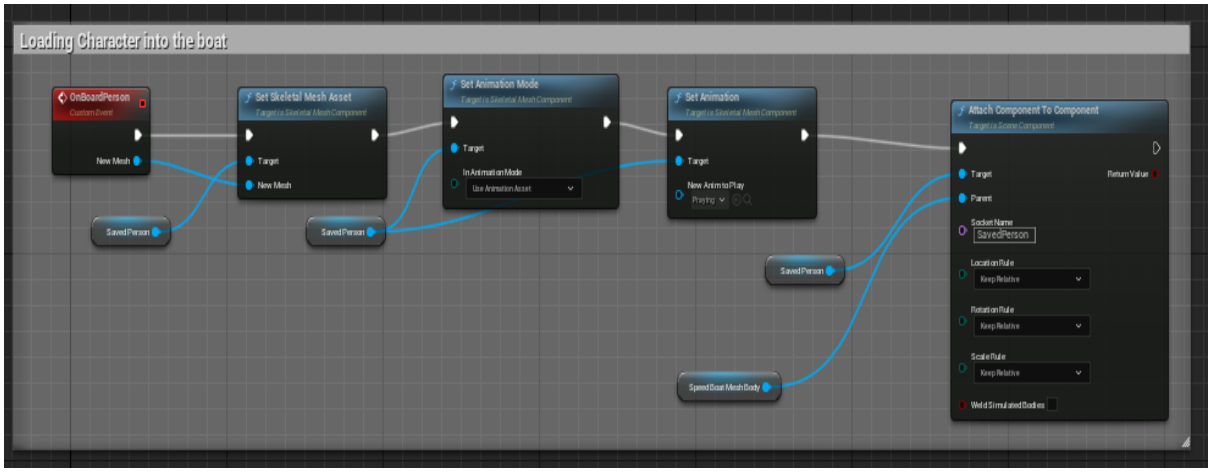

Figure 4.17. *OnBoardPerson* custom event inside *BP_SpeedBoatControlled*

4.2.5 Autonomous Rescue of the Victim by the Speed boat

Next, the triggering point for the *Move To* event in the speed boat was considered to be the moment when the drone reaches the vicinity of the victim. For that a *Box Collision* component named *boxcollision2* was added into the blueprint of the character Leonard *BP_Leonard* as illustrated in Figure 4.18. The *boxcollision2* component was scaled in all directions but by higher amount in the Z-direction to account for the drone flying up in the air.

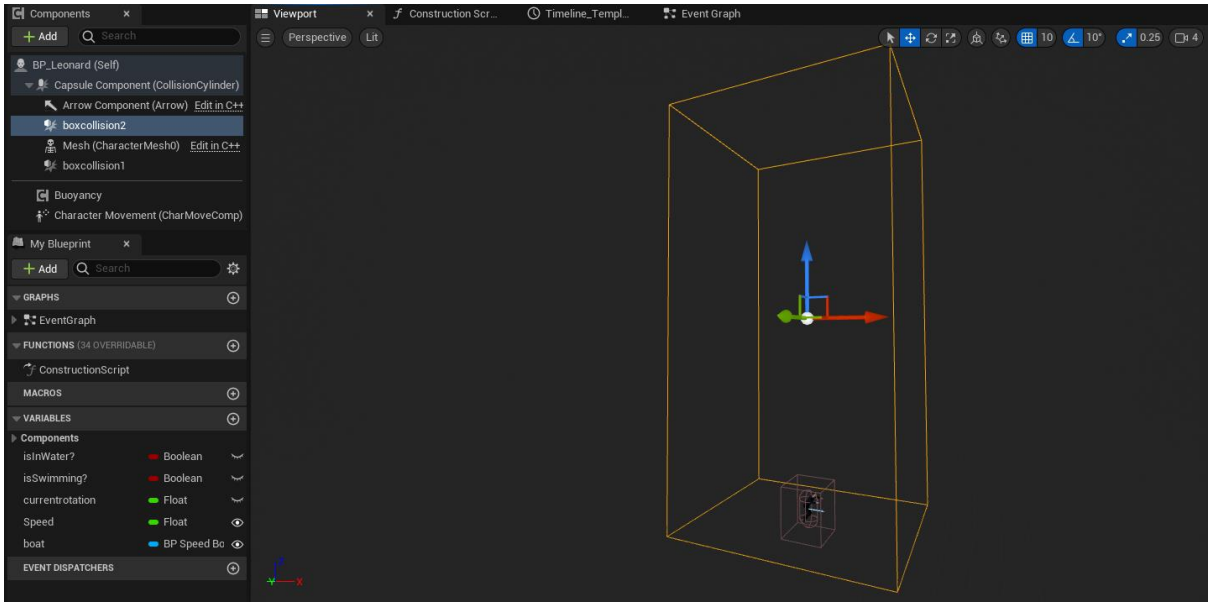

Figure 4.18. Addition of box collision components in *BP_Leonard*

Then, in the *Event Graph* of *BP_Leonard* a default collision detection event was created *On Component Begin Overlap (boxcollision2)* that detected the collision between the *boxcollision2* and the drone as shown in Figure 4.19. The *Get Player Pawn* function node was used to get the reference of the blueprint class of the AirSim Drone *BP_FlyingPawn*. When the collision was

detected inside *boxcollision2*, it was checked whether the *Other Actor* was the drone, and if it was *True* then the custom event inside the boat titled *Trigger Move To* was activated which in turn started the movement of the boat towards the person by triggering the *Move To* event after some delay.

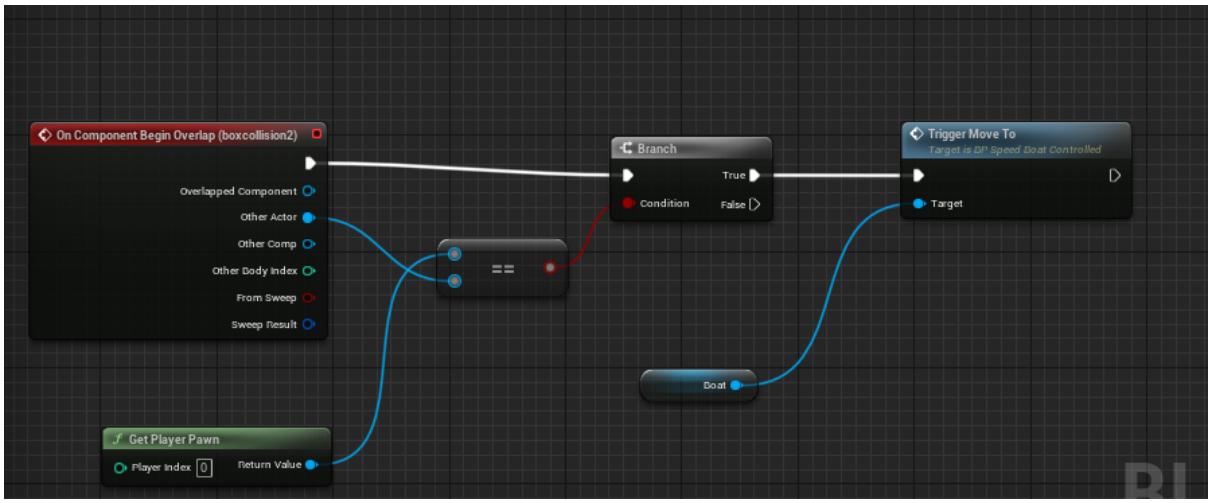

Figure 4.19. Collision event for boxcollision2 and the drone

Now, with the setup of triggering the boat towards the person completed, next step was to on board the victim onto the boat when the boat reached near it. For that, another *Box Collision* component named *boxcollision1* was also added in BP_Leonard as seen in Figure 4.18. Similarly, with the drone, in the *Event Graph* of *BP_Leonard* a default collision detection event was created *On Component Begin Overlap (boxcollision1)* that detected the collision between the *boxcollision1* and the boat as shown in Figure 4.20. Firstly, *Cast to BP_SpeedBoatControlled* node checked whether the *Other Actor* was *BP_SpeedBoatControlled*, and if it was *valid* then the *OnBoardPerson* custom event was triggered with the skeletal mesh of Leonard character also sent as input. It simulated the person loaded onto the boat when the boat reached near the victim.

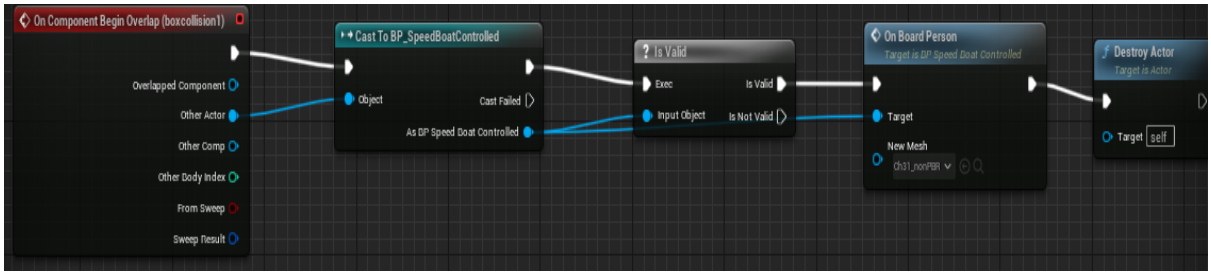

Figure 4.20. Collision event for boxcollision1 and the boat

4.2.6 Initial Setup of the Drone using AirSim

Following the similar procedures as in Section 3.3.2, with a difference in the use of the AirSim version to [9] developed by Codex Laboratories LLC to work with Unreal Engine version 5.1, the drone was initially placed in the developed *Environment* as shown in Figure 4.21. Furthermore, the *origin geopoint* parameter in the AirSim settings [7] was set to *Latitude* of 57.963589°, *Longitude* of 9.130108°, and *Altitude* of 122 as seen in Appendix H. And it is important to note that when the *Environment* is started in *AirSim Gamemode*, the origin point both in geographical coordinates and NED coordinates starts from the initial spawning point of the drone. And, as the drone was kept on top of the boat, the difference in height from the starting point of the drone and the sea level was found to be 3.28366 m that was set as *z_correction* used in the georeferencing calculation later. The gimbal location was set in the middle point of the drone without any offset in this study, so it was ignored. But any difference in camera height known can be adjusted to *z_correction*.

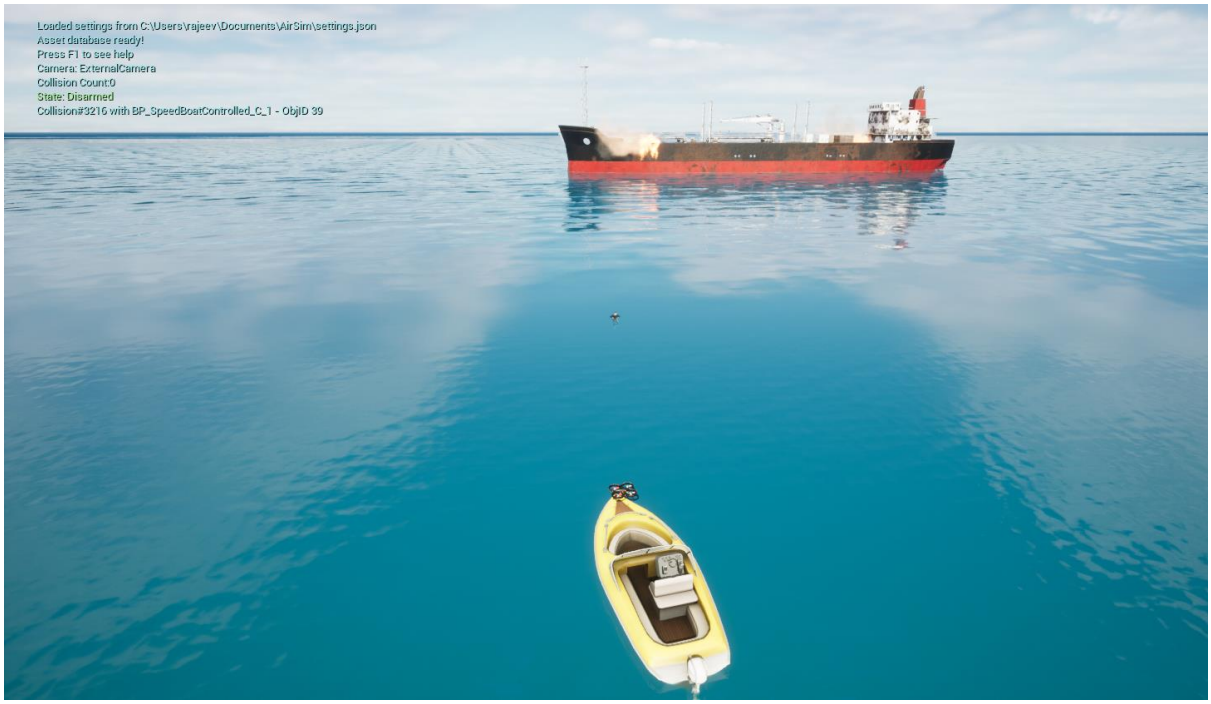

Figure 4.21. Initial Setup of the Drone in the Environment

4.3 Detection and Tracking by YOLOv8

Based on the official GitHub repository [44] and [45], YOLOv8 was implemented in ROS in a separate node named *yolov8_sea* whose complete code implementation is in Appendix E. Basically, the node *yolov8_sea* subscribed the image topic */airsim_node/Drone_1/camera_1/Scene* from the AirSim Drone, and the model was implemented in the *image callback* of the subscriber for Real-Time Detection. Here, both the object detection and segmentation models of YOLOv8 were implemented in the same node with a *Boolean flag* for selection amongst the two models. Furthermore, the center and bounding box pixel coordinates along with the tracking ID of the detections for each image were published as *Detection2DArray* message in the topic titled */detections*. Also, the detection image, with bounding boxes, class name of the objects, track ID, and confidence score drawn onto it, was displayed in an OpenCV window for real-time viewing as well as published as *sensor_msgs.msg. Image* message in the topic */detection_image* for remote viewing from anywhere. This allowed for the experimentation as in Section 3.4.2. For the current context, the drone was manually flown at different heights and the detection results along with the corresponding height were viewed in real-time, as shown in Figure 4.22, and it was observed that the YOLOv8 pretrained model detects the person treading and swimming in Sea with high confidence score till around 6.5m from the sea level. The model detected the person perfectly in any pose till that height, so no other configurations were tested for this study.

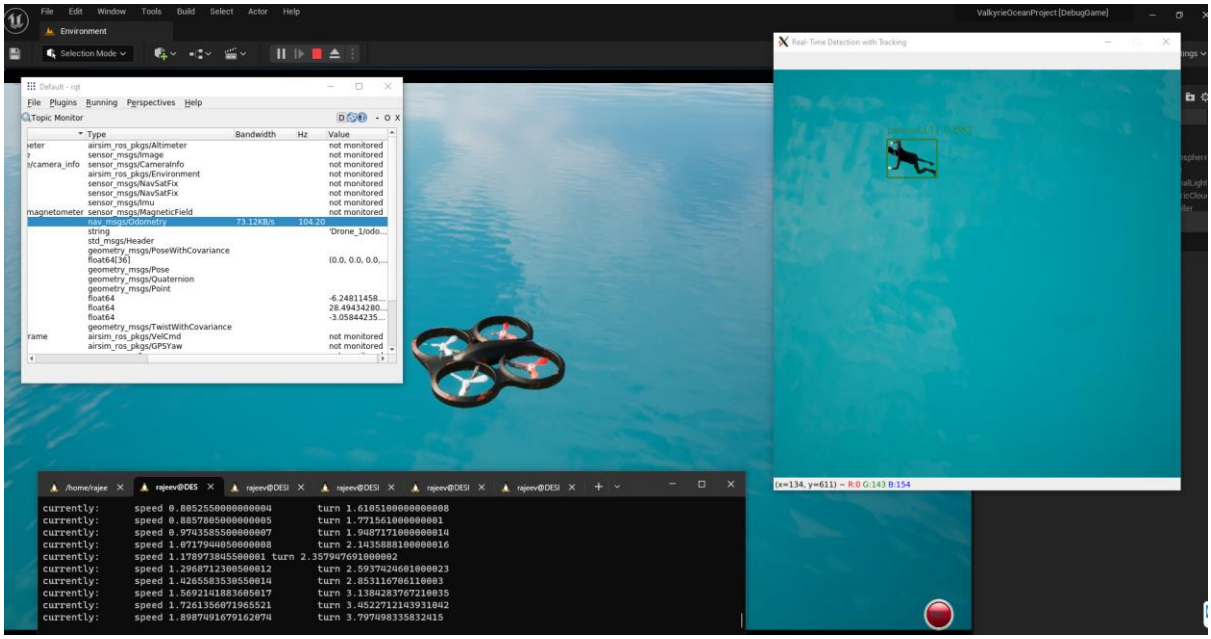

Figure 4.22. Experimentation with YOLOv8 pretrained model by manually flying the drone.

4.4 Autonomous Archimedean Spiral Path Planning

Based on the theory explained in Section 2.4, Autonomous Archimedean Spiral Path Planning was implemented on separate node named *Archimedean_Spiral_Path* in ROS whose complete code implementation is included in Appendix D, and its visualization is shown in Figure 4.23.

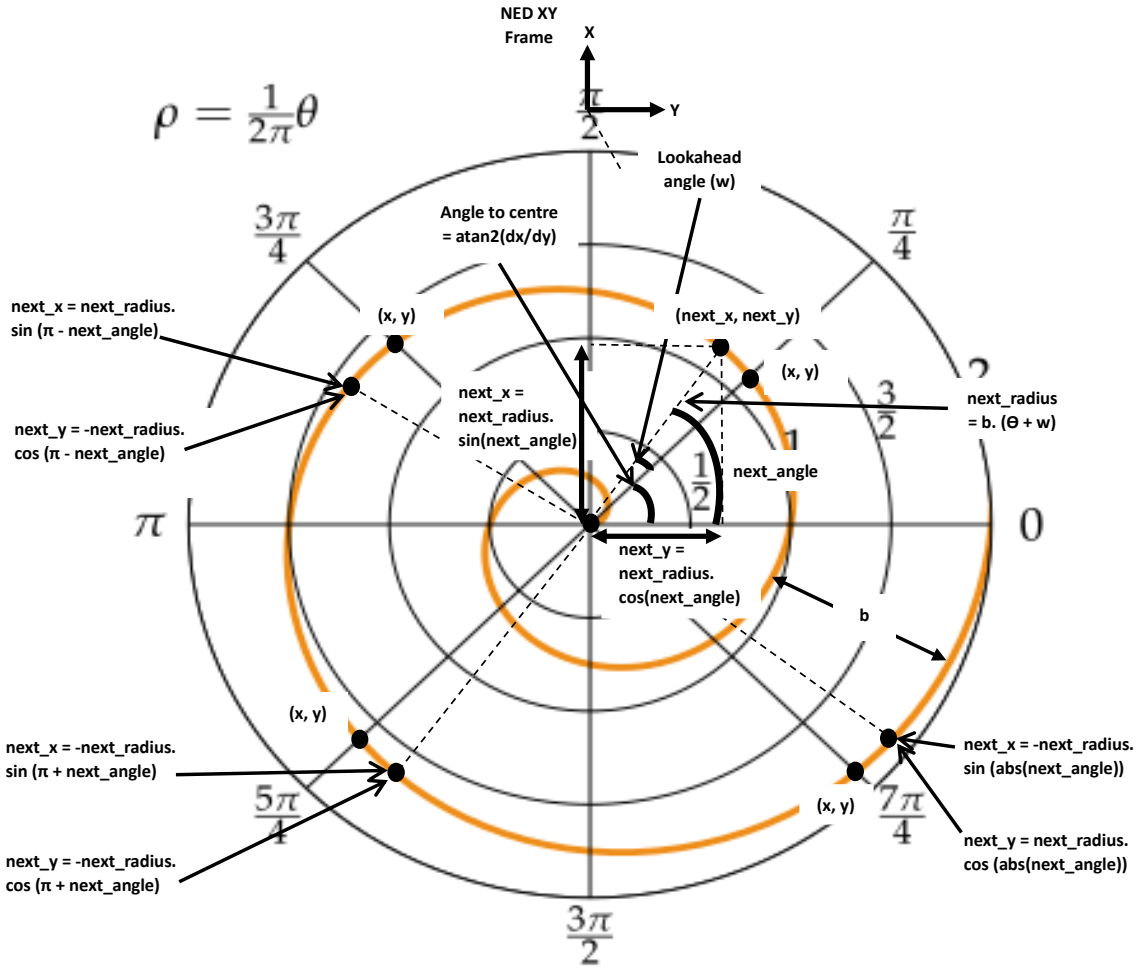

Figure 4.23. Visualization of the implementation of the Archimedean Spiral

For the easiness of understanding, the Table 4 describes the various parameters portrayed in the Figure 4.23. Visualization of the implementation of the Archimedean Spiral starting from the first quadrant.

Table 4. Explanation of the Archimedean Spiral implementation

Quadrant	Parameters	Description
First	NED XY Frame	The X and Y direction of the NED frame that the AirSim Drone works on

	(x, y)	The current (x, y) location of the drone in NED Frame
	Angle to centre = atan2(dx/dy)	The angle to the origin point from the current location of the drone where dx = x - 0 = x, dy = y - 0 = y, as the origin point in this study was taken as (0,0). The *atan2* is the arc or inverse tangent function of the *math* library in python.
	Lookahead angle (w)	The small lookahead angle in radians from the current angle to centre. For this work, the chosen value was 3° x π/180 radians.
	next_angle	Angle to centre + Lookahead angle.
	next_radius = b. (Θ + w)	The distance from the origin to the next point in the locus according to the {Equation}. Here, parameter *a* was ignored. (Θ + w) denotes the polar coordinate angle Θ for the next point, and it was implemented as the running sum of the difference between the present next_angle and the previous next_angle in each step.
	next_x = next_radius. sin(next_angle)	The x value of the next point in the locus after the conversion from polar coordinate. It is different from the normal cartesian coordinate because of the change of frame in NED. Here, "." represents the multiplication
	next_y = next_radius. cos(next_angle)	The y value of the next point in the locus after the conversion from polar coordinate.
Second	next_x = next_radius. sin (π - next_angle)	For mathematical simplicity and intuition, it was decided to keep the next_angle between 0 and π/2 radians to always calculate as a right-angled triangle. The *atan2* function yields values from 0 to π/2 in the first quadrant, and π/2 to π radians for the second quadrant in anticlockwise direction. Hence,

		(π –next_angle) gave values between 0 and π/2 in clockwise direction.
	next_y = -next_radius. cos (π - next_angle)	In the second quadrant, the value in vertical direction remains same in polarity but the value in horizontal direction has reverse polarity. Hence, the sign "-" for next_y.
Third	next_x = -next_radius. sin (π + next_angle)	The *atan2* function gave output from - π/2 to - π for the angles in the third quadrant in clockwise direction. Hence, addition of π gave positive values between 0 and π/2 in the anticlockwise direction.
	next_y = -next_radius. cos (π + next_angle)	The polarities are negative for both vertical and horizontal directions in third quadrant.
Fourth	next_x = -next_radius. sin (abs(next_angle))	The *atan2* function outputs values between 0 and - π/2 radians for the angles in the fourth quadrant in clockwise direction. Therefore, the absolute value of next_angle yielded positive values between 0 and π/2 radians in clockwise direction.
	next_y = next_radius. cos (abs(next_angle))	In the fourth quadrant, the values in the vertical direction are negative whereas the values in the horizontal direction are positive.

With the values of the next point in the path known autonomously, the next step was to send proper velocity commands to follow the spiral path by the drone in the form of Twist message that contained linear and angular velocity components in x, y, and z direction. The velocities in x and y directions were sent PD controlled values with error being the difference in next point (next_x, next_y) and current point (x, y) with controller parameters *Kp* and *Td* heuristically tuned to 1 and 0.5 respectively. Likewise, the altitude of the drone i.e., z value was also PD controlled to a desired altitude with controller parameters *Kp* and *Td* as 2 and 5 respectively. In addition, the roll and pitch of the drone were also tried to be controlled with PI controller having parameters *Kp* and *Ti* as 2 and 10 respectively after heuristic tuning as seen in the code in {Appendix}. Then, with a small value of parameter b of 1 as in Figure 4.23, the *Archimedean_Spiral_Path* node was tested for the result to give the output as shown in Figure 4.24.

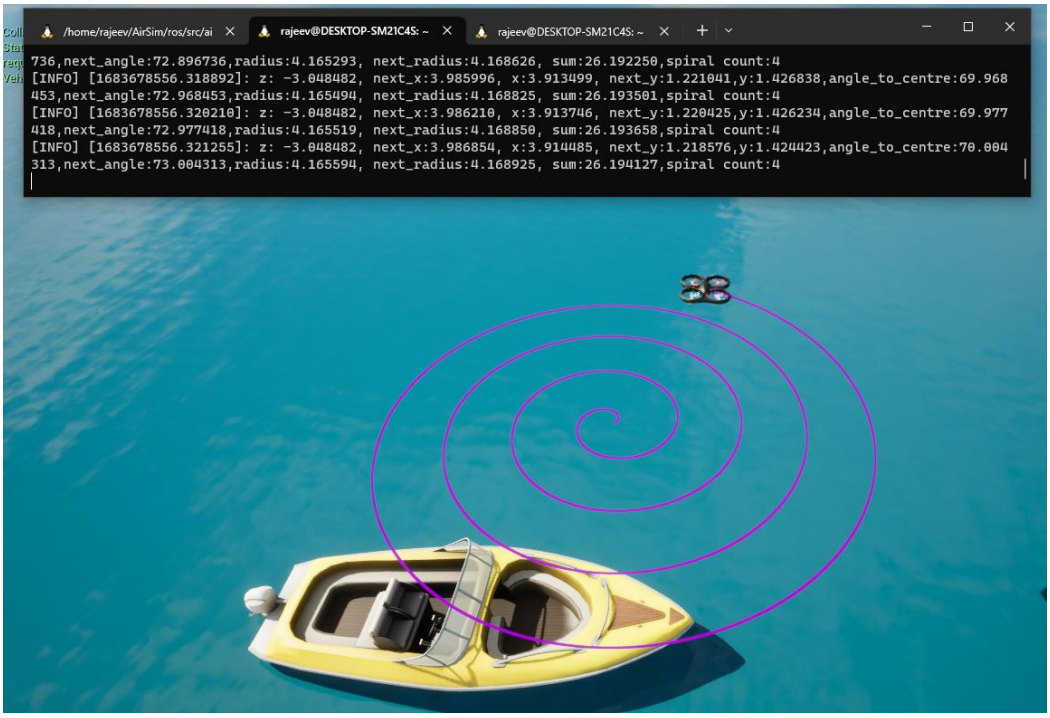

Figure 4.24. Testing of the path following by *Archimedean_Spiral_Path* node in the *Environment*

The pink traced path seen in the Figure 4.24 is the path moved by the drone, which is a default feature of AirSim, and is activated by pressing *T* on the keyboard. It can be seen from Figure 4.24 that the drone perfectly follows the Archimedean spiral as in Figure 4.23 with unit equal distance between the loops. Furthermore, due to the addition of extra control parameters the drone not only followed the spiral path, but it was also able to endure very high manual force trying to disrupt the path, making the path following more robust for search and rescue missions as shown in Figure 4.25.

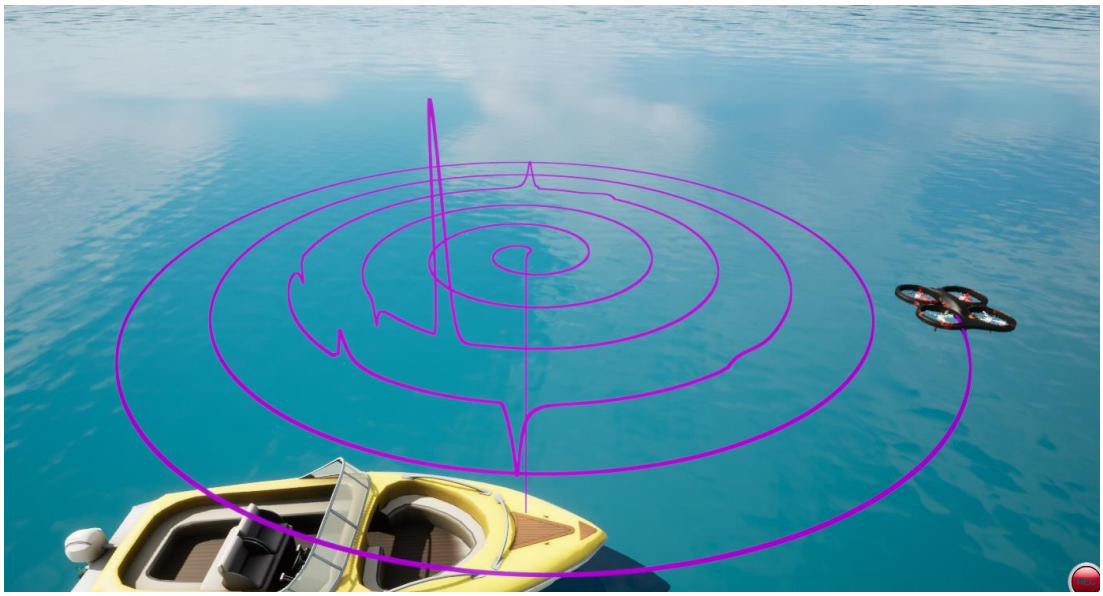

Figure 4.25. Testing for the robustness of the path following

4.5 Direct Georeferencing

With the conceptual framework as shown in Figure 4.26, the detection centre in pixels which was outputted from the YOLOv8 model in *yolov8_sea* node, was converted into NED coordinate position in the world, i.e., direct georeferencing, with the complete code implementation in ROS as a separate node named as *Georeferencing* is presented in Appendix F.

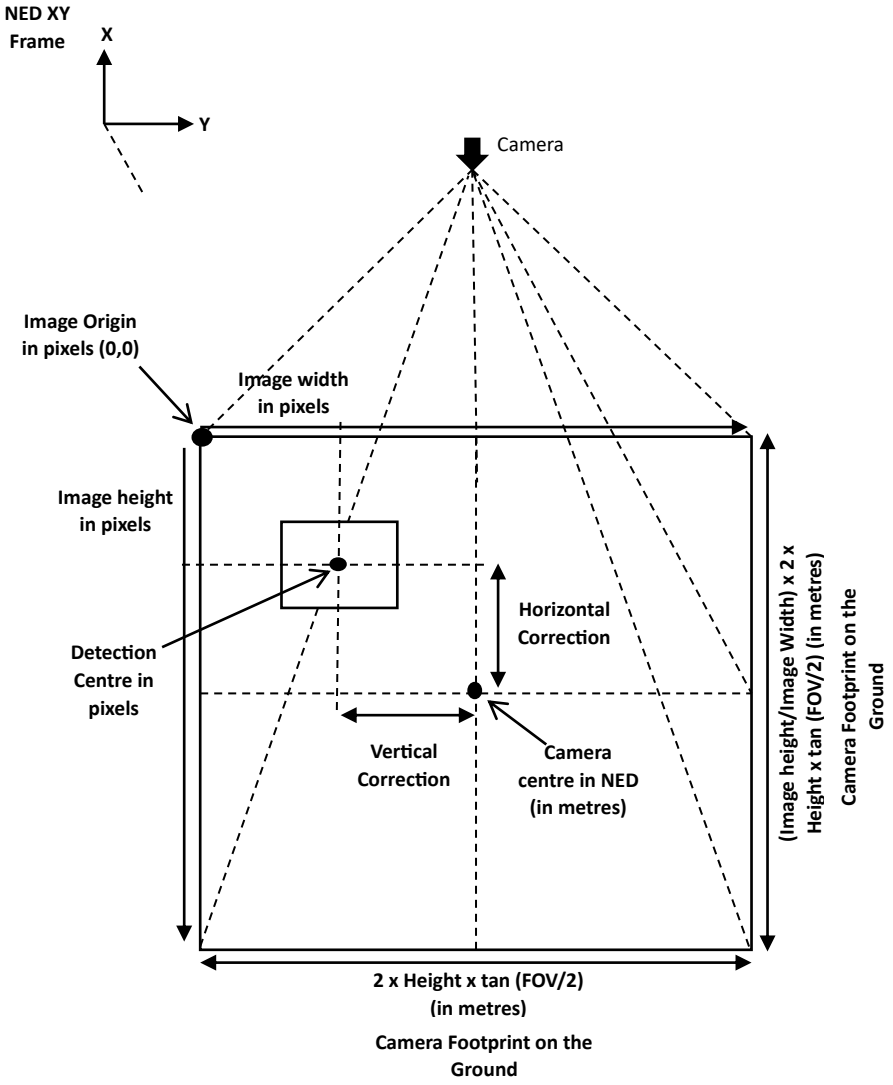

Figure 4.26. Conceptual diagram for Direct Georeferencing.

And the flowchart for the logical flow with pseudocode of the code implementation based on the Figure 4.26 is illustrated in Figure 4.27.

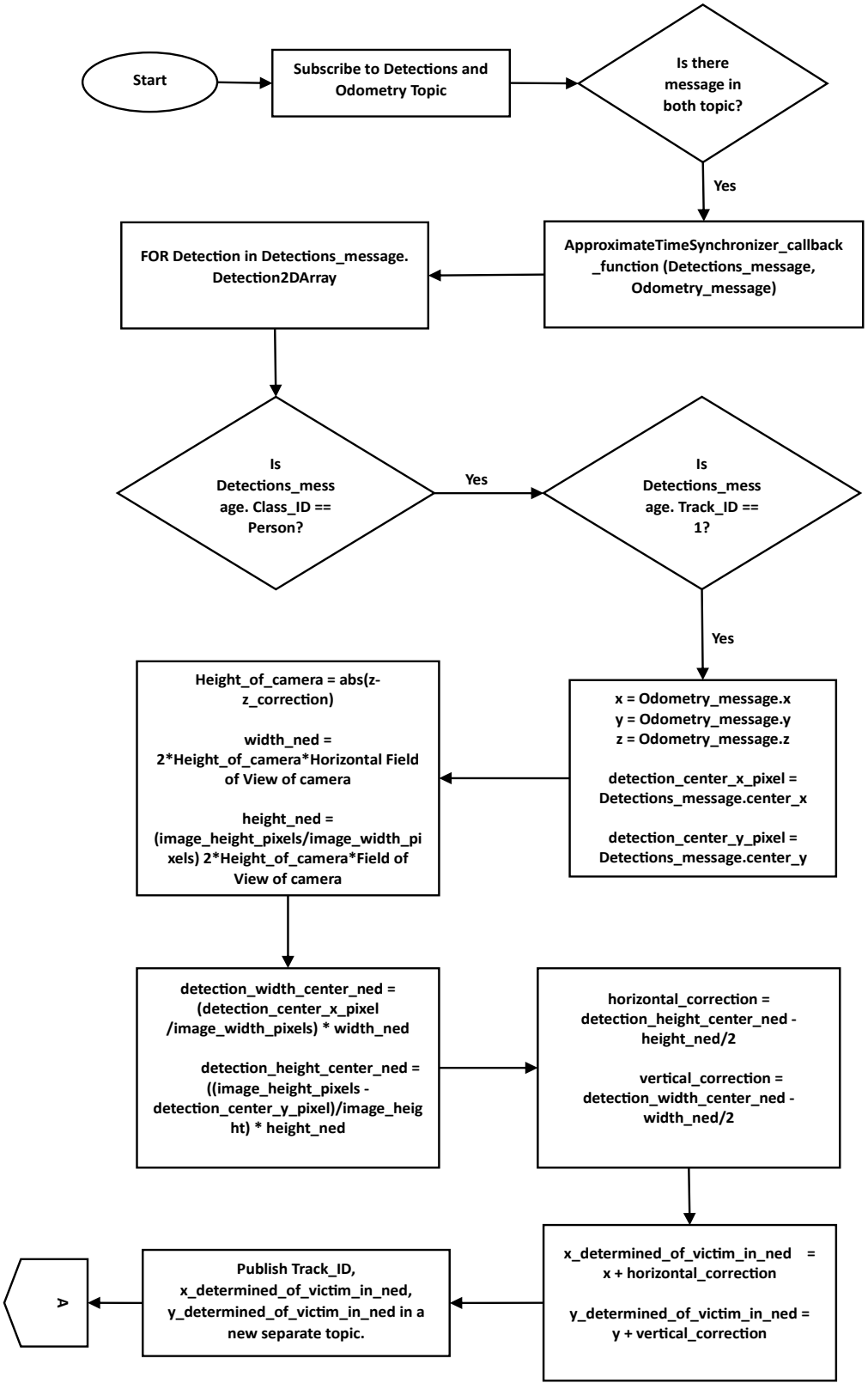

Figure 4.27. Flowchart for the code implementation of Direct Georeferencing

Here, as seen in Figure 4.27, basically, the *Georeferencing* node listens to two topics */detections* from *yolov8_sea* node and the *Odometry topic* from the AirSim using *Approximate Time Synchronizer* in ROS that synchronizes the messages in both the topics based on the maximum difference of time in the timestamps of both the messages and produces a single callback function. For this study, the maximum difference in time given by the *slop* parameter was set to a very low value of 0.1 seconds. Then, as */detections* topic contained *Detection2Darray* message which comprised of all the detections in the single frame of image, only the first person detected and tracked by the model was selected for the tracking by the drone in this study. All the steps used after that are self-explanatory based on Figure 4.26 and Figure 4.27.

4.6 PD Tracking by Drone

Furthermore, the continuous tracking by drone until the victim was rescued was also implemented inside the *Georeferencing* node as seen in Appendix F. The continuation of the flowchart shown in Figure with the addition of the logical flow for tracking is shown in Figure 4.28.

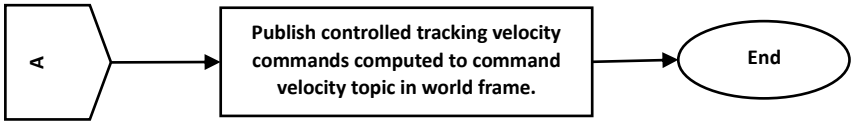

Figure 4.28. Flowchart for PD tracking of the victim by the drone

The position errors were calculated between the NED coordinates of the detected victim and the current NED position of the drone obtained from the Odometry topic. Similarly, the altitude of the drone was also controlled to a desired height specified by the developer. Both position and altitude of the drone were controlled using PD controller with proportional gain *Kp* set to 0.2 and the derivative gain *Kd* set to 2 after exhaustive heuristic tuning for best performance. After that, the computed controlled velocity values were published to the command velocity in world frame topic of the drone using *Twist* message as seen in the code implementation. Moreover, the testing procedure for the successful operation of the Georeferencing node with proper PD parameters was carried out with the continuously swimming forward *Pete* character described in Section 4.2.3 as shown in Figure 4.29. The drone was manually flown near the swimming person till the person was inside the frame of the camera, then the node would get activated itself and start tracking the person autonomously. The pink traced line shows the autonomous movement of the drone. The parameters were changed until stable and satisfactory tracking was obtained for the swimmer. In addition, manual disturbances were also added to the drone to change its path, orientation, and altitude while autonomous tracking, but it overcame all the disturbances also to continue tracking the person.

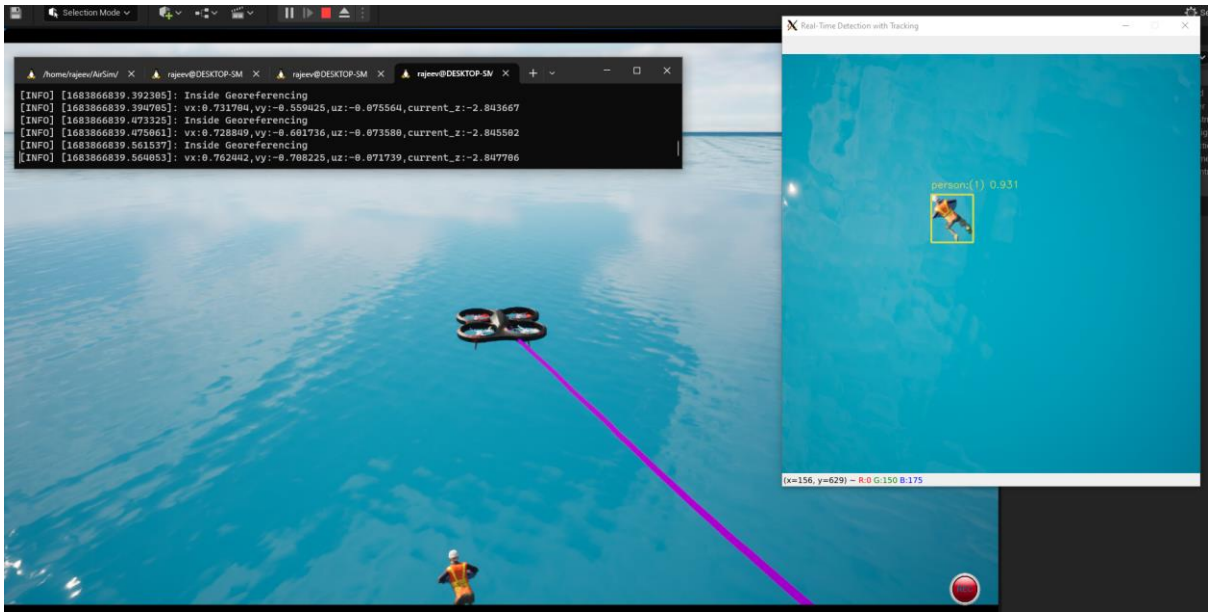

Figure 4.29. Testing of *Georeferencing* node with YOLOv8 pretrained detection model

The tracking action looked like Figure 4.30 from the side view.

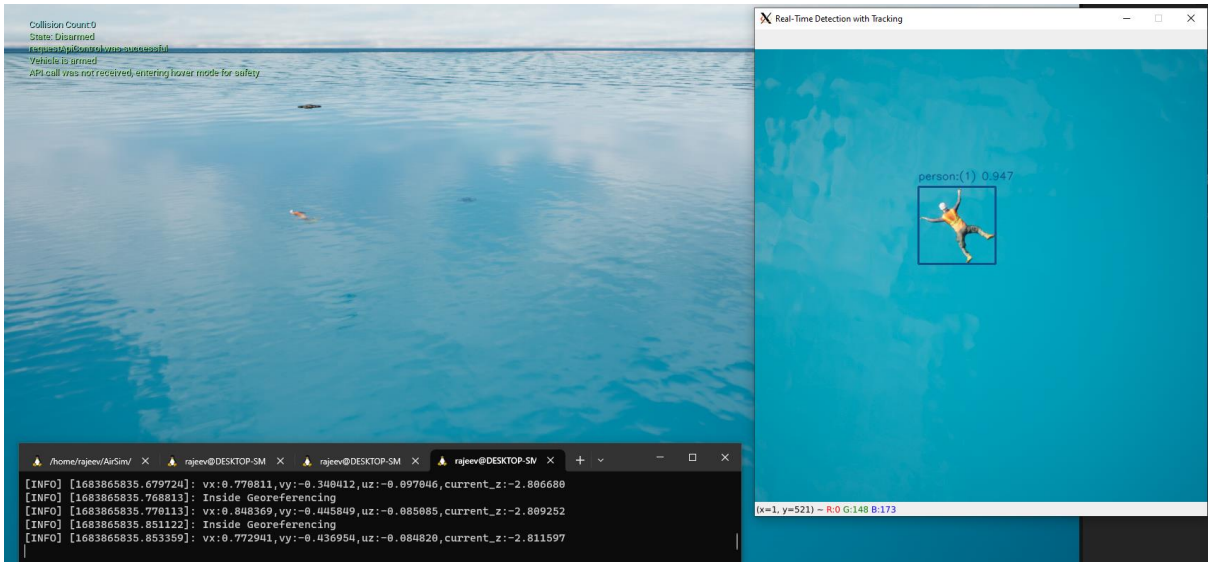

Figure 4.30. Testing of Georeferencing node from the side view

Similarly, the pretrained YOLOv8 segmentation model was also tested for tracking performance as shown in Figure 4.31, and it also gave satisfactory performance.

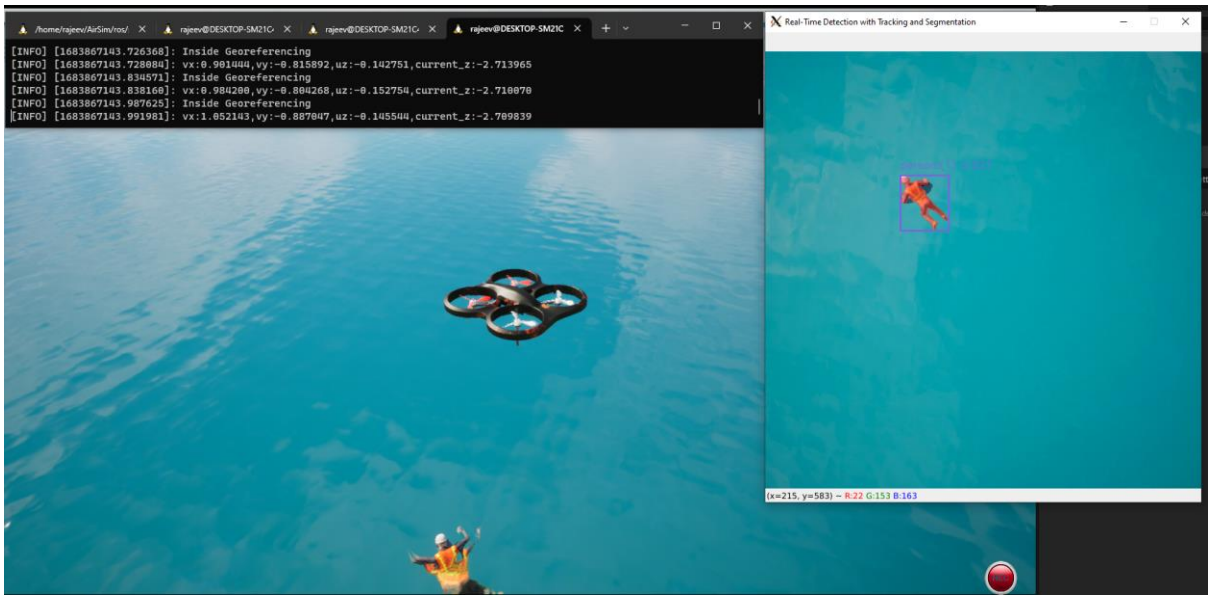

Figure 4.31. Testing of *Georeferencing* node for tracking with pretrained YOLOv8 segmentation model

4.7 Victim Geolocation

Even though the NED coordinates were useful for the local control of the drone, for external use and transmission, the local NED coordinate position of the detected victim were converted into GPS coordinates in latitude and longitude. For that, a separate node was created titled *Victim_Geolocation* whose complete code implementation is kept in Appendix G. It was decided to use a separate node from the *Georeferencing* node to separate the conversion and potential external transmission of the Victim_ID and GPS coordinates to various destinations which might have their own latency. The *Victim_Geolocation* node subscribed to the topic published by the *Georeferencing* node, which can be seen at the end of Figure. For the conversion of NED coordinate into GPS coordinate, a separate *ROS service* named *returngeolocationfromned* was created by modifying the *AirSim ROS wrapper* that used the default *nedToGeodetic* C++ function available inside *Earth Utils* module in AirSim C++ library. The *nedToGeodetic* function uses various independent parameters like the radius and curvature of the earth for conversion. The ROS service *returngeolocationfromned* took the NED coordinate point as input *Request* and returned output as *Response* the GPS coordinates. Also, the *Victim_Geolocation* node displayed the Person_ID, latitude, longitude, and altitude on the screen.

4.8 Autonomous Search and Rescue Mission

Finally, all the independent processes described in earlier sections and illustrated by the block diagram in Figure 4.1, were combined for working together to a single search and rescue autonomous mission with the collaboration between the different nodes as shown in Figure 4.32.

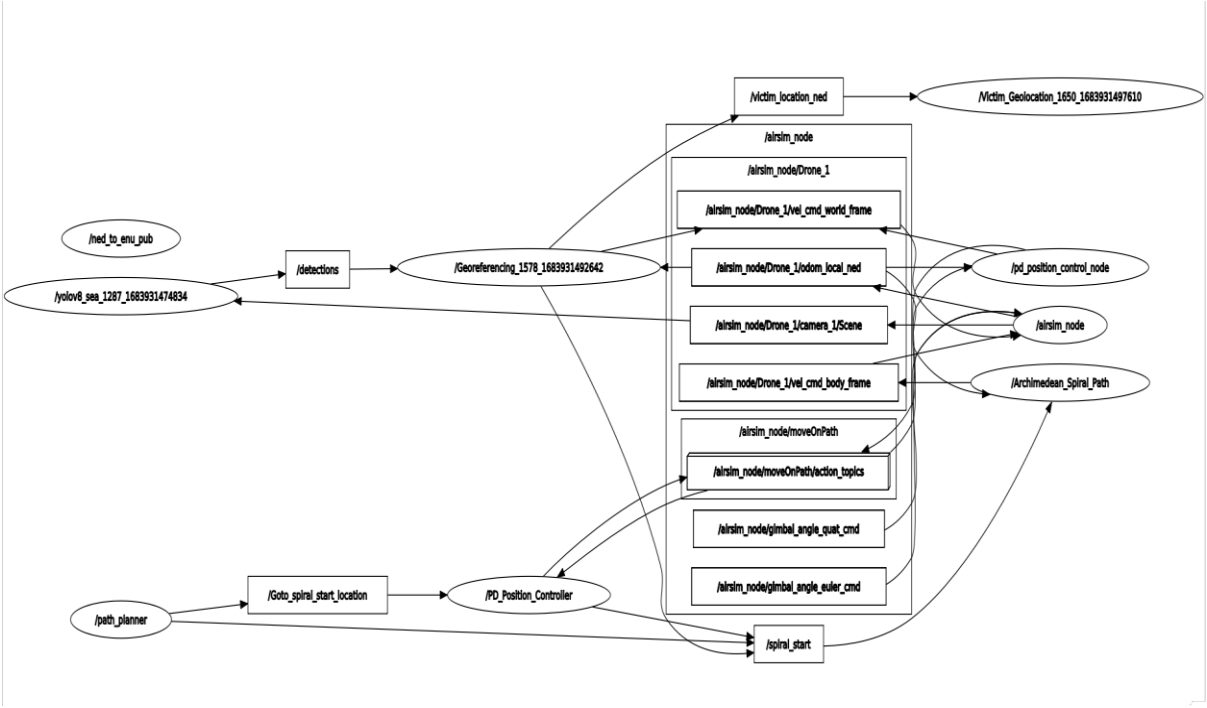

Figure 4.32. Collaboration between the nodes during the final autonomous mission obtained using *rqt_graph*.

In summary, an additional node named *path_planner* was developed, as seen in the lower left corner of Figure 4.32 with code implementation included in Appendix B, whose responsibility was to autonomously initiate the overall process by publishing a *message* with the start location including x, y and desired z values for the drone to start spiral search to a topic */Goto_spiral_start_location*. Also, another new *ROS Action Client* node named *PD_Position_Controller* was fabricated, with full code implementation kept in Appendix C, to listen to the message from the *path_planner* node and transfer the message to ROS PD position controller Action server node titled *pd_position_control_node* , same one discussed in Section 3.4.3 of the paper, for actually moving the drone to the spiral start location. When the drone reached the spiral search start location, the *Result* back from the action server became *True*, and then the *PD_Position_Controller* published a message in the topic */spiral_start* which was subscribed by the *Archimedean_Spiral_Path* node that started the spiral search process. The drone then started following the spiral search path until the *yolov8_sea* node detected the first person which in turn triggered the *Georeferencing* node. The *Georeferencing* node was modified to initially publish a message to the *Archimedean_Spiral_Path* node through */spiral_start* topic to stop the spiral path following of the drone. Then, the usual functioning of the *Georeferencing* node started along with the PD tracking of the victim. Finally, the mission ended with the *Victim_Geolocation* node displaying the ID, latitude, longitude, and altitude values of the victim on the screen until the boat autonomously came to the detected victim location and on boarded the person onto it after user specified *delay* as explained in detail in Section 4.2.5.

4.9 Results and Discussion

This section will present and discuss the results of the complete autonomous search and rescue mission described in the earlier section. The Figure 4.33 illustrates the autonomous launching of the drone by *path_planner* node and *PD_Position_Controller* node to the spiral start location, then moving in the spiral path with *Archimedean_Spiral_Path* node until the swimming victim detected, the detection by the *yolov8_sea* node and the tracking of the victim by the drone through *Georeferencing* node before the boat came to the rescue. Here, the pink traced line shows the path followed by the drone. The character *Leonard* that continuously swims in a square pattern was kept nearby the launching point of the drone from the boat for the easiness of visualization of the mission result in a single frame like in Figure 4.33.

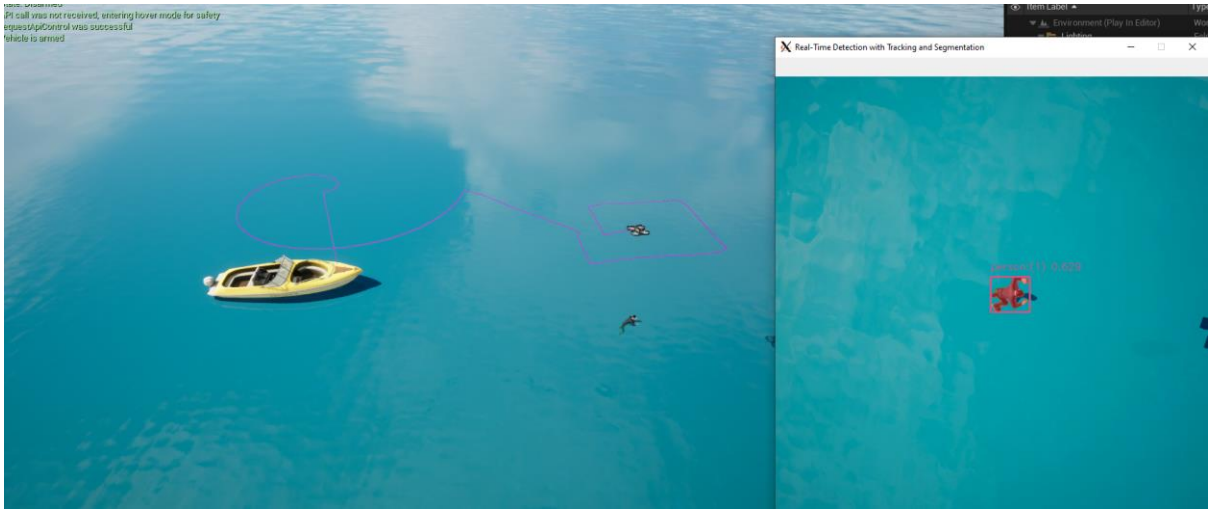

Figure 4.33. Autonomous search mission by the drone before the boat came to the rescue.

Then, the Figure 4.34 shows the boat coming to the rescue of the person, who is continuously detected and followed by the drone, and on boarding the victim on it after 120 seconds of delay specified by the author, starting from the point in time the drone first starts detecting the victim.

Figure 4.34. Simulation result of the rescue of the victim by the boat autonomously

The Figure 4.35 portrays the result in a different closer view of the external camera which makes the path followed by the drone in the autonomous search and rescue mission clearer.

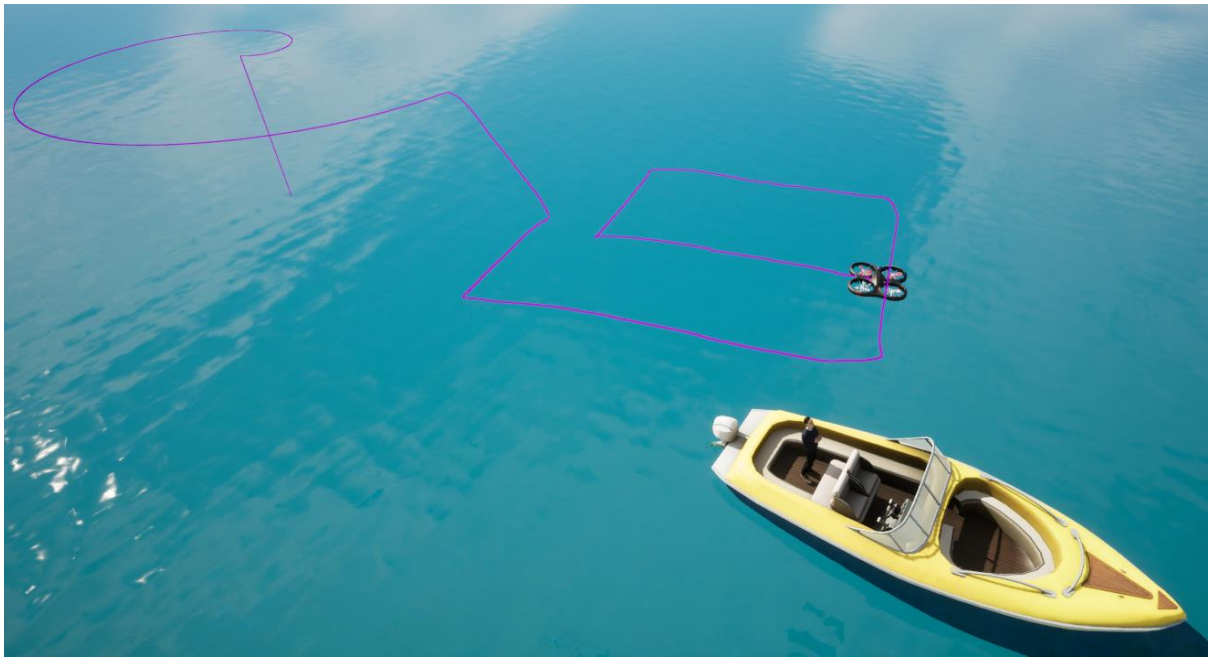

Figure 4.35. Closer view of the autonomous rescue of the victim by the speed boat with the help of the autonomous drone

Furthermore, the screenshot of the ID and geographical location of the detected victim calculated by the *Victim_Geolocation* node simultaneously is shown in Figure 4.36.

```
rajeev@DESKTOP-SM21C4S:~$ rosrun airsim_ros_pkgs VictimGeolocation.py
[INFO] [1683931607.882854]: Person_ID:1,Latitude:57.963547, Longitude:9.130315, Altitude:118.724541
[INFO] [1683931608.001397]: Person_ID:1,Latitude:57.963546, Longitude:9.130315, Altitude:118.724541
[INFO] [1683931608.128979]: Person_ID:1,Latitude:57.963543, Longitude:9.130315, Altitude:118.724541
[INFO] [1683931608.258590]: Person_ID:1,Latitude:57.963543, Longitude:9.130316, Altitude:118.724541
[INFO] [1683931608.395781]: Person_ID:1,Latitude:57.963544, Longitude:9.130316, Altitude:118.724541
[INFO] [1683931608.674785]: Person_ID:1,Latitude:57.963544, Longitude:9.130317, Altitude:118.724541
[INFO] [1683931608.814981]: Person_ID:1,Latitude:57.963544, Longitude:9.130318, Altitude:118.724541
[INFO] [1683931608.949293]: Person_ID:1,Latitude:57.963544, Longitude:9.130319, Altitude:118.724541
[INFO] [1683931609.075282]: Person_ID:1,Latitude:57.963543, Longitude:9.130320, Altitude:118.724541
[INFO] [1683931609.226897]: Person_ID:1,Latitude:57.963543, Longitude:9.130322, Altitude:118.724541
[INFO] [1683931609.356133]: Person_ID:1,Latitude:57.963543, Longitude:9.130322, Altitude:118.724541
[INFO] [1683931609.499037]: Person_ID:1,Latitude:57.963543, Longitude:9.130322, Altitude:118.724541
[INFO] [1683931609.635607]: Person_ID:1,Latitude:57.963543, Longitude:9.130323, Altitude:118.724541
[INFO] [1683931609.768541]: Person_ID:1,Latitude:57.963543, Longitude:9.130322, Altitude:118.724541
[INFO] [1683931609.910045]: Person_ID:1,Latitude:57.963542, Longitude:9.130324, Altitude:118.724541
[INFO] [1683931610.040223]: Person_ID:1,Latitude:57.963542, Longitude:9.130325, Altitude:118.724541
[INFO] [1683931610.175909]: Person_ID:1,Latitude:57.963541, Longitude:9.130325, Altitude:118.724541
[INFO] [1683931610.315414]: Person_ID:1,Latitude:57.963542, Longitude:9.130326, Altitude:118.724541
[INFO] [1683931610.447224]: Person_ID:1,Latitude:57.963542, Longitude:9.130326, Altitude:118.724541
[INFO] [1683931610.586629]: Person_ID:1,Latitude:57.963542, Longitude:9.130327, Altitude:118.724541
[INFO] [1683931610.716908]: Person_ID:1,Latitude:57.963541, Longitude:9.130329, Altitude:118.724541
[INFO] [1683931610.859586]: Person_ID:1,Latitude:57.963541, Longitude:9.130330, Altitude:118.724541
[INFO] [1683931610.999914]: Person_ID:1,Latitude:57.963541, Longitude:9.130330, Altitude:118.724541
[INFO] [1683931611.149074]: Person_ID:1,Latitude:57.963541, Longitude:9.130330, Altitude:118.724541
[INFO] [1683931611.282071]: Person_ID:1,Latitude:57.963541, Longitude:9.130328, Altitude:118.724541
[INFO] [1683931611.418061]: Person_ID:1,Latitude:57.963541, Longitude:9.130328, Altitude:118.724541
[INFO] [1683931611.538669]: Person_ID:1,Latitude:57.963540, Longitude:9.130329, Altitude:118.724541
[INFO] [1683931611.682546]: Person_ID:1,Latitude:57.963540, Longitude:9.130328, Altitude:118.724541
[INFO] [1683931611.829502]: Person_ID:1,Latitude:57.963540, Longitude:9.130328, Altitude:118.724541
[INFO] [1683931611.961178]: Person_ID:1,Latitude:57.963540, Longitude:9.130328, Altitude:118.724541
[INFO] [1683931612.088798]: Person_ID:1,Latitude:57.963540, Longitude:9.130328, Altitude:118.724541
[INFO] [1683931612.228923]: Person_ID:1,Latitude:57.963540, Longitude:9.130328, Altitude:118.724541
[INFO] [1683931612.361756]: Person_ID:1,Latitude:57.963539, Longitude:9.130329, Altitude:118.724541
[INFO] [1683931612.501503]: Person_ID:1,Latitude:57.963539, Longitude:9.130330, Altitude:118.724541
[INFO] [1683931612.635628]: Person_ID:1,Latitude:57.963538, Longitude:9.130331, Altitude:118.724541
[INFO] [1683931612.767926]: Person_ID:1,Latitude:57.963538, Longitude:9.130331, Altitude:118.724541
[INFO] [1683931612.912541]: Person_ID:1,Latitude:57.963538, Longitude:9.130331, Altitude:118.724541
[INFO] [1683931613.050725]: Person_ID:1,Latitude:57.963538, Longitude:9.130331, Altitude:118.724541
[INFO] [1683931613.192275]: Person_ID:1,Latitude:57.963539, Longitude:9.130330, Altitude:118.724541
[INFO] [1683931613.332014]: Person_ID:1,Latitude:57.963538, Longitude:9.130330, Altitude:118.724541
[INFO] [1683931613.466835]: Person_ID:1,Latitude:57.963538, Longitude:9.130330, Altitude:118.724541
[INFO] [1683931613.616292]: Person_ID:1,Latitude:57.963538, Longitude:9.130330, Altitude:118.724541
[INFO] [1683931613.750426]: Person_ID:1,Latitude:57.963537, Longitude:9.130329, Altitude:118.724541
[INFO] [1683931613.875559]: Person_ID:1,Latitude:57.963537, Longitude:9.130329, Altitude:118.724541
[INFO] [1683931614.012450]: Person_ID:1,Latitude:57.963537, Longitude:9.130329, Altitude:118.724541
[INFO] [1683931614.146433]: Person_ID:1,Latitude:57.963537, Longitude:9.130329, Altitude:118.724541
[INFO] [1683931614.291022]: Person_ID:1,Latitude:57.963536, Longitude:9.130331, Altitude:118.724541
[INFO] [1683931614.425584]: Person_ID:1,Latitude:57.963536, Longitude:9.130330, Altitude:118.724541
[INFO] [1683931614.546081]: Person_ID:1,Latitude:57.963535, Longitude:9.130330, Altitude:118.724541
[INFO] [1683931614.683251]: Person_ID:1,Latitude:57.963535, Longitude:9.130331, Altitude:118.724541
```

Figure 4.36. Victim ID, Latitude, Longitude, and Altitude calculated by the *Victim_Geolocation* node.

As the drone was continuously detecting the victim along with the calculation of the location, only the few of the beginning detection results are shown in Figure 4.36. Furthermore, the

geographical location in latitude and latitude being detected by the drone in Google maps [46] is illustrated in Figure 4.37.

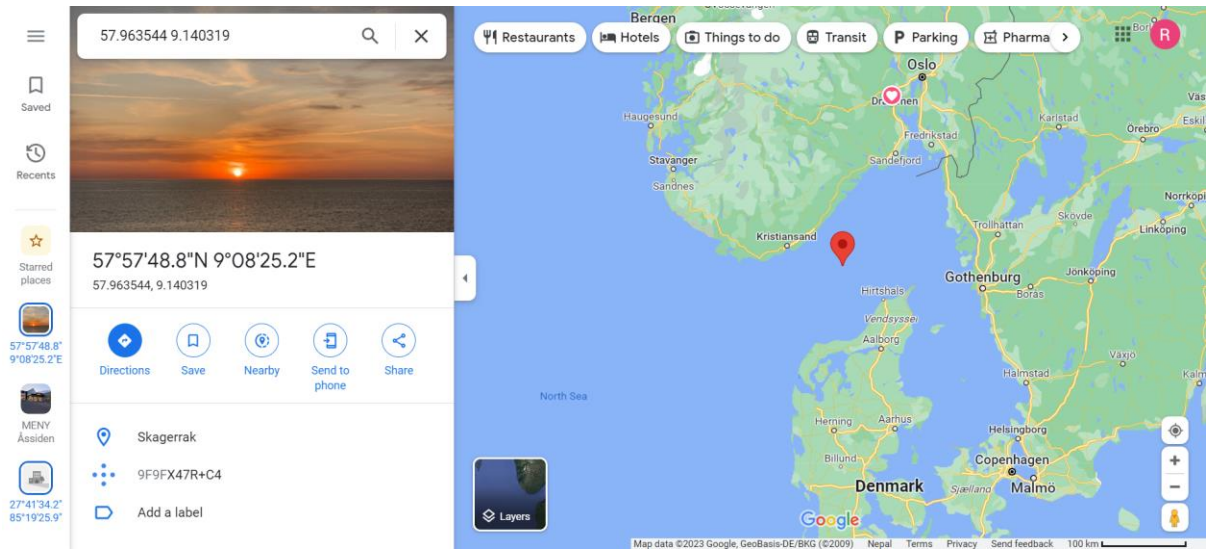

Figure 4.37. Geographical location of the detection in Google maps [46]

The area between Norway, Denmark, and the Netherlands shown in Figure is the area of interest for USN in Project Valkyries which was achieved by previously setting *the origin geopoint* in AirSim settings as explained in {section}. Hence, the Figure and Figure indicates that the conversion between the detected and georeferenced NED coordinates in meters and the GPS coordinates in latitude and longitude of the position of the victim in the simulated *Environment* is working correctly. Therefore, with this study, the simulated information of the victim like ID, latitude, longitude, and altitude detected by the autonomous drone during the simulation of the oil tanker accident are available in a separate node implemented with Python in ROS which can be easily used or transmitted to remote cloud-based system like Valkyries Dashboard for further testing.

Furthermore, with the drone continuously tracking the first person it detects until the rescue teams reach the victim for rescue can try to solve the issue of dynamic marine environment.

5 Discussion of Collaboration between Multiple Drones

This chapter will discuss the possibility of cooperation between multiple drones in the same framework. It was first important to develop and test the different techniques for controlling a single drone autonomously for various use cases in search and rescue missions which was carried out in previous chapters. Then, multiple drones could easily be spawned and controlled in the simulation environment as shown in Figure 5.1 with simple changes in AirSim settings based on AirSim Documentation [38]. In addition to cameras, other sensors like LIDAR, Distance Sensor can also be activated.

Figure 5.1. Multiple drones spawned in the simulation environment.

The Figure 5.1 is analogous to the Figure 3.8 in Section 3.3.2 of the paper but with three drones spawned. Also, the environment seen in Figure 5.1 is the recreated environment in a same way like in the paper described in Chapter 3, but in Unreal Engine version 5.1 by downloading and updating the version that worked in Unreal Engine 5.0 from Environment Project [36]. Even though this updated environment was available, the custom environment was used in the third chapter to illustrate the flexibility of the overall framework used in this research thesis. The same ROS nodes work with both the marine environments, or any other custom environment of different geography developed in Unreal Engine, and the assets and blueprints developed can also be used interchangeably among the environments.

Now, again coming back to the multiple drones spawned in the environment, the AirSim ROS wrapper automatically duplicates all the topics available for each drone to multiple drones distinctly separated by their name in namespace as shown in Figure 5.2.

```
rajeev@DESKTOP-SM21C4S:~$ rostopic list
/airsim_node/Drone_1/altimeter/barometer
/airsim_node/Drone_1/camera_1/DepthPlanar
/airsim_node/Drone_1/camera_1/DepthPlanar/camera_info
/airsim_node/Drone_1/camera_1/Scene
/airsim_node/Drone_1/camera_1/Scene/camera_info
/airsim_node/Drone_1/distance/Distance_1
/airsim_node/Drone_1/distance/Distance_2
/airsim_node/Drone_1/environment
/airsim_node/Drone_1/global_gps
/airsim_node/Drone_1/gps/gps
/airsim_node/Drone_1/imu/imu_1
/airsim_node/Drone_1/magnetometer/magnetometer
/airsim_node/Drone_1/odom_local_ned
/airsim_node/Drone_1/vel_cmd_body_frame
/airsim_node/Drone_1/vel_cmd_world_frame
/airsim_node/Drone_2/altimeter/barometer
/airsim_node/Drone_2/camera_2/DepthPlanar
/airsim_node/Drone_2/camera_2/DepthPlanar/camera_info
/airsim_node/Drone_2/camera_2/Scene
/airsim_node/Drone_2/camera_2/Scene/camera_info
/airsim_node/Drone_2/distance/Distance_1
/airsim_node/Drone_2/distance/Distance_2
/airsim_node/Drone_2/environment
/airsim_node/Drone_2/global_gps
/airsim_node/Drone_2/gps/gps
/airsim_node/Drone_2/imu/imu_2
/airsim_node/Drone_2/magnetometer/magnetometer
/airsim_node/Drone_2/odom_local_ned
/airsim_node/Drone_2/vel_cmd_body_frame
/airsim_node/Drone_2/vel_cmd_world_frame
/airsim_node/Drone_3/altimeter/barometer
/airsim_node/Drone_3/camera_3/DepthPlanar
/airsim_node/Drone_3/camera_3/DepthPlanar/camera_info
/airsim_node/Drone_3/camera_3/Scene
/airsim_node/Drone_3/camera_3/Scene/camera_info
/airsim_node/Drone_3/environment
/airsim_node/Drone_3/global_gps
/airsim_node/Drone_3/gps/gps
/airsim_node/Drone_3/imu/imu_3
/airsim_node/Drone_3/lidar/lidar_3
/airsim_node/Drone_3/magnetometer/magnetometer
/airsim_node/Drone_3/odom_local_ned
/airsim_node/Drone_3/vel_cmd_body_frame
/airsim_node/Drone_3/vel_cmd_world_frame
/airsim_node/all_robots/vel_cmd_body_frame
/airsim_node/all_robots/vel_cmd_world_frame
/airsim_node/gimbal_angle_euler_cmd
/airsim_node/gimbal_angle_quat_cmd
/airsim_node/group_of_robots/vel_cmd_body_frame
/airsim_node/group_of_robots/vel_cmd_world_frame
/airsim_node/origin_geo_point
```

Figure 5.2. ROS Topics available with 3 drones spawned in the simulation environment.

Similarly, the Figure 5.3 illustrates the ROS services available with three drones activated.

```
rajeev@DESKTOP-SM21C4S:~$ rosservice list
/airsim_node/Drone_1/land
/airsim_node/Drone_1/returngeolocationfromned
/airsim_node/Drone_1/takeoff
/airsim_node/Drone_2/land
/airsim_node/Drone_2/returngeolocationfromned
/airsim_node/Drone_2/takeoff
/airsim_node/Drone_3/land
/airsim_node/Drone_3/returngeolocationfromned
/airsim_node/Drone_3/takeoff
/airsim_node/all_robots/land
/airsim_node/all_robots/takeoff
/airsim_node/get_loggers
/airsim_node/group_of_robots/land
/airsim_node/group_of_robots/takeoff
/airsim_node/reset
/airsim_node/set_logger_level
/ned_to_enu_pub/get_loggers
/ned_to_enu_pub/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
```

Figure 5.3. ROS Services available with 3 drones spawned in the simulation environment.

Hence, with distinct topics and services for each drone available as shown in Figure 5.2 and Figure 5.3, multiple drones can also be simulated for autonomous missions with the same code implementation as in the Section 3.4.4 and Section 4.8 with the addition of higher layers of parent path_planner nodes for coordination as shown in Figure 5.4.

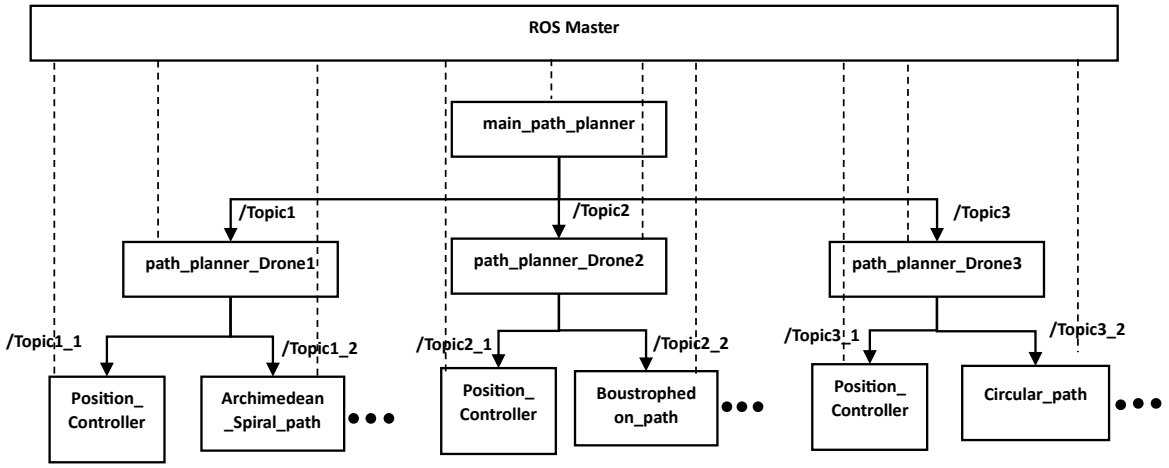

Figure 5.4. Example Hierarchical level for control and cooperation between multiple drones.

Due to the distributed parallel node processing capability of ROS, multiple nodes can run simultaneously, and each node can be given a different responsibility for the control of multiple drones. Another benefit of using ROS in controlling multiple drones as highlighted in Figure is all communication in a single ROS framework is handled by a single ROS master where the nodes register themselves in the beginning. Therefore, all the nodes in any level in the framework can communicate with each other when necessary. Some examples for this can be seen in {section}, when the *Georeferencing* node stopped the *Archimedean_Spiral_Path* node to change from search mode to tracking mode or in {section} where the *Boustrophedon_path* node was stopped by the main *path_planner* node after mission time was over and the drone was brought back to the home location. Also, the ROS Master can be over a network in the cloud in different machine, controlling nodes connected to it parallelly in distributed systems. Furthermore, the different drones can be sent to different starting locations following different path planning to search or track the victims, detect the disaster ship location or other concerned objects, and communicate with each other as well as external concerned parties through different topics. To exemplify this, the first drone can initially search for the location of the post disaster ship with spiral path in local NED position autonomously, then send the location message to other drones to extensively search around the detected disaster location.

Therefore, the combination of ROS, Microsoft AirSim and Unreal Engine is very important for the simulation of multi-agent drones. And this thesis primarily shows how they can be controlled with different responsibilities assigned for different nodes.

6 Conclusion

In conclusion, high fidelity marine simulation environments containing the post disaster ship, other debris such as containers, oil barrels, buoys, and most importantly people having different actions such as treading and swimming with natural buoyancy were developed in both Unreal Engine version 4.27 and 5.1 with the integration of Microsoft AirSim and ROS. Then, various strategies for controlling the spawned single drone in the simulated environments were developed and tested with the primary focus in the search and rescue of the victims followed by the discussion to scale the same solutions with multiple drones.

Furthermore, the work presented in this thesis has tremendous potential for further work. The immediate next work will be to test the various algorithms and models used in this study with real physical drones. In addition, collision avoidance strategies can be added on the drones to make them more robust as discussed by the authors in [13],[47], [48], and [49]. Additionally, various new Vision transformers [50] based object detection, and segmentation models such as Grounding DINO [51], Segment Anything [52], and a recently released transformer model faster than all YOLOs called as Real Time Detection Transformer (RT-DETR) [53] , and in CNN category newer models like YOLO-NAS [54] could be applied in the detection and path following solutions like proposed in [55].

Moreover, the main purpose of showing the boat autonomously coming to the rescue of the victim detected in Chapter 4 was to discuss the possible future work for the pair of aerial drones and the surface vehicles working together for the complete autonomous search and rescue. Various interesting vision based deep reinforcement learning methods proposed for surface systems as FastRLAP: A System for Learning High-Speed Driving via Deep RL and Autonomous Practicing [56], Legged Locomotion in Challenging Terrains using Egocentric Vision [57], Deep Whole-Body Control: Learning a Unified Policy for Manipulation and Locomotion [58], and UAV/USV Cooperative Trajectory Optimization Based on Reinforcement Learning [59] could be tried to be applied on the boat for autonomous collision avoiding movement for rescue. Similarly, some of the MPC based methods as discussed by the previous USN Student Syed Sami in his Master thesis [60], UAV-USV cooperative tracking based on MPC [61], Cooperative Path Planning for UAV, USV, and AUV together for search and rescue missions [62] and Autonomous Collision Avoidance MPC for USV [63]. Finally, another relevant interesting vision based collision avoidance work ongoing in USN that could be applied on the autonomous rescue boat is [64].

References

[1] ‘Harmonization and Pre-Standardization of Equipment, Training and Tactical Coordinated procedures for First Aid Vehicles deployment on European multi-victim Disasters | VALKYRIES Project | Fact Sheet | H2020 | CORDIS | European Commission’. https://cordis.europa.eu/project/id/101020676 (accessed Apr. 24, 2022).

[2] ‘Unreal Engine | The most powerful real-time 3D creation tool’, *Unreal Engine*. https://www.unrealengine.com/en-US (accessed May 14, 2023).

[3] ‘Understanding the Basics’. https://docs.unrealengine.com/5.0/en-US/understanding-the-basics-of-unreal-engine/ (accessed May 14, 2023).

[4] S. Shah, D. Dey, C. Lovett, and A. Kapoor, ‘AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles’. arXiv, Jul. 18, 2017. doi: 10.48550/arXiv.1705.05065.

[5] ‘Introduction · MAVLink Developer Guide’. https://mavlink.io/en/ (accessed May 14, 2023).

[6] ‘Flight Controller - AirSim’. https://microsoft.github.io/AirSim/flight_controller/ (accessed May 14, 2023).

[7] ‘Home - AirSim’. https://microsoft.github.io/AirSim/ (accessed May 14, 2023).

[8] ‘Aerial Autonomy: Project AirSim’, *Microsoft AI*. https://www.microsoft.com/en-us/AI/autonomous-systems-project-airsim (accessed May 14, 2023).

[9] ‘Welcome to Colosseum, a successor of AirSim’. Codex Laboratories LLC, May 08, 2023. Accessed: May 14, 2023. [Online]. Available: https://github.com/CodexLabsLLC/Colosseum

[10] ‘Documentation - ROS Wiki’. http://wiki.ros.org/Documentation (accessed May 14, 2023).

[11] ‘ROS: AirSim ROS Wrapper - AirSim’. https://microsoft.github.io/AirSim/airsim_ros_pkgs/ (accessed May 14, 2023).

[12] ‘Archimedean spiral’, *Wikipedia*. Apr. 01, 2023. Accessed: May 09, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Archimedean_spiral&oldid=1147744547

[13] F. A. de Alcantara Andrade *et al.*, ‘Autonomous Unmanned Aerial Vehicles in Search and Rescue Missions Using Real-Time Cooperative Model Predictive Control’, *Sensors*, vol. 19, no. 19, Art. no. 19, Jan. 2019, doi: 10.3390/s19194067.

[14] F. Balampanis, I. Maza, and A. Ollero, ‘Spiral-like coverage path planning for multiple heterogeneous UAS operating in coastal regions’, in *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, Jun. 2017, pp. 617–624. doi: 10.1109/ICUAS.2017.7991461.

[15] G. Xu, X. Chen, B. Wang, K. Li, J. Wang, and X. Wei, ‘A search strategy of UAV’s automatic landing on ship in all weathe’, in *2011 International Conference on Electrical*

and Control Engineering, Sep. 2011, pp. 2857–2860. doi: 10.1109/ICECENG.2011.6057120.

[16] R. Poudel, L. Lima, and F. Andrade, 'A Novel Framework to Evaluate and Train Object Detection Models for Real-Time Victims Search and Rescue at Sea with Autonomous Unmanned Aerial Systems Using High-Fidelity Dynamic Marine Simulation Environment', in *2023 IEEE/CVF Winter Conference on Applications of Computer Vision Workshops (WACVW)*, Waikoloa, HI, USA: IEEE, Jan. 2023, pp. 239–247. doi: 10.1109/WACVW58289.2023.00030.

[17] A. G. C. & Specialty, 'Safety and Shipping Review 2022'. [Online]. Available: https://www.agcs.allianz.com/news-and-insights/reports/shipping-safety.html

[18] Regjeringen, 'The Norwegian Search and Rescue Service'. [Online]. Available: https://www.regjeringen.no/globalassets/upload/kilde/jd/bro/2003/0005/ddd/pdfv/183865-infohefte_engelsk.pdfl

[19] Y.-Y. Choong, G. Salvendy, and others, 'Voices of First Responders–Applying Human Factors and Ergonomics Knowledge to Improve the Usability of Public Safety Communications Technology'. NISTIR, 2021.

[20] S. Wang, Y. Han, J. Chen, Z. Zhang, G. Wang, and N. Du, 'A deep-learning-based sea search and rescue algorithm by UAV remote sensing', in *2018 IEEE CSAA Guidance, Navigation and Control Conference (CGNCC)*, IEEE, 2018, pp. 1–5.

[21] R. Zheng, R. Yang, K. Lu, and S. Zhang, 'A search and rescue system for maritime personnel in disaster carried on unmanned aerial vehicle', in *2019 18th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, IEEE, 2019, pp. 43–47.

[22] S. Sambolek and M. Ivasic-Kos, 'Automatic person detection in search and rescue operations using deep CNN detectors', *IEEE Access*, vol. 9, pp. 37905–37922, 2021.

[23] B. Mishra, D. Garg, P. Narang, and V. Mishra, 'Drone-surveillance for search and rescue in natural disaster', *Computer Communications*, vol. 156, pp. 1–10, 2020.

[24] J. Lorincz, A. Tahirović, and B. R. Stojkoska, 'A Novel Real-Time Unmanned Aerial Vehicles-based Disaster Management Framework', in *2021 29th Telecommunications Forum (TELFOR)*, IEEE, 2021, pp. 1–4.

[25] C. D. Rodin, L. N. de Lima, F. A. de Alcantara Andrade, D. B. Haddad, T. A. Johansen, and R. Storvold, 'Object classification in thermal images using convolutional neural networks for search and rescue missions with unmanned aerial systems', in *2018 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2018, pp. 1–8.

[26] S. S. A. Zaidi, M. S. Ansari, A. Aslam, N. Kanwal, M. Asghar, and B. Lee, 'A survey of modern deep learning based object detection models', *Digital Signal Processing*, p. 103514, 2022.

[27] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, 'YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors'. arXiv, 2022. doi: 10.48550/ARXIV.2207.02696.

[28] E. Games, 'Unreal Engine'. [Online]. Available: https://www.unrealengine.com

[29] S. Shah, D. Dey, C. Lovett, and A. Kapoor, ‘AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles’, in *Field and Service Robotics*, 2017. [Online]. Available: https://arxiv.org/abs/1705.05065

[30] Stanford Artificial Intelligence Laboratory et al., ‘Robot Operating System’. [Online]. Available: https://www.ros.org/

[31] T. Do Trong, Q. T. Hai, N. T. Duc, and H. T. Thanh, ‘A Novelty Approach to Emulate Field Data Captured by Unmanned Aerial Vehicles for Training Deep Learning Algorithms Used for Search-and-Rescue Activities at Sea’, in *2020 IEEE Eighth International Conference on Communications and Electronics (ICCE)*, IEEE, 2021, pp. 288–293.

[32] F. A. D. A. Andrade *et al.*, ‘Virtual Reality Simulation of Autonomous Solar Plants Inspections with Unmanned Aerial Systems’, in *2021 Aerial Robotic Systems Physically Interacting with the Environment (AIRPHARO)*, IEEE, 2021, pp. 1–8.

[33] A. Redulla and S. P. Singh, ‘Simulating differential games with improved fidelity to better inform cooperative & adversarial two vehicle UAV flight’, in *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, IEEE, 2018, pp. 130–136.

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

[35] S. Wang, J. Chen, Z. Zhang, G. Wang, Y. Tan, and Y. Zheng, ‘Construction of a virtual reality platform for UAV deep learning’, in *2017 Chinese Automation Congress (CAC)*, IEEE, 2017, pp. 3912–3916.

[36] DotCam, TK-Master, Zoc, and S. Elble, ‘EnvironmentProject’, *GitHub repository*. GitHub, 2022. [Online]. Available: https://github.com/UE4-OceanProject/Environment-Project

[37] Adobe, ‘Mixamo by Adobe’. [Online]. Available: http://www.mixamo.com

[38] Microsoft, ‘AirSim’. [Online]. Available: https://microsoft.github.io/AirSim/

[39] T.-Y. Lin *et al.*, ‘Microsoft COCO: Common Objects in Context’. arXiv, 2014. doi: 10.48550/ARXIV.1405.0312.

[40] J. Higgins, *Canonical views of objects and scenes*. University of Illinois at Urbana-Champaign, 2011.

[41] L. A. Varga, B. Kiefer, M. Messmer, and A. Zell, ‘SeaDronesSee: A Maritime Benchmark for Detecting Humans in Open Water’. arXiv, 2021. doi: 10.48550/ARXIV.2105.01922.

[42] B. Kiefer, ‘SeaDronesSee’, *GitHub repository*. GitHub, 2022. [Online]. Available: https://github.com/Ben93kie/SeaDronesSee

[43] H. Choset and P. Pignon, ‘Coverage path planning: The boustrophedon cellular decomposition’, in *Field and service robotics*, Springer, 1998, pp. 203–209.

[44] G. Jocher, A. Chaurasia, and J. Qiu, ‘YOLO by Ultralytics’. Jan. 2023. Accessed: May 14, 2023. [Online]. Available: https://github.com/ultralytics/ultralytics

[45] M. Á. G. Santamarta, ‘yolov8_ros’. May 15, 2023. Accessed: May 15, 2023. [Online]. Available: https://github.com/mgonzs13/yolov8_ros

[46] ‘Google Maps’, *Google Maps*. https://www.google.com/maps/place/57%C2%B057'48.8%22N+9%C2%B008'25.2%22E/@57.8163417,4.339613,6.21z/data=!4m4!3m3!8m2!3d57.963544!4d9.140319 (accessed May 13, 2023).

[47] H. M. Jayaweera and S. Hanoun, ‘Real-time Obstacle Avoidance for Unmanned Aerial Vehicles (UAVs)’, in *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct. 2021, pp. 2622–2627. doi: 10.1109/SMC52423.2021.9659197.

[48] S. Song, Y. Zhang, X. Qin, K. Saunders, and J. Liu, ‘Vision-guided Collision Avoidance Through Deep Reinforcement Learning’, in *NAECON 2021 - IEEE National Aerospace and Electronics Conference*, Aug. 2021, pp. 191–194. doi: 10.1109/NAECON49338.2021.9696380.

[49] A. Lombard, L. Durand, and S. Galland, ‘Velocity Obstacle Based Strategy for Multi-agent Collision Avoidance of Unmanned Aerial Vehicles’, in *2020 IEEE International Conference on Sensing, Communication and Networking (SECON Workshops)*, Jun. 2020, pp. 1–6. doi: 10.1109/SECONWorkshops50264.2020.9149770.

[50] A. Dosovitskiy *et al.*, ‘An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale’. arXiv, Jun. 03, 2021. doi: 10.48550/arXiv.2010.11929.

[51] S. Liu *et al.*, ‘Grounding DINO: Marrying DINO with Grounded Pre-Training for Open-Set Object Detection’. arXiv, Mar. 20, 2023. doi: 10.48550/arXiv.2303.05499.

[52] A. Kirillov *et al.*, ‘Segment Anything’. arXiv, Apr. 05, 2023. doi: 10.48550/arXiv.2304.02643.

[53] W. Lv *et al.*, ‘DETRs Beat YOLOs on Real-time Object Detection’. arXiv, Apr. 17, 2023. doi: 10.48550/arXiv.2304.08069.

[54] ‘Deci-AI/super-gradients’. deci.ai, May 15, 2023. Accessed: May 15, 2023. [Online]. Available: https://github.com/Deci-AI/super-gradients

[55] Y. M. R. da Silva *et al.*, ‘Computer Vision Based Path Following for Autonomous Unmanned Aerial Systems in Unburied Pipeline Onshore Inspection’, *Drones*, vol. 6, no. 12, Art. no. 12, Dec. 2022, doi: 10.3390/drones6120410.

[56] K. Stachowicz, D. Shah, A. Bhorkar, I. Kostrikov, and S. Levine, ‘FastRLAP: A System for Learning High-Speed Driving via Deep RL and Autonomous Practicing’. arXiv, Apr. 19, 2023. doi: 10.48550/arXiv.2304.09831.

[57] A. Agarwal, A. Kumar, J. Malik, and D. Pathak, ‘Legged Locomotion in Challenging Terrains using Egocentric Vision’. arXiv, Nov. 14, 2022. Accessed: May 15, 2023. [Online]. Available: http://arxiv.org/abs/2211.07638

[58] Z. Fu, X. Cheng, and D. Pathak, ‘Deep Whole-Body Control: Learning a Unified Policy for Manipulation and Locomotion’. arXiv, Oct. 18, 2022. doi: 10.48550/arXiv.2210.10044.

[59] P. Yao and Z. Gao, ‘UAV/USV Cooperative Trajectory Optimization Based on Reinforcement Learning’, in *2022 China Automation Congress (CAC)*, Nov. 2022, pp. 4711–4715. doi: 10.1109/CAC57257.2022.10055417.

[60] S. S. A. Haq, ‘Trajectory following using model predictive control for SeaDrone collision avoidance using Robot Operating System’, Master thesis, University of South-Eastern Norway, 2022. Accessed: May 15, 2023. [Online]. Available: https://openarchive.usn.no/usn-xmlui/handle/11250/3011349

[61] W. Li, Y. Ge, and G. Ye, ‘UAV-USV cooperative tracking based on MPC’, in *2022 34th Chinese Control and Decision Conference (CCDC)*, Aug. 2022, pp. 4652–4657. doi: 10.1109/CCDC55256.2022.10034383.

[62] Y. Wu, K. H. Low, and C. Lv, ‘Cooperative Path Planning for Heterogeneous Unmanned Vehicles in a Search-and-Track Mission Aiming at an Underwater Target’, *IEEE Transactions on Vehicular Technology*, vol. 69, no. 6, pp. 6782–6787, Jun. 2020, doi: 10.1109/TVT.2020.2991983.

[63] Y. Yu, Y. Fan, Y. Zhang, D. Mu, and X. Sun, ‘An autonomous collision avoidance system unified with the TFMS and the FCS-MPC strategy for USV’, in *2022 41st Chinese Control Conference (CCC)*, Jul. 2022, pp. 3438–3443. doi: 10.23919/CCC55666.2022.9902408.

[64] F. A. A. Andrade *et al.*, ‘Detection and tracking of crossing vessels for small autonomous vessels equipped with stereo camera’, in *OCEANS 2022 - Chennai*, Feb. 2022, pp. 1–6. doi: 10.1109/OCEANSChennai45887.2022.9775279.

Appendices

Appendix A **Thesis Task Description**

University of South-Eastern Norway

Faculty of Technology, Natural Sciences and Maritime Sciences, Campus Porsgrunn

FMH606 Master's Thesis

Title: Sea Search and Rescue with Autonomous Drones in High-Fidelity Visual and Physical Simulation

USN supervisor: Fabio Andrade

External partner: N/A

Task background:

With the advancements of computer science and the quality of computer hardware, it became possible to build platforms that use Virtual Reality to create environments that are not only able to simulate realistic dynamics but are also visually close to reality. This is a factor of extreme relevance for the development of computer vision and deep learning algorithms.
This kind of environment can be used for generating synthetic data and for optimizing and testing search and rescue systems without the need of real flights, which are costly, demand special operational measures and have a risk of damage to the drone and the plant in case of a crash. Unreal Engine 4 is a gaming engine that with the AirSim plugin by Microsoft is being used for the simulation of drone and rover missions. AirSim also has an interface with ROS (Robot Operating System), that can be used to develop the system.

Task description:
To develop a solution for search and rescue in a post-disaster environment in the sea using drones. This system should be able to detect the area of interest (for example, a disaster ship) and survey this area searching for survivors or other objects. The navigation of the multi-agent drones should be fully autonomous, cooperative and implemented with ROS. Machine Learning models should be used for real-time object detection.

Student category: IIA

Is the task suitable for online students (not present at the campus)? Yes

Practical arrangements:
The simulation platform is heavy and needs a good computer to run. The Autonomous Systems Research Group has a suitable computer in campus Vestfold.

Supervision:
As a general rule, the student is entitled to 15-20 hours of supervision. This includes necessary time for the supervisor to prepare for supervision meetings (reading material to be discussed, etc).

Signatures:

Supervisor (date and signature):
15.01.2023

Student (write clearly in all capitalized letters): RAJEEV POUDEL

Student (date and signature):  15.01.2023

Appendix B **Implementation of *path_planner* node in ROS**

```
#!/usr/bin/env python
import rospy
import actionlib
from math import pi,sin,cos,sqrt
from airsim_ros_pkgs.msg import MoveOnPathAction,MoveOnPathGoal
from geometry_msgs.msg import Point
from std_msgs.msg import String
from airsim_ros_pkgs.msg import ControlDrones
from airsim_ros_pkgs.msg import GoHome



goto_spiralstart_location = True

x_start = 0  # starting location in x for spiral start
y_start = 0  # starting location in y for spiral start
desired_z = -3.05  # desired altitude of the drone during the
mission


if __name__ == '__main__':
    try:

        rospy.init_node('path_planner')

        pub1 =
        rospy.Publisher("/Goto_spiral_start_location",ControlDrones,latch=True
        ,queue_size=1)

        pub2 =
        rospy.Publisher("/spiral_start",ControlDrones,latch=True,queue_size=1)

        msg1 = ControlDrones()

        while goto_spiralstart_location:
            if (pub1.get_num_connections()>0):
                msg1.enable = "Go to spiral start location"
                msg1.x = x_start
                msg1.y = y_start
                msg1.z = desired_z
                pub1.publish(msg1)
                rospy.loginfo("Going to spiral start location!!!!")
                goto_spiralstart_location = False

        while True:
            rospy.sleep(-1)

    except rospy.ROSInterruptException:
        pass
```

Appendix C **Implementation of *PD_Position_Controller* node in ROS**

```
#!/usr/bin/env python
import rospy
import actionlib
from math import pi,sin,cos,sqrt
from airsim_ros_pkgs.msg import MoveOnPathAction,MoveOnPathGoal
from geometry_msgs.msg import Point
from airsim_ros_pkgs.msg import ControlDrones
from airsim_ros_pkgs.srv import Takeoff



single_time = True
enable = ""

x_start = 0
y_start = 0
search_height = 0

start_spiral_search = False

def get_location_cb(msg):
    global enable,x_start,y_start,search_height
    enable = msg.enable
    x_start = msg.x
    y_start = msg.y
    search_height = msg.z
    rospy.loginfo("Moving Drone to the spiral start location!!!")

def setTakeoffMode():
    rospy.wait_for_service('/airsim_node/Drone_1/takeoff')
    try:
        takeoffService = rospy.ServiceProxy('/airsim_node/Drone_1/takeoff',
Takeoff)

        takeoffService(True)
    except rospy.ServiceException as e:
        print ("Service takeoff call failed: %s"%e)


if __name__ == '__main__':
    try:
        # Initializes a rospy node so that the SimpleActionClient can publish
and subscribe over ROS.
        rospy.init_node('PD_Position_Controller')
        sub1 =
rospy.Subscriber("/Goto_spiral_start_location",ControlDrones,get_location_cb)
        rospy.sleep(1)
        pub =
rospy.Publisher('/spiral_start',ControlDrones,latch=True,queue_size=1)
```

```
        pub =
rospy.Publisher('/spiral_start',ControlDrones,latch=True,queue_size=1)


        rospy.loginfo("Inside moveonpath action path-planner_movetolocation
1!!")
        rospy.loginfo("%s",enable)
        while (enable != "Go to spiral start location"):
            rospy.sleep(-1)
        rospy.loginfo("Inside moveonpath action path-planner_movetolocation
2!!")
        if enable == "Go to spiral start location":
            setTakeoffMode()
            msg1 = ControlDrones()
            rospy.loginfo("Inside moveonpath action path-
planner_movetolocation 3!!")
            client = actionlib.SimpleActionClient('/airsim_node/moveOnPath',
MoveOnPathAction)

            client.wait_for_server()
            goal1 = MoveOnPathGoal()
            point1 = Point()
            msg = ControlDrones()

            point1.x = x_start
            point1.y = y_start
            point1.z = search_height
            goal1.point = point1

            goal1.vehicle_name = "Drone_1"

            goal1.velocity = 5.0
            goal1.timeout_sec= sqrt(point1.x**2 + point1.y**2 +
point1.z**2)/goal1.velocity + 60

            goal1.yaw = 0

            client.send_goal(goal1)

            client.wait_for_result()

            if client.get_result():
                start_spiral_search = True
                rospy.loginfo("Starting spiral search")


            while start_spiral_search:
                if (pub.get_num_connections()>0):
```

```
                msg1.enable = "Start Spiral Search"
                msg1.x = x_start
                msg1.y = y_start
                msg1.z = search_height
                pub.publish(msg1)
                rospy.loginfo("Started the spiral search process!!!")
                start_spiral_search = False

        while True:
            rospy.sleep(-1)
except rospy.ROSInterruptException:
    pass
```

Appendix D **Implementation of *Archimedean_Spiral_Path* node in ROS**

```
#!/usr/bin/env python

import rospy
import math
from airsim_ros_pkgs.srv import *
import time

from airsim_ros_pkgs.msg import VelCmd
from airsim_ros_pkgs.msg import ControlDrones
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist


x= 0.0
y= 0.0
z = 0.0

pitch= 0.0
roll = 0.0
yaw = 0.0


def to_eularian_angles(q):
    z = q.z
    y = q.y
    x = q.x
    w = q.w
    ysqr = y * y

    # roll (x-axis rotation)
    t0 = +2.0 * (w*x + y*z)
    t1 = +1.0 - 2.0*(x*x + ysqr)
    roll = math.atan2(t0, t1)

    # pitch (y-axis rotation)
    t2 = +2.0 * (w*y - z*x)
    if (t2 > 1.0):
        t2 = 1
    if (t2 < -1.0):
        t2 = -1.0
    pitch = math.asin(t2)

    # yaw (z-axis rotation)
    t3 = +2.0 * (w*z + x*y)
    t4 = +1.0 - 2.0 * (ysqr + z*z)
    yaw = math.atan2(t3, t4)

    return (pitch, roll, yaw)
```

```
desired_x = 30
desired_y = 10

dx = desired_x
dy = desired_y

desired_yaw = math.atan2(dx,dy)

distance = math.sqrt(dx**2+dy**2)
    #distance_xy = distance * math.cos(math.asin(dz/distance))
distance_cal = 0
n = 1
e_prev_z = 0
e_prev_roll =0
e_prev_pitch = 0
u_roll_prev = 0
u_pitch_prev = 0
ex_prev = 0
ey_prev = 0
ew_prev = 0
count = 0
sum = 0
prev_sum = 0
prev_angle_to_centre = 0

enable = ""
start_x = 0
start_y = 0
desired_z = 0


def spiral_start_cb(msg):
    global enable,start_x,start_y,desired_z
    enable = msg.enable
    start_x = msg.x
    start_y = msg.y
    desired_z = msg.z
    rospy.loginfo("Got the spiral start command!!!")

def listenercb(msg):
    global desired_yaw,e_prev_z,e_prev_roll, e_prev_pitch,
u_roll_prev,u_pitch_prev,ex_prev,ey_prev,ew_prev,count,sum,prev_angle_to_centr
e,start_x,start_y,desired_z

    if enable == "Start Spiral Search":
        vel = VelCmd()
        x = msg.pose.pose.position.x
        y = msg.pose.pose.position.y
        z = msg.pose.pose.position.z
```

```
pitch,roll,yaw = to_eularian_angles(msg.pose.pose.orientation)

#a = 3.0
b = 8 / (2*math.pi)

dx = x - start_x
dy = y - start_y
angle_to_centre=  math.atan2(dx,dy)
angle_to_centre_degree = angle_to_centre * 180 / math.pi
w =  3 * math.pi/180

actual_radius = math.sqrt(dx**2+dy**2)

next_angle = (w + angle_to_centre)

count = int(sum/(2*math.pi))



if next_angle >= 0 and next_angle <= math.pi/2:
    present_angle = next_angle
    sum = sum + abs(present_angle - prev_angle_to_centre)
    prev_angle_to_centre = present_angle
    #sum = sum + w
    next_radius = b*sum
    next_x = next_radius*math.sin(present_angle) #First Quadrant
    next_y = next_radius*math.cos(present_angle)

elif next_angle > math.pi/2  and next_angle <= math.pi:
    present_angle = math.pi - next_angle
    sum = sum + abs(present_angle - prev_angle_to_centre)
    prev_angle_to_centre = present_angle
    #sum = sum + w
    next_radius = b*sum
    next_x = next_radius*math.sin(present_angle) #Second Quadrant
    next_y = -next_radius*math.cos(present_angle)

elif next_angle >= -math.pi and next_angle < - math.pi/2:
    present_angle = math.pi + next_angle
    sum = sum + abs(present_angle - prev_angle_to_centre)
    prev_angle_to_centre = present_angle
    #sum = sum + w
    next_radius = b*sum
    next_x = -next_radius*math.sin(present_angle) # Third Quadrant
    next_y = -next_radius*math.cos(present_angle)


elif next_angle >= -math.pi/2 and next_angle < 0:
    present_angle = abs(next_angle)
```

```
            sum = sum + abs(present_angle - prev_angle_to_centre)
            prev_angle_to_centre = present_angle
            #sum = sum + w
            next_radius = b*sum
            next_x = -next_radius*math.sin(present_angle) ##Fourth Quadrant
            next_y = next_radius*math.cos(present_angle)

        elif next_angle < - math.pi:
            present_angle = abs(next_angle + math.pi)
            sum = sum
            prev_angle_to_centre = present_angle
            #sum = sum + w
            next_radius = b*sum
            next_x = next_radius*math.sin(present_angle) # Second Quadrant
            next_y = -next_radius*math.cos(present_angle)

        elif next_angle > math.pi:
            present_angle = abs(next_angle - math.pi)
            sum = sum
            prev_angle_to_centre = present_angle
            #sum = sum + w
            next_radius = b*sum
            next_x = -next_radius*math.sin(present_angle) # Third Quadrant
            next_y = -next_radius*math.cos(present_angle)

        else:
            pass

        time = rospy.Time.now().to_sec()

        rospy.loginfo("z: %f, next_x:%f, x:%f,
next_y:%f,y:%f,angle_to_centre:%f,next_angle:%f,radius:%f, next_radius:%f,
sum:%f,spiral
count:%d",z,next_x,x,next_y,y,angle_to_centre_degree,next_angle*180/math.pi,ac
tual_radius,next_radius,sum,count)

        #PD Control for z
        Kp_p = 2
        Td = 5

        ez = desired_z - z
        uz = Kp_p*ez + Kp_p*Td*(ez - e_prev_z)
        e_prev_z = ez


        #PID Control
        Kp_angle = 2
        Ti = 10
```

```
        e_roll = 0 - roll
        u_roll = u_roll_prev + Kp_angle *(e_roll - e_prev_roll) +
(Kp_angle/Ti)*0.01*e_roll
        e_prev_roll = e_roll
        u_roll_prev = u_roll

        e_pitch = 0 - pitch
        u_pitch = u_pitch_prev + Kp_angle*(e_pitch - e_prev_pitch) +
(Kp_angle/Ti)*0.01*e_pitch
        e_prev_pitch = e_pitch
        u_pitch_prev = u_pitch


        #Velocity control
        Kp = 1 #1
        Kd = 0.5 #0.5
        ex = next_x - x
        vx = Kp * ex + Kd * (ex - ex_prev)
        ex_prev = ex
        ey = next_y - y
        vy = Kp * ey + Kd*(ey-ey_prev)
        ey_prev = ey



        vel.twist.linear.x = vx ; vel.twist.linear.y = vy ; vel.twist.linear.z
= uz
        vel.twist.angular.x = -u_roll; vel.twist.angular.y = -u_pitch;
vel.twist.angular.z = 0

        pub.publish(vel)


if __name__=="__main__":
    try:
        rospy.init_node("Archimedean_Spiral_Path",anonymous=True)
        start_time = rospy.Time.now().to_sec()
        pub =
rospy.Publisher('/airsim_node/Drone_1/vel_cmd_body_frame',VelCmd,
queue_size=1)
        sub =
rospy.Subscriber('/airsim_node/Drone_1/odom_local_ned',Odometry,listenercb)
        sub1 = rospy.Subscriber('/spiral_start',
ControlDrones,spiral_start_cb)
        rospy.sleep(1)
        rospy.spin()

    except rospy.ROSInterruptException:
        pass
```

Appendix E **Implementation of *yolov8_sea* node in ROS**

```
#!/usr/bin/env python
import cv2
import torch
import random

import rospy

from cv_bridge import CvBridge

from ultralytics import YOLO

from sensor_msgs.msg import Image
from vision_msgs.msg import Detection2D
from vision_msgs.msg import ObjectHypothesisWithPose
from vision_msgs.msg import Detection2DArray

class Yolov8Node():

    def __init__(self) -> None:


        self.device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
        # params
        # self.model = rospy.get_param("model", "yolov8x.pt")

        # self.tracker = rospy.get_param("tracker", "bytetrack.yaml")

        # self.img_topic = rospy.get_param("img_topic",
"/airsim_node/Drone_1/camera_1/Scene")


        # self.threshold = rospy.get_param("threshold", 0.5)

        # self.enable = rospy.get_param("enable", True)

        # # params
        self.model1 = "yolov8x.pt"
        self.model2 = "yolov8x-seg.pt"
        self.bool_yolo_seg = True

        self.img_topic = "/airsim_node/Drone_1/camera_1/Scene"

        self.threshold =  0.5

        self.enable =  True

        self._class_to_color = {}
```

```
        self.cv_bridge = CvBridge()
        if self.bool_yolo_seg:
            self.yolo = YOLO(self.model2)
            self.yolo.fuse()
        else:
            self.yolo = YOLO(self.model1)
            self.yolo.fuse()
        rospy.sleep(1)
        self.yolo.to(self.device)

        # topics
        self._pub = rospy.Publisher("detections",Detection2DArray, queue_size=
10)
        self._detection_image_pub = rospy.Publisher("detection_image", Image,
queue_size= 10)
        rospy.sleep(1)
        self._sub = rospy.Subscriber(self.img_topic, Image, self.image_cb)


    def image_cb(self, msg: Image) -> None:

        if self.enable:

            # convert image + predict + track
            cv_image = self.cv_bridge.imgmsg_to_cv2(msg)

            results = self.yolo.track(source=cv_image,show=False,
verbose=False, tracker="bytetrack.yaml")


            # create detections msg
            detections_msg = Detection2DArray()
            detections_msg.header = msg.header

            results = results[0].cpu()


            for b in results.boxes:

                label = self.yolo.names[int(b.cls)]
                score = float(b.conf)

                if score < self.threshold:
                    continue

                detection = Detection2D()

                detection.header = msg.header
```

```
                detection.source_img = msg

                box = b.xywh[0]

                # get boxes values
                detection.bbox.center.x = float(box[0])
                detection.bbox.center.y = float(box[1])
                detection.bbox.size_x = float(box[2])
                detection.bbox.size_y = float(box[3])

                # get track id
                track_id = 0
                if not b.id is None:
                    track_id = int(b.id)
                #detection.id = track_id


                # get hypothesis
                hypothesis = ObjectHypothesisWithPose()
                hypothesis.id = int(b.cls)
                hypothesis.score = score
                hypothesis.pose.pose.position.x = float(track_id)
                detection.results.append(hypothesis)

                # draw boxes for debug
                if label not in self._class_to_color:
                    r = random.randint(0, 255)
                    g = random.randint(0, 255)
                    b1 = random.randint(0, 255)
                    self._class_to_color[label] = (r, g, b1)
                color = self._class_to_color[label]

                min_pt = (round(detection.bbox.center.x -
detection.bbox.size_x / 2.0),
                          round(detection.bbox.center.y -
detection.bbox.size_y / 2.0))
                max_pt = (round(detection.bbox.center.x +
detection.bbox.size_x / 2.0),
                          round(detection.bbox.center.y +
detection.bbox.size_y / 2.0))
                cv_image = cv2.rectangle(cv_image, min_pt, max_pt, color, 2)

                label = "{}:({}) {:.3f}".format(label, str(track_id), score)
                pos = (min_pt[0], max(15, int(min_pt[1] - 10)))
                font = cv2.FONT_HERSHEY_SIMPLEX
                cv_image = cv2.putText(cv_image, label, pos, font,
                            0.5, color, 1, cv2.LINE_AA)
```

```
                # append msg
                detections_msg.detections.append(detection)

            # publish detections and dbg image
            self._pub.publish(detections_msg)


            if self.bool_yolo_seg:
                annotated_results = results[0].plot(conf=False, labels =
False, img= cv_image,  boxes = False,masks = True)



                self._detection_image_pub.publish(self.cv_bridge.cv2_to_imgmsg
(annotated_results, encoding=msg.encoding))
                cv2.imshow("Real-Time Detection with Tracking and
Segmentation",annotated_results)
                cv2.waitKey(1)
            else:
                self._detection_image_pub.publish(self.cv_bridge.cv2_to_imgmsg
(cv_image,encoding=msg.encoding))
                cv2.imshow("Real-Time Detection with Tracking",cv_image)
                cv2.waitKey(1)

            if rospy.is_shutdown():
                cv2.destroyAllWindows()

if __name__ == "__main__":
    rospy.init_node("yolov8_sea", anonymous= True)
    Yolov8Node()
    rospy.spin()
```

Appendix F **Implementation of *Georeferencing* node in ROS**

```
#!/usr/bin/env python
import rospy
import message_filters
from nav_msgs.msg import Odometry
from vision_msgs.msg import Detection2DArray
from airsim_ros_pkgs.msg  import Altimeter
from geometry_msgs.msg import Point
from airsim_ros_pkgs.msg  import IDandLocation
from airsim_ros_pkgs.msg import VelCmd
from airsim_ros_pkgs.msg import ControlDrones
from math import tan,pi


x_det = 0
y_det = 0
ex_prev = 0
ey_prev = 0
e_prev_z = 0
desired_z = 1.5

stop_spiral_search = True

#PD Control for tracking
Kp = 0.5
Kd = 2

#PD Control for height
Kp_z = 0.5
Kd_z = 2

 #Parameters
image_width = 640
image_height = 640
HFOV = 90 * pi/180
z_correction = 3.28366



def callback(odom,bbox):
    global x_det,y_det,ex_prev,ey_prev,Kp,Kd,e_prev_z,desired_z,im-
age_width,image_height,HFOV,z_correction,Kp_z,Td_z

    try:
        only_person = 0
        if len(bbox.detections) > 1:
        #     rospy.loginfo("length:%d",len(bbox.detections))
            for i,det in enumerate(bbox.detections):
                if (det.results[0].id == 0) and (det.results[0].pose.pose.po-
sition.x == 1.0):
```

```
                    only_person = i
                    break

        else:
            only_person = 0


        class_id = bbox.detections[only_person].results[0].id
        track_id = int(bbox.detections[only_person].results[0].pose.pose.posi-
tion.x)

        if (class_id == 0) and (track_id == 1) :

            while stop_spiral_search:
                if (spiral_pub.get_num_connections()>0):
                    msg1 = ControlDrones()
                    msg1.enable = "Stop Spiral Search"
                    msg1.x = 0
                    msg1.y = 0
                    msg1.z = 0
                    spiral_pub.publish(msg1)
                    rospy.loginfo("Stopped the spiral search process!!!")
                    stop_spiral_search = False

            rospy.loginfo("Inside Georeferencing")

            vel = VelCmd()
            IDandLoc = IDandLocation()

            x = odom.pose.pose.position.x
            y = odom.pose.pose.position.y
            z = odom.pose.pose.position.z

            x_center = bbox.detections[0].bbox.center.x
            y_center =  bbox.detections[0].bbox.center.y

            H = abs(z - z_correction)
            width_ned = 2*H*tan(HFOV/2)
            height_ned = image_height/image_width * width_ned

            detection_width_center_ned = (x_center/image_width) * width_ned
            detection_height_center_ned = ((image_height-y_center)/im-
age_height) * height_ned

            horizontal_correction = detection_height_center_ned - height_ned/2
            vertical_correction = detection_width_center_ned - width_ned/2
```

```
            x_det = x + horizontal_correction
            y_det = y + vertical_correction

            IDandLoc.point.x = x_det
            IDandLoc.point.y = y_det
            IDandLoc.point.z = z_correction
            IDandLoc.ID = track_id

            #PD Control for z
            ez = desired_z - z
            uz = Kp_z*ez + Kp_z*Td_z*(ez - e_prev_z)
            e_prev_z = ez

            #PD Control for position tracking
            ex = x_det - x
            vx = Kp * ex + Kd * (ex - ex_prev)
            ex_prev = ex
            ey = y_det - y
            vy = Kp * ey + Kd*(ey-ey_prev)
            ey_prev = ey
            rospy.loginfo("vx:%f,vy:%f,uz:%f,current_z:%f",vx,vy,uz,z)
            vel.twist.linear.x = vx ; vel.twist.linear.y = vy ;
vel.twist.linear.z = uz
            vel.twist.angular.x = 0; vel.twist.angular.y = 0;
vel.twist.angular.z = 0

            pub.publish(vel)
            victim_pub.publish(IDandLoc)

    except IndexError:
        rospy.loginfo("Error getting detections!!")

if __name__ == "__main__":
    rospy.init_node("Georeferencing",anonymous=True)
    pub = rospy.Publisher('/airsim_node/Drone_1/vel_cmd_body_frame',VelCmd,
queue_size=1)
    victim_pub =
rospy.Publisher("/victim_location_ned",IDandLocation,queue_size=1)
    spiral_pub =
rospy.Publisher("/spiral_start",ControlDrones,latch=True,queue_size=1)
    odom_sub =
message_filters.Subscriber('/airsim_node/Drone_1/odom_local_ned', Odometry)
    bboxes_sub = message_filters.Subscriber('/detections', Detection2DArray)
    approx =
message_filters.ApproximateTimeSynchronizer([odom_sub,bboxes_sub],1000,slop=0.
1)
    approx.registerCallback(callback)
    rospy.spin()
```

Appendix G **Implementation of *Victim_Geolocation* node in ROS**

```
#!/usr/bin/env python

import rospy

from geometry_msgs.msg import Point
from airsim_ros_pkgs.msg  import IDandLocation
from airsim_ros_pkgs.srv import ReturnGeolocation


def ReturnGeoLocationfromNed_client(ned_point):
    rospy.wait_for_service('/airsim_node/Drone_1/returngeolocationfromned')
    try:
        returnGeolocationfromnedService =
rospy.ServiceProxy('/airsim_node/Drone_1/returngeolocationfromned',
ReturnGeolocation)

        response = returnGeolocationfromnedService(ned_point)
        return response.outputpoint
    except rospy.ServiceException as e:
        print ("Service returngeolocationfromned call failed: %s"%e)

def geolocation_cb(IDandned_point):
    gps_location = Point()
    #rospy.loginfo("x_det:%f, y_det:%f,
z:%f",IDandned_point.point.x,IDandned_point.point.y, IDandned_point.point.z)
    gps_location = ReturnGeoLocationfromNed_client(IDandned_point.point)
    rospy.loginfo("Person_ID:%d,Latitude:%f, Longitude:%f,
Altitude:%f",IDandned_point.ID,gps_location.x,gps_location.y, gps_location.z)


if __name__ == "__main__":

    rospy.init_node("Victim_Geolocation",anonymous=True)

    sub1 = rospy.Subscriber("/victim_location_ned",IDandLocation,
geolocation_cb, queue_size=100)

    rospy.spin()
```

Appendix H **Main AirSim Settings**

```
{
  "ClockSpeed": 1,
  "SettingsVersion": 1.2,
  "SimMode": "Multirotor",

  "Vehicles": {
    "Drone_1": {
      "VehicleType": "SimpleFlight",
      "DefaultVehicleState": "Disarmed",

      "Cameras": {
        "camera_1": {
          "CaptureSettings": [
            {
              "ImageType": 0,
              "Width": 640,
              "Height": 640,
              "FOV_Degrees": 90
            }
          ],
          "Gimbal": {
            "Stabilization": 1,
            "Pitch": -90.0, "Roll": 0, "Yaw": 0
          },
          "X": 0.0, "Y": 0.0, "Z": 0.0,
          "Pitch": 0.0, "Roll": 0.0, "Yaw": 0.0
        }
      },
      "X": 0.0, "Y": 0.0, "Z": 0,
      "Pitch": 0.0, "Roll": 0, "Yaw": 0
    }
  },
  "OriginGeopoint": {
    "Latitude": 57.963589,
    "Longitude": 9.130108,
    "Altitude": 122
  }
}
```