Technical University of Crete

School of Electrical and Computer Engineering

Diploma Thesis

Uncrewed Cargo Ship Virtual Reality Simulator

by

Epameinondas Chrysis

Thesis Committee

Professor Katerina Mania (ECE)
Professor Micail G. Lagoudakis (ECE)
Professor Nikolaos Giatrakos (ECE)

Chania, Crete
October 2024

Πολυτεχνείο Κρήτης

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

Προσομοιωτής Φορτηγού Πλοίου χωρίς Πλήρωμα σε Εικονική Πραγματικότητα

του

Επαμεινώνδα Χρυσή

Τριμελής Επιτροπή

Καθηγήτρια Κατερίνα Μανιά (ΗΜΜΥ)
Καθηγητής Μιχαήλ Γ. Λαγουδάκης (ΗΜΜΥ)
Καθηγητής Νικόλαος Γιατράκος (ΗΜΜΥ)

Χανιά, Κρήτη
Οκτώβριος 2024

Abstract

The increasing shift towards unmanned maritime vessels necessitates advanced training tools to prepare remote ship navigators for the complexities of operating cargo ships from a distance. This thesis presents a Virtual Reality (VR) simulator designed to contribute to the education and skill development of remote vessel controllers. The simulator provides a realistic depiction of the control room environment that navigators encounter, complete with various technologies and instruments they will use in real-world scenarios. The core of the simulator is physics-based, accurately modeling the movement of the vessel by simulating buoyancy, weight, drag forces, and propulsion. The system is equipped with a comprehensive array of screens displaying sensor data and a variety of buttons, levers, and electronic systems that simulate the operational environment of a remote control room. The sensors provide critical measurements on various aspects of the ship's status, ensuring that users receive all the necessary information to navigate and operate the vessel effectively. These features are designed to provide a realistic and immersive experience, allowing users to interact with the virtual ship as they would in a real-world scenario. The VR integration elevates the user experience, providing vivid and immersive interactions that closely mimic real-world conditions. Technologies such as Unity, OpenXR, and the Meta Quest Pro VR headset were utilized to achieve a high degree of realism. The simulator was developed using C sharp scripting for precise control of interactions, while the OpenXR framework enabled seamless integration of VR functionalities. Users can choose between a free-roam mode and a scenario-based mode, where they must navigate from one port to another within a specified time frame. The scenario mode includes customizable environmental conditions, adding layers of complexity to the training experience. Evaluation of the system focused on user feedback regarding the VR interactions and their effectiveness in providing a realistic and engaging training tool. The study also assessed the users' ability to complete the navigation scenarios without collisions, under varying environmental conditions. The results suggest that the VR-based simulator is a valuable tool for educating future remote ship navigators, offering an immersive and practical learning environment that can be tailored to a wide range of training needs.

Περίληψη

Ο στόχος αυτής της διπλωματικής εργασίας ήταν η ανάπτυξη ενός προσομοιωτή Εικονικής Πραγματικότητας (VR) υψηλής πιστότητας για την εκπαίδευση απομακρυσμένων χειριστών πλοίων στον έλεγχο μη επανδρωμένων φορτηγών πλοίων. Το έργο ανταποκρίνεται στην αυξανόμενη ανάγκη για προηγμένα εργαλεία εκπαίδευσης, καθώς η ναυτιλιακή βιομηχανία στρέφεται προς την αυτοματοποίηση. Ο προσομοιωτής γεφυρώνει το θεωρητικό υπόβαθρο με την πρακτική εφαρμογή, παρέχοντας ένα ρεαλιστικό και ασφαλές περιβάλλον εκπαίδευσης για πολύπλοκα συστήματα πλοίων. Ο προσομοιωτής αναπτύχθηκε χρησιμοποιώντας την πλατφόρμα Unity και εργαλεία όπως τα OpenXR, XR Origin και XR Interaction Toolkit, ενώ χρησιμοποιεί το VR headset Meta Quest Pro, προσφέροντας εξαιρετική οπτική ποιότητα και ακριβή παρακολούθηση των κινήσεων των χεριών. Η εφαρμογή προσομοιώνει το φορτηγό πλοίο Maersk Honam, απεικονίζοντας με ακρίβεια τις διαστάσεις και τη δυναμική του πλοίου. Δυνάμεις όπως η άνωση, η αντίσταση και η προώθηση ενσωματώθηκαν για να διασφαλίσουν ρεαλιστική συμπεριφορά του πλοίου. Παρόλο που το Maersk Honam είναι συμβατικό πλοίο, χρησιμεύει ως ένα ισχυρό μοντέλο για την κατανόηση του ελέγχου μεγάλων φορτηγών πλοίων. Ο προσομοιωτής προσφέρει δύο λειτουργίες: τη λειτουργία ελεύθερης εξερεύνησης για την κατανόηση των συστημάτων του πλοίου και τη λειτουργία σεναρίου για προκλήσεις πλοήγησης, όπου οι χρήστες πρέπει να διαχειριστούν περιβαλλοντικές συνθήκες και να αποφύγουν συγκρούσεις. Προσαρμόσιμα στοιχεία καιρικών συνθηκών, όπως η βροχή και τα κύματα, ενισχύουν περαιτέρω τον ρεαλισμό, βοηθώντας τους εκπαιδευόμενους να αποκτήσουν μια ολοκληρωμένη κατανόηση της απομακρυσμένης πλοήγησης. Η κύρια συνεισφορά αυτής της διπλωματικής είναι η ανάπτυξη ενός ολοκληρωμένου προσομοιωτή βασισμένου στην Εικονική Πραγματικότητα, που βυθίζει τους χρήστες σε ένα ρεαλιστικό περιβάλλον για ασφαλή και αποτελεσματική εκπαίδευση. Η έρευνα αναδεικνύει τις δυνατότητες της VR στην εκπαίδευση στη ναυτιλία, προσφέροντας μια ασφαλέστερη εναλλακτική στα παραδοσιακά εκπαιδευτικά μέσα και συμβάλλοντας στη βέλτιστη διαχείριση των μη επανδρωμένων πλοίων.

Acknowledgements

Initially, I would like to thank my supervisor, Professor Katerina Mania, for her advice and support throughout the whole development of the thesis and her trust in me for undertaking this challenge on combining Virtual Reality and unmanned vessels.

I am sincerely appreciative to the members of the Surreal team, whose continuous feedback and unwavering support were instrumental throughout the daily development process. Their insights and encouragement were crucial in shaping this work.

Finally, I would like to express my deepest gratitude to my girlfriend, Maria C, and my friends for their unwavering support and encouragement during the challenging months dedicated to this thesis. I am profoundly thankful to my father, Konstantinos, my mother, Spyridoula, and my sister, Dimitra, for their steadfast belief in me and for the countless ways they have supported me during this journey. Their insights, ideas, and assistance have been invaluable, and I am truly grateful for their presence and encouragement at every step of the way.

Contents

List of Figures

Chapter 1

Introduction

1.1 Brief Introduction

The integration of VR into the maritime sector offers unprecedented opportunities for enhancing the training of remote vessel controllers. By simulating real-world conditions within a virtual environment, VR allows trainees to experience and interact with the complexities of navigating large vessels without the associated risks. This immersive approach is particularly valuable for preparing navigators to manage unmanned cargo ships, where the ability to monitor and control a vessel remotely is paramount [1].

As the demand for unmanned vessels grows, so does the need for advanced training tools that can provide realistic and comprehensive simulations. VR offers a unique platform that goes beyond traditional training methods, enabling users to engage with virtual environments that closely replicate the challenges of remote navigation. Through the use of physics-based simulations and a variety of virtual instruments, including sensors, control systems, and interactive displays, VR facilitates a deeper understanding of ship operations and enhances the skill set required for effective remote navigation.

This thesis focuses on the development of a VR-based simulator designed to train and educate future remote ship navigators. By providing a realistic depiction of a remote control room and simulating the physical forces acting on a vessel, the simulator aims to bridge the gap between theoretical knowledge and practical application. The immersive nature of VR, combined with the detailed modeling of ship dynamics, creates a powerful tool for exploring the intricacies of unmanned cargo ship navigation.

1.2 Purpose of the Thesis

This thesis seeks to harness the transformative capabilities of Virtual Reality (VR) technology in the maritime industry, specifically in the training and education of remote vessel navigators. The primary objective is to bridge the gap between the theoretical knowledge of remote ship operation and the practical experience required to navigate unmanned cargo ships safely and efficiently. VR technology, with its immersive and interactive environment, provides an unparalleled platform to simulate the complexities of maritime navigation in ways that traditional training methods cannot.

The central purpose of this thesis is to explore the use of high-fidelity VR simulations to create realistic and interactive training environments for remote ship controllers. By leveraging advanced VR technologies, the thesis aims to equip future navigators with the necessary skills and experience to operate unmanned vessels effectively. The simulation incorporates detailed physics modeling and realistic control interfaces, allowing users to interact with the virtual ship as they would in a real-world scenario.

Furthermore, this thesis provides a comprehensive overview of the research objectives and methodology underlying the development of the VR-based simulator. It explores the integration of key navigational tools and control systems into the virtual environment, ensuring that the training experience is as close to reality as possible.

Moreover, this research aims to highlight the potential benefits of using VR in maritime training. By allowing trainees to immerse themselves in a fully interactive simulation, it holds the potential to significantly improve the quality of training, reduce the risk of accidents, and prepare navigators for the challenges of remote ship operation. Through the use of VR, this study this study seeks to explore novel avenues for enhanced precision, interactivity, and visualization, with the ultimate goal of contributing to a deeper understanding of the challenges and intricacies involved in remote vessel navigation. By integrating these elements into a VR-based simulator, this thesis aims to set a new standard for the training of remote ship navigators, ensuring they are well-prepared to manage the complexities of unmanned cargo ship operations in a safe and controlled environment.

1.3 Brief Description

In this thesis, we developed a virtual reality simulator for the remote operation of an unmanned cargo ship. The 3d model used is a simulation of the ship Maersk Honam although this ship is not an unmanned. In reality it is a classic cargo ship. The application simulates the control and navigation of the vessel within a virtual reality environment, providing an immersive training experience for remote ship navigators. The Meta Quest Pro VR headset was used to present the virtual environment, offering high-quality visuals and intuitive hand-tracking features to enhance user interaction with the control systems. The Meta Quest Pro, known for its exceptional visual fidelity and hand-tracking capabilities, allowed for realistic and intuitive interactions in the control room simulation. This setup provides users with a highly immersive experience, enabling them to manage the ship's systems and navigation seamlessly [3].

The application was developed using Unity and transitioned into a VR environment utilizing Unity's VR tools such as the OpenXR standard, XR Origin, and XR Interaction Toolkit. These tools ensured the simulator could integrate smoothly with the chosen VR hardware, maximizing immersion and interactivity. Within the simulation, users interact with a fully modeled version of the Maersk Honam, complete with the ship's real

dimensions and accurate physics modeling. Forces such as buoyancy, resistance, weight, and propulsion were integrated into the simulation to ensure realistic ship behavior. While the ship's propulsion force mirrors the real-world data, certain adjustments, like weight, were made to optimize performance within the virtual environment.

The simulator offers two distinct modes: free roam mode, where users can explore the ship's systems and environment without specific objectives, and scenario mode, where users must navigate the ship from one port to another while avoiding collisions, managing time pressure, and accounting for environmental conditions. Users can customize the weather conditions, such as rain, waves, and visibility, further enhancing the realism of the experience.

The primary contribution of this thesis is the development of a comprehensive VR-based training simulator for unmanned cargo ship navigation, allowing users to interact with realistic ship systems and experience the complexities of maritime navigation in a safe, virtual environment.

Figure 1.1: Operator's view from the control room

1.4 Structure of the Thesis

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

- **Chapter 1: Introduction**

This chapter presents the development of a virtual reality simulator for the unmanned cargo ship Maersk Honam, aimed at providing a realistic training environment for remote ship navigation. The chapter discussed the hardware and software technologies used, including the Meta Quest Pro VR headset and Unity game engine, along with the simulator's key features such as free roam and scenario modes, which allow users to explore and navigate the ship in varying conditions. The chapter also highlighted the overall

contributions of the simulator in advancing maritime training by offering an immersive and interactive virtual experience.

- **Chapter 2: Research Overview**

Chapter 2 describes the research conducted for the development of the VR-based unmanned cargo ship simulator. This chapter begins with an overview of the history and types of unmanned cargo ships, followed by a discussion on the information presented to remote ship controllers and the forces acting on a ship, including how they influence the vessel's buoyancy and navigation. The chapter then examines the role of unmanned vessel simulators in providing effective training for remote operators, emphasizing their importance in the maritime sector.The history and evolution of virtual reality (VR) technology is explored, with a focus on its immersive and interactive capabilities. A comparison of the three leading Head Mounted Displays (HMDs) is provided, leading to an explanation of why the Meta Quest Pro was selected for this project. Lastly, an overview of existing VR applications and simulators is conducted, with particular attention to those relevant to the maritime sector, providing a foundation for understanding the simulator's unique contributions to the field.

- **Chapter 3: Technological Background**

Chapter 3 delves into the technological foundations employed in the thesis. It offers an in-depth examination of the Unity game engine, highlighting its key features and capabilities. Additionally, this chapter presents an overview of the physics and mathematical equations applied throughout the development process.

- **Chapter 4: Users View**

Chapter 4 offers an in-depth exploration of the application and its functionalities. It outlines the various tasks users can perform within the 3D environment, including interacting with menus, user interfaces, and 3D features such as buttons, levers, and wheels. Additionally, use case diagrams will be provided for each interface to illustrate their functions.

- **Chapter 5: Implementation**

Chapter 5 details the technical implementation of the project, building on the tools and information introduced in earlier sections. It covers the integration with the Meta Quest Pro, the ship's movement, and the development of the physics system. Additionally, each electronic system is explained in detail, along with how it was implemented. Furthermore different parts are shown such as interactions and weather features.

- **Chapter 6: Evaluation, Results, Future work**

Chapter 6 presents the effectiveness of the VR in maritime sector, and especially of a

USV simulator, and how well the user performed. Lastly the conclusions are shown up as well as the suggestions for future work.

Chapter 2

Research Overview

2.1 Introduction

The rapid advancements in maritime automation have brought significant attention to unmanned cargo ships, particularly their potential to revolutionize the industry by reducing labor costs and improving operational efficiency. This chapter delves into the foundational research that informed the development of the virtual reality (VR) simulator for the Maersk Honam. It begins with a historical overview of unmanned cargo ships, exploring the evolution of this technology and its various types. The role of remote ship controllers and the information they rely on for vessel navigation are discussed, with particular emphasis on the forces acting on a ship, such as buoyancy, drag, and propulsion. These forces not only affect ship performance but are crucial for the training of remote operators, highlighting the importance of accurate simulation in a training environment. Further, the chapter provides a comprehensive review of unmanned vessel simulators and their role in preparing remote ship operators for the complexities of modern maritime navigation. In addition, the history and evolution of VR technology is examined, focusing on its capabilities to offer immersive and interactive experiences. The Meta Quest Pro, selected for its advanced features and superior visual fidelity, is compared to other Head Mounted Displays (HMDs) in this context, highlighting the rationale behind its integration into this project. Lastly, this chapter concludes with an overview of existing VR applications and simulators, particularly those relevant to the maritime sector, setting the foundation for understanding the unique contributions of our unmanned cargo vessel VR simulator.

2.2 Backround on Unmanned Surface Vehicles

2.2.1 Definition

Unmanned Surface Vehicles (USVs) are autonomous or remotely operated watercraft that navigate and perform tasks on the surface of the water without the need for onboard human crew [4]. They are equipped with sensors, navigation systems, and communication tools to carry out various missions, such as environmental monitoring, maritime security,

defense, and commercial operations like cargo transportation. USVs can be controlled from a remote location or operate autonomously based on pre-programmed instructions and onboard artificial intelligence (AI) systems.

2.2.2 History of USVs

The development of Unmanned Surface Vehicles (USVs) has its origins in the early 20th century, primarily within military and defense sectors. Initial concepts for USVs were linked to remote-controlled naval vessels used during World War II, where rudimentary radio-controlled boats were developed for mine sweeping and target practice. However, these early efforts were limited by the technology of the time and did not see widespread use [5].

It was not until the late 20th century that advancements in computing, navigation systems, and communication technologies began to significantly influence the development of modern USVs. During the 1980s and 1990s [6], USVs began to evolve, particularly in the context of naval research. These early USVs were primarily used for tasks such as surveillance, environmental monitoring, and mine detection. They were still mostly controlled by operators through radio signals, but they laid the foundation for more advanced autonomous systems.

The early 21st century saw rapid advancements in automation, artificial intelligence (AI), and sensor technology, which propelled the evolution of USVs. As autonomy levels increased, USVs started to be equipped with sophisticated navigation systems and AI-based decision-making capabilities, allowing them to perform more complex missions without direct human intervention. This shift enabled the use of USVs in a broader range of applications, including commercial shipping, environmental monitoring, and offshore oil and gas exploration.

Today, USVs are at the forefront of maritime innovation, with cutting-edge systems that incorporate advanced AI, LIDAR, radar, and sonar technologies [6]. They are capable of operating autonomously for extended periods, conducting tasks such as data collection, security patrols, and even cargo transportation. As industries continue to explore the benefits of unmanned systems, USVs have become a key player in the ongoing shift toward automation in the maritime sector.

2.2.3 Types of USVs and their uses

Unmanned Surface Vehicles (USVs) come in a variety of types, each designed for specific purposes based on their size, capabilities, and level of autonomy. These vehicles are deployed in both military and civilian sectors, fulfilling tasks that range from naval operations to environmental monitoring and scientific research. Below are the main types of USVs and their typical uses:

1. Military USVs are perhaps the most developed category, with applications in mine detection, reconnaissance, and combat operations [7]. These USVs are often equipped with advanced sensors, sonar, and weaponry systems. An example is the Seahawk USV used by the U.S. Navy for surveillance and combat. These USVs can operate autonomously or remotely, allowing naval forces to perform dangerous missions such as mine clearing without risking human lives. They can also carry out patrolling and intelligence gathering in contested waters.

Figure 2.1: Sea Hunter an USV of the U.S. Navy

2. Scientific and Environmental Monitoring USVs. Another important class of USVs is those used for scientific research and environmental monitoring [4, 7]. These vessels are usually designed to collect data over long periods and vast ocean areas. USVs like the Saildrone and DriX are commonly employed for mapping the ocean floor, studying marine ecosystems, and monitoring climate change. They can be equipped with sensors to collect data on water quality, temperature, and salinity, and they often operate on renewable energy sources like wind and solar power, allowing for extended missions.

Figure 2.2: USV to research water pollution

3. Commercial USVs. In the commercial sector, USVs are increasingly being used for logistical tasks such as cargo transport, particularly for short sea shipping or port operations. These vessels, like the Maersk Honam, represent a new frontier in unmanned logistics, reducing the need for human crew and enhancing the efficiency of shipping operations [4]. USVs in this category are also employed for offshore oil and gas exploration, providing an efficient and safer alternative to traditional manned vessels in potentially hazardous environments.

Figure 2.3: Yara Birkeland cargo USV

4. Survey and Inspection USVs used for survey and inspection purposes are typically equipped with high-resolution sonar and LIDAR systems to perform detailed surveys of underwater structures, pipelines, or maritime infrastructure [4, 8]. They are crucial in industries such as offshore wind farms, where underwater inspections are necessary for maintenance and construction. These USVs can operate in shallow waters and perform tasks with high precision, reducing the need for costly manned operations.

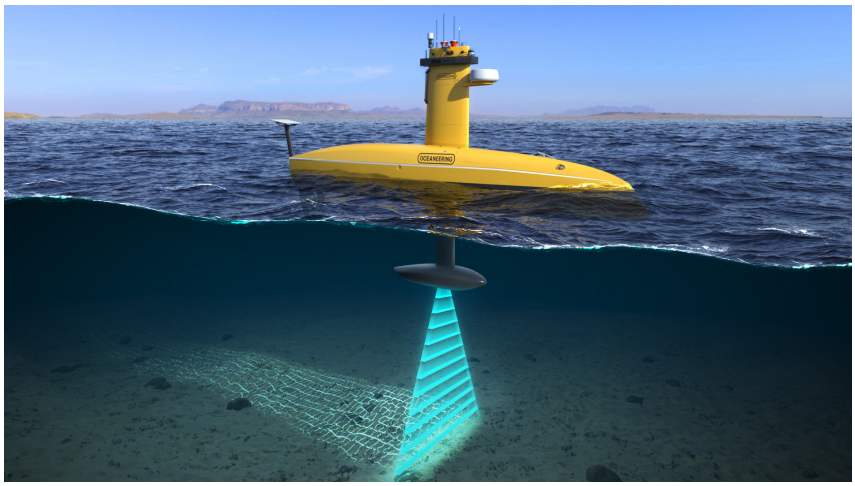

Figure 2.4: USV for remote survey

5. Autonomous Security Patrol USVs. Security patrol USVs are used for monitoring and protecting restricted or sensitive areas such as ports, harbors, and offshore installations [4, 8]. Equipped with cameras, radar, and communication systems, these USVs can autonomously patrol designated areas and alert authorities to any suspicious activities. Some are also armed to respond to potential threats, providing a level of deterrence and security.

Figure 2.5: Autonomous Guard USV

2.2.4 Information presented to remote operators

Operating a remotely controlled unmanned vessel requires the remote operator to have access to a wide array of real-time data to ensure safe and efficient navigation. Since there is no crew on board to monitor systems directly, all critical information must be

communicated through a control interface that replicates the bridge environment of a manned vessel. Below are the key types of data presented to remote operators and how they are processed and displayed.

1. Navigation Data.
   One of the most critical types of information provided to a remote operator is navigation data. This includes:

   - GPS (Global Positioning System): Real-time positioning data allows the operator to track the vessel's current location and plan routes. This data is displayed on an electronic map, giving the operator a clear view of the ship's position relative to other vessels, obstacles, and landmasses [9].
   - Radar: Radar provides information about nearby ships, land, and other obstacles that may pose a risk to the vessel [10]. The radar system detects objects at various distances and displays them on a radar screen, alerting the operator to potential hazards.
   - LiDAR (Light Detection and Ranging): LiDAR is often used for more precise detection of objects, particularly underwater obstacles or during docking maneuvers [9]. The data from LiDAR is processed and displayed as a 3D map, helping operators understand the vessel's surroundings.
   - AIS (Automatic Identification System): AIS data is used to track other ships in the vicinity. This system provides information such as the ship's name, course, speed, and direction, allowing the remote operator to make informed navigation decisions.

2. Vessel Health and Performance Data.
   The remote operator is also responsible for monitoring the health and performance of the vessel. This data is typically displayed on control panels in the form of gauges, meters, or graphical displays:

   - Engine Status and Propulsion Data: Real-time feedback on the engine's RPM (Revolutions Per Minute), fuel consumption, and propulsion power is essential for ensuring the vessel operates efficiently [11]. Operators are alerted to any engine issues, and they can remotely adjust propulsion settings as necessary.
   - Electrical and Power Systems: Data on the vessel's power supply, battery levels (if applicable), and energy usage is also critical. The operator can monitor energy consumption and ensure that backup power systems are available if needed.

- Weather and Environmental Conditions: Information on wind speed, wave height, and weather forecasts is provided to the operator to optimize navigation routes and ensure safe operations in varying environmental conditions.

3. Control and Communication Systems.
   The remote operator has access to various systems that allow them to control the ship and communicate with external entities:

   - Autopilot System: The autopilot system allows the operator to set a course for the ship, which it can follow autonomously [12]. The operator can monitor the ship's adherence to the set course and make adjustments as needed.
   - Thrusters and Maneuvering Systems: Real-time control over the vessel's thrusters and other maneuvering systems is provided through the interface. This allows the operator to make precise movements, especially during docking or when navigating tight spaces.
   - Communication Systems: Operators must maintain communication with maritime authorities, port operators, and other vessels. Communication systems provide voice and data transmission capabilities, ensuring that operators can comply with maritime traffic control and receive instructions from external entities [12].

4. Safety and Security Monitoring.
   Remote operators are also responsible for the vessel's security and safety:

   - Camera Feeds: High-definition cameras placed around the ship provide the operator with a visual feed of the ship's surroundings. This helps in assessing situations that may not be fully captured by radar or LiDAR, such as nearby vessels, docking operations, or potential threats.
   - Fire Detection and Alarm Systems: In the event of a fire or other onboard emergency, the operator is immediately alerted [13]. Systems such as fire alarms, temperature sensors, and flood detectors are constantly monitored to ensure quick responses to any onboard issues.

Data processing and display

All this data is collected from various sensors and systems onboard the vessel and transmitted to the operator via secure communication channels [10]. Data is processed in real-time by the vessel's onboard computers and control systems, which convert raw sensor data into meaningful information, such as graphical maps, alerts, and system status indicators.

The data is then displayed on a control interface, which typically mimics the layout of a traditional ship's bridge. Operators interact with the system through user-friendly graphical interfaces, where they can click on various icons, monitor dashboards, and adjust settings. Alerts and critical warnings are usually highlighted to draw the operator's immediate attention, ensuring swift responses to any potential issues.

Figure 2.6: Inshore control room of an USV

In this thesis, the majority of the data presented are simulated with accuracy so as a realistic control room is presented to the user. Each of the electronic systems will be later presented with precision.

2.3 How ships float and move through the water

In this thesis, we focused on the physics of ships, and thus the physics of unmanned vessels. Ships are massive structures, but they manage to stay afloat and navigate through the water due to fundamental principles of physics and the interaction of forces. At the core of this process are two primary concepts: buoyancy and propulsion. These forces ensure that a ship can float, remain stable, and move through the water effectively.

Buoyancy: Why Ships Float

The principle of buoyancy is what allows ships to float, and it is based on Archimedes' principle, which states that any object submerged in a fluid experiences an upward force equal to the weight of the fluid it displaces. When a ship is placed in water, it pushes or displaces a certain amount of water [14]. If the weight of the displaced water is equal to or greater than the weight of the ship, the vessel will float.

Ships are designed with hulls that displace large volumes of water relative to their weight. Even though ships are heavy, their size and shape ensure they displace enough water to create a buoyant force that keeps them afloat. The hollow structure of a ship also contributes to its ability to displace more water than its actual mass, which is why even large, heavy ships like tankers or cargo vessels can float.

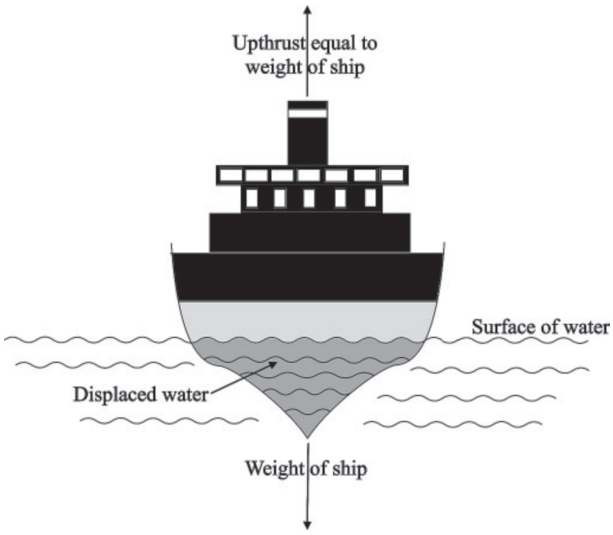

Figure 2.7: Vertical forces acting on a ship

Forces acting on a ship

- Buoyancy Force: This is the upward force exerted by the water on the ship, which counteracts the force of gravity pulling the ship down [14]. As long as the buoyant force is equal to or greater than the ship's weight, the vessel remains afloat. The design of the hull plays a crucial role in optimizing buoyancy by maximizing the volume of displaced water.
- Gravity: The weight of the ship (or the gravitational force acting on the ship) pulls it down towards the center of the Earth. This force works against buoyancy, and the balance between the ship's weight and the buoyant force determines whether the ship floats or sinks. For a ship to float stably, its center of gravity needs to be balanced with the center of buoyancy.
- Resistance (Drag): Ships face different types of resistance as they move through water. Resistance is the force that opposes a ship's motion and consists of three main types [15]:
  - Frictional Resistance: This occurs due to the friction between the ship's hull and the water. As the hull moves through the water, it drags along a thin layer of water molecules, creating resistance. The smoother and more streamlined the hull, the less frictional resistance the ship will experience.

  - Wave-Making Resistance (Residual Resistance): As the ship moves through water, it displaces it and generates waves. The energy required to create these waves leads to resistance. The shape of the hull and the speed of the ship play a significant role in determining the extent of this wave-making resistance.
  - Air Resistance: Although smaller compared to water resistance, air resistance is also a factor, particularly for ships with large structures above the waterline. Air flowing against the ship's superstructure creates drag, which can slightly slow the ship, especially in high winds or at high speeds.

- Propulsion: Ships are equipped with propulsion systems that provide the forward force needed to overcome drag and move through the water. Most modern ships use engines that power propellers, which rotate and push water backward, propelling the ship forward through Newton's third law of motion (for every action, there is an equal and opposite reaction) [16]. Other propulsion methods include jet propulsion and sails in some cases. The efficiency of the propulsion system is key to a ship's speed and fuel consumption.

- Lift and Side Forces (for Stability and Turning): Ships often use side forces, generated by the shape of the hull and rudders, to maintain stability and steer [16]. When turning, a ship utilizes rudder deflection to change the direction of water flow, creating a sideways force that turns the vessel. These forces work together with propulsion to navigate through the water.

In this thesis, all those basic forces were simulated with aiming to create a product application that has a physical essence. Some simplifications were made will be analyzed later. The result of the movement of our ship, Maersk Honam is the result of those forces and how much they impact the rotation, the speed and the orientation of our vessel.

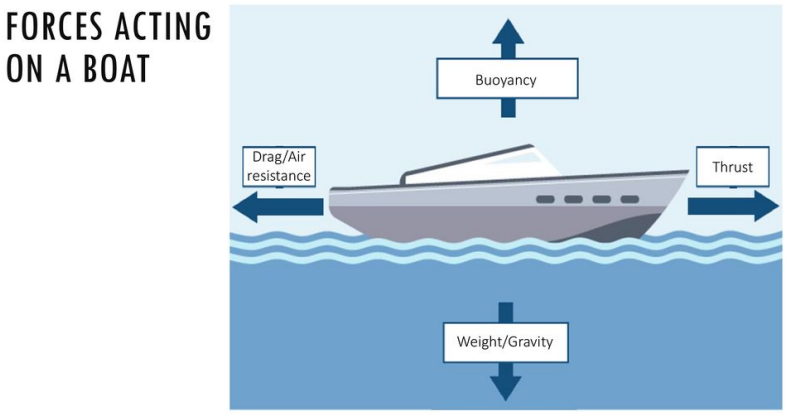

Figure 2.8: Forces acting on a ship

2.4 Virtual Reality

2.4.1 History of VR

1960s – The Early Concepts

The concept of Virtual Reality (VR) began in the 1960s with devices like Morton Heilig's Sensorama (1962), which was one of the first to attempt an immersive sensory experience [17]. Around the same time, Ivan Sutherland developed the Sword of Damocles (1968), the first head-mounted display (HMD) offering basic 3D computer graphics. These early systems laid the groundwork for VR, but their bulky designs and limited technology constrained their widespread use.

Figure 2.9: Sensorama VR

1970s – Early Military and Medical Uses

In the 1970s, VR started to find its way into military and medical fields [17]. The U.S. military developed early VR flight simulators, enhancing pilot training. Around the same period, medical professionals began to explore VR for training doctors in surgeries and medical procedures. These applications showed the first signs of VR's potential for training and education. The Aspen Movie Map at MIT in 1978 was one of the first interactive virtual tours, which further demonstrated VR's potential for immersive experiences.

1980s – Birth of Modern VR

In the 1980s, the modern era of VR began when Jaron Lanier coined the term "virtual reality" and founded VPL Research, which developed key technologies such as the DataGlove and the EyePhone HMD [18]. These devices allowed users to interact more naturally with virtual environments and were critical in advancing VR as a medium for research, entertainment, and education. Around the same time, flight and combat simulators using VR were being refined for military training.

1990s – Commercial Exploration and Challenges

The 1990s saw attempts to bring VR into the commercial market. Sega and Nintendo developed VR systems like Sega VR and Nintendo's Virtual Boy, although these devices failed commercially due to technical limitations, such as low resolution and uncomfortable design [18]. Despite these setbacks, industries like automotive design and healthcare continued to explore VR for specialized training and simulations, where precision and immersive environments were critical.

Figure 2.10: Nintendo Vitrual Boy

2000s – Technological Refinement and Mobile VR

With the dawn of the 2000s, improvements in processing power and display technologies led to significant advancements in VR. The rise of Oculus Rift in 2012, which was funded through a Kickstarter campaign, marked a revival of VR in the public consciousness .

The development of better screen resolution, more powerful computing, and the integration of motion tracking systems allowed VR to provide more immersive and interactive experiences [17]. Mobile VR, powered by smartphones, also became popular, making VR more accessible to the public.

2010s – The VR Boom

The 2010s marked a turning point in the widespread use of VR across multiple industries. With the release of consumer-grade VR systems like Oculus Rift, HTC Vive, and PlayStation VR, VR gained traction not only in gaming but also in fields such as architecture, healthcare, education, and industrial training. These systems provided high-resolution displays, advanced tracking systems, and greater levels of immersion. As a result, VR started being adopted in areas that required immersive, interactive environments for learning, design, and entertainment.

Present and Future

Today, VR is integral to sectors such as aerospace, maritime, education, and healthcare. Technologies like haptic feedback, eye-tracking, and full-body motion capture are continuously being integrated into VR systems, making the experiences even more immersive and precise [17]. As VR hardware and software continue to evolve, it is expected that virtual environments will become more realistic, paving the way for broader adoption in areas like remote work, social interaction, and scientific research. VR is increasingly seen as a transformative technology with applications that extend far beyond entertainment.

2.4.2 Virtual Reality applications

Virtual Reality (VR) has grown from its early stages as a visualization tool into a versatile medium that supports immersive and interactive experiences across a wide range of fields [19]. Today, VR enables users to engage with virtual environments in real-time, offering a dynamic platform for interaction, learning, and simulation. Interactive VR allows participants to directly influence their virtual surroundings, making them active creators of their experiences. This engagement is powered by a variety of input technologies, including hand controllers, motion sensors, and gesture recognition systems, which allow users to manipulate objects, navigate environments, and interact with virtual elements intuitively [20].

VR has found extensive applications in domains such as entertainment, education, training, healthcare, and scientific research. In gaming, VR provides an unprecedented level of immersion, enabling players to step into virtual worlds and interact in ways previously unimaginable [21]. Beyond entertainment, educational institutions are leveraging

VR to create immersive learning environments where students can explore complex subjects such as history, science, and geography in a hands-on manner.

In training simulations, VR allows professionals to practice real-world tasks in a risk-free virtual environment, which has proven invaluable for industries such as aviation, military, and healthcare. These realistic simulations facilitate skill development and knowledge transfer by replicating scenarios that would otherwise be costly or dangerous to practice in real life [19].

VR is also showing great potential in therapeutic applications, particularly in the treatment of psychological disorders such as phobias, PTSD, and anxiety. Using VR, therapists can safely expose patients to controlled environments, helping them confront fears and develop coping strategies [20]. In the medical field, VR is increasingly used in surgical training, allowing surgeons to practice complex procedures in a simulated environment, and for patient education, where immersive tools help patients better understand their medical conditions and treatments [21].

Moreover, VR is becoming an essential tool in scientific research and data visualization, where researchers can explore and analyze complex phenomena within interactive virtual environments. This immersive approach enables scientists to visualize intricate datasets in ways that are often not possible using traditional methods.

In summary, Virtual Reality has become a transformative platform, empowering users to actively participate in a wide array of fields. As VR technology advances and becomes more accessible, the potential for innovation and impact continues to grow, promising exciting future developments across entertainment, education, training, therapy, and scientific research.

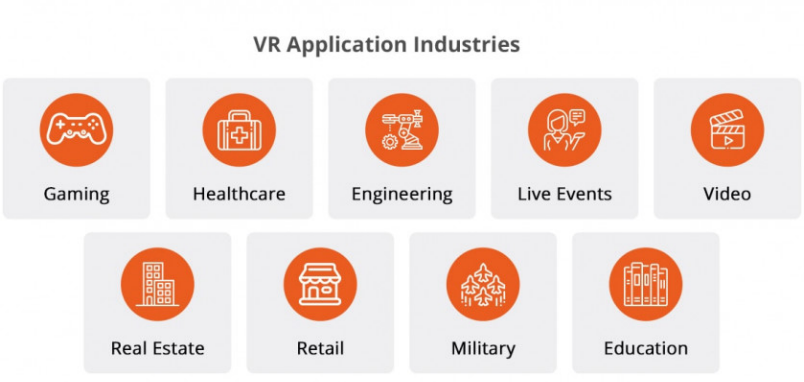

Figure 2.11: VR Applications Sectors

2.4.3 Head Mounted Displays (HMDs)

Head-mounted displays (HMDs) are wearable devices that place users directly into virtual environments by displaying stereoscopic images to each eye [22]. Typically, HMDs

are comprised of display screens, lenses, and motion sensors embedded within a head-worn unit, secured by straps or frames to ensure stability and comfort. The design of HMDs ranges from lightweight models for casual users to advanced systems intended for professional and industrial applications [23].

HMDs serve as the central interface between users and virtual reality (VR), creating an immersive experience by delivering visual and auditory feedback that mimics real-world sensations. This combination of visual, auditory, and motion tracking capabilities enables users to feel fully immersed in the virtual environment [22, 23]. The quality of these stimuli—particularly in terms of resolution, field of view, and sound—determines how realistic and engaging the virtual experience is for the user.

Over the years, technological advancements have greatly improved HMDs, making them more comfortable, lighter, and capable of rendering high-resolution images with minimal latency [24]. Improved motion sensors and inertial measurement units (IMUs) allow for more precise tracking of head movements, while advancements in optics and lens design provide clearer visuals with a wider field of view [22, 24]. Modern HMDs also often feature spatial audio technology, which delivers immersive 3D sound to complement the visuals, enhancing the sense of presence in the virtual environment.

Figure 2.12: HTC Vive Pro

HMDs come equipped with various ergonomic features, such as adjustable straps and padding, to maximize user comfort during extended VR sessions [23]. This design consideration, along with high-fidelity visual displays and accurate motion tracking, contributes to the overall immersive experience, making HMDs ideal for applications in entertainment, education, training, and medical simulations.

For this thesis, a state-of-the-art HMD with ultra-high-resolution displays and a wide field of view was utilized to provide an immersive experience for the unmanned vessel simulator. This advanced hardware allowed for a more engaging and realistic interaction within the virtual control environment.

Figure 2.13: Meta Quest Pro

2.4.4 Position Tracking

Position tracking is a critical component of virtual reality (VR) technology, allowing for the accurate detection of a user's physical movements within a virtual space. By tracking both position and orientation, VR systems enable users to interact with virtual environments in a natural and intuitive way. Several techniques have been developed to ensure precise tracking, each with its own advantages and limitations.

Inertial Measurement Units (IMUs): One common technique for position tracking is the use of IMUs, which incorporate sensors such as accelerometers and gyroscopes. These sensors monitor changes in speed, acceleration, and rotational motion, enabling the system to track the user's head and body movements in real-time. IMUs provide low-latency tracking, which is ideal for creating a smooth and responsive VR experience [25]. However, IMU-based tracking systems can experience drift, meaning small inaccuracies accumulate over time, which may affect tracking precision during extended use or rapid movements.

External Tracking Systems: Another approach involves the use of external sensors or cameras placed around the user's environment to track the position of VR devices like headsets or controllers. These systems use visual markers or cues to accurately determine the user's position within a designated space. Popular examples of external tracking

systems include the HTC Vive's Lighthouse and Oculus Constellation, which use infrared sensors and lasers to ensure precise tracking within a large play area.

Hybrid Tracking Solutions: To mitigate the limitations of individual tracking systems, many VR platforms now utilize hybrid tracking solutions. These systems combine multiple tracking technologies, such as IMUs and external sensors, to enhance both precision and responsiveness. By integrating different tracking modalities, hybrid systems balance the benefits of low-latency IMU tracking with the positional accuracy of external sensors, resulting in a more immersive and reliable VR experience [26].

In this thesis, we utilized the Meta Quest Pro, a state-of-the-art VR headset equipped with advanced position tracking capabilities. The Meta Quest Pro uses inside-out tracking, which leverages built-in cameras and sensors to track the user's position and movements without the need for external hardware. This allows for seamless interaction within the virtual control environment, providing the precision required for accurate unmanned vessel simulations.

2.4.5 Head Tracking

Head tracking is a vital feature in VR technology, enabling systems to track the user's head movements in real time. This allows the virtual environment to respond dynamically to changes in the user's head position and orientation, enhancing the feeling of immersion and presence in the virtual world. The system continuously adjusts the user's viewpoint, ensuring the visual experience aligns with their movements.

Several technologies can be used for head tracking. Inertial Measurement Units (IMUs), composed of accelerometers and gyroscopes, track head rotation with low latency, making them effective for capturing real-time movement. However, IMUs may suffer from accuracy issues like drift over time. More precise tracking can be achieved using optical tracking systems, such as those in the Oculus Rift or HTC Vive, which employ external cameras or sensors to monitor head position and orientation within a defined space [27].

Head tracking systems typically offer either 3 degrees of freedom (3DoF), capturing only rotational movements, or 6 degrees of freedom (6DoF), which also track the user's physical movements along the X, Y, and Z axes. This additional freedom enhances the user experience by allowing physical movement within the virtual environment.

Challenges such as latency, jitter, and occlusion can affect the quality of head tracking. Hybrid systems that combine IMUs and external sensors help mitigate these issues, providing smoother and more precise tracking [28].

In this thesis, the Meta Quest Pro was used, which features inside-out tracking. This hybrid system integrates IMUs and onboard cameras to provide 6DoF, allowing for accurate head tracking without external sensors, enhancing the immersive experience in the

unmanned vessel simulator.

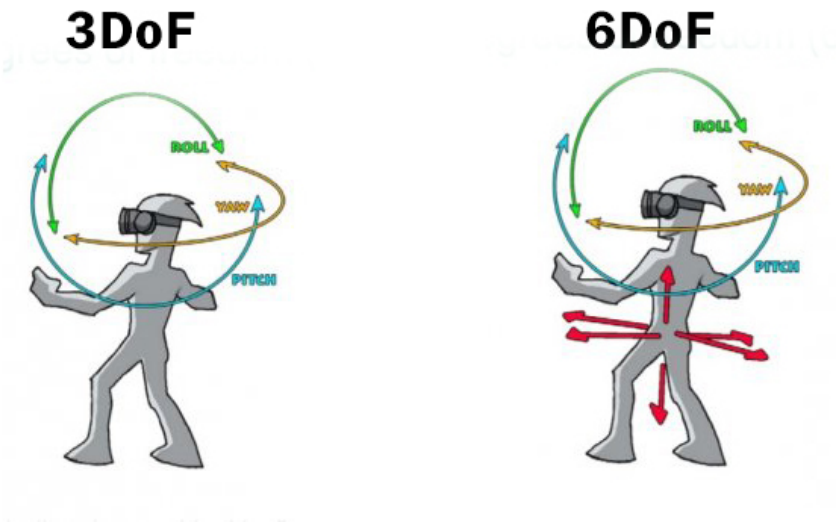

Figure 2.14: Degrees Of Freedom

2.4.6 Input Methods

Input methods are essential for enabling interaction within virtual reality (VR) environments, allowing users to engage naturally and intuitively with virtual objects and surroundings. These methods include hand tracking, controllers, and eye tracking, each offering different levels of immersion and control in VR experiences.

Hand Tracking Technology: Hand tracking allows VR systems to detect and analyze hand movements in real-time, enabling users to interact with virtual objects using their hands. This technology often uses optical sensors and depth cameras to capture hand and finger movements, while machine learning algorithms interpret these movements to provide smooth and precise control. Advanced hand tracking systems recognize intricate gestures, enabling users to perform complex interactions, such as grabbing or manipulating virtual objects without the need for physical controllers [29].

Controller-Based Input: VR controllers, commonly equipped with buttons, triggers, and joysticks, offer users tactile feedback, enhancing the interaction experience. These controllers provide precise control over movements and actions in virtual environments. With ergonomic designs and haptic feedback, they allow users to grab objects, navigate menus, and interact with interfaces in a way that feels natural and engaging. The tactile feedback helps users feel more connected to the virtual world, enhancing immersion [30].

Eye Tracking Integration: Eye tracking technology monitors the user's gaze and eye movements, allowing the VR system to respond dynamically based on where the user is looking. By tracking the user's visual focus, VR applications can adjust interactions

Figure 2.15: Hand Tracking Visualisation

and provide more natural responses. Eye tracking also enables advanced features like foveated rendering, where processing power is concentrated on areas the user is looking at, optimizing system performance while maintaining high visual quality in focused areas.

Hybrid Input Solutions: Some VR systems combine hand tracking, controller input, and eye tracking into hybrid input methods, offering flexibility based on the user's needs or the type of task. These hybrid systems allow users to seamlessly switch between different input methods, enhancing versatility and user comfort in various VR applications. By combining the precision of controllers, the intuitiveness of hand gestures, and the responsiveness of eye tracking, hybrid solutions provide a more immersive and dynamic user experience [30].

In this thesis, the Meta Quest Pro was utilized, which integrates both hand tracking and controller-based inputs. This hybrid input system allowed users to control the unmanned vessel simulator intuitively, ensuring that interactions within the virtual control room were both precise and immersive.

2.5 Virtual Reality Simulators

Over the years, virtual reality (VR) simulators have become widely used across industries, offering immersive and interactive environments for training, education, and various professional applications.

NASA's Virtual Reality Lab (VRL) has been a pioneer in using VR for astronaut training [31]. Their VR simulator replicates the International Space Station (ISS) environment, allowing astronauts to practice spacewalks and repairs in a virtual setting that mimics the challenges of space without the associated risks. This has been instrumental in preparing astronauts for real missions.

Flight Simulators, employed by organizations such as Lockheed Martin and Boeing, provide highly detailed virtual cockpits for both military and commercial pilot training. These simulators recreate different flight scenarios, including emergencies and complex

navigation, making pilot training more cost-effective and reducing the need for extensive real-world flight time.

Surgical VR Simulators, such as Osso VR, allow medical professionals to practice surgeries in a virtual environment [32]. These simulators are designed to replicate detailed anatomy and surgical procedures, providing a safe, repeatable training ground for surgeons to refine their skills without the risks of operating on live patients.

Military Training Simulators, such as the Dismounted Soldier Training System (DSTS), offer soldiers the chance to practice battlefield scenarios in a fully immersive virtual environment. These simulators allow soldiers to experience realistic combat settings, improving their decision-making and teamwork skills in life-like scenarios.

Driving and Racing Simulators, used by F1 racing teams and driver training programs, allow participants to practice navigating race tracks and handling vehicles in a range of conditions, from wet surfaces to high-speed corners. These simulators enhance both performance and safety in racing and general driving.

Industrial Simulators, like those from Immersive Technologies, train workers to operate heavy machinery, such as cranes and excavators. By simulating high-risk job scenarios, these VR systems reduce the chance of accidents and allow operators to become familiar with complex machinery in a controlled environment.

Firefighter Training Simulators, such as the FLAIM Trainer, use VR to simulate dangerous fire scenarios, allowing trainees to practice firefighting techniques in a risk-free environment. These simulations help prepare firefighters for high-stress situations without the dangers of live fire training.

Figure 2.16: Osso Surgical VR Simulator

2.5.1 Virtual Reality Unmanned/Ship Simulators

Important Ship Simulators

- Transas Marine Simulators: Transas offers one of the most widely used maritime training simulators. It provides virtual environments for crew to practice ship handling, navigation, and emergency procedures. The simulator replicates real-world conditions such as ocean currents, weather, and vessel traffic, ensuring operators are well-prepared for diverse situations at sea.

  Purpose: Training deck officers, engine room crews, and emergency responders.

- Kongsberg Maritime Simulators: Kongsberg is another industry leader, offering advanced simulators that cover ship navigation, engine room training, and even dynamic positioning operations. The high-fidelity visual systems and real-world physics simulations offer users a near-realistic experience.

  Purpose: Navigation, dynamic positioning, and crisis management.

- Panama Canal Authority's Simulator: The Panama Canal Authority has developed simulators for training personnel on the specific challenges of navigating the canal. It enables users to practice operating different types of ships through the Panama Canal's unique lock system and busy maritime traffic.

  Purpose: Training pilots and tugboat captains for navigating the Panama Canal.

Unmanned Vessel Simulators

- ST Engineering's USV Simulator: This simulator is used for training operators of Unmanned Surface Vessels (USVs). It offers detailed virtual environments and complex mission scenarios, allowing trainees to control the vessel remotely. It simulates conditions such as weather, tides, and onboard system malfunctions, improving remote operational readiness.

  Purpose: Training operators to control and navigate unmanned vessels remotely.

- Dynautics Autonomous Vessel Simulator: Dynautics offers a simulator designed for autonomous and unmanned surface vessels. It provides hydrodynamic models, environmental conditions, and control systems for testing vessel behavior under

Figure 2.17: Kongsberg Maritime Simulator

various conditions. The simulator helps developers design and test autonomous vessels, reducing real-world trial costs and improving safety.

Purpose: Design, testing, and training for autonomous vessels.

- Maritime Simulation Institute (MSI): The MSI offers high-level simulations for both manned and unmanned vessels. Their simulators are used for NOAA officer training, integrating control systems for autonomous platforms like unmanned research vessels, which have grown in use for oceanographic surveys and naval applications.

  Purpose: Manned and unmanned vessel training with specific use cases in maritime research and navigation.

Figure 2.18: A simulator for testing and assessing human supervised autonomous ship navigation

2.5.2 VR-Based Training and Its Impact on Maritime Safety and Efficiency

The adoption of VR-based ship simulators has reshaped how maritime operators are trained by offering a controlled, immersive, and risk-free environment. Safety training through VR has become a key factor in preventing real-world accidents by simulating complex, high-risk scenarios such as fires, engine failures, or harsh weather conditions, which would be unsafe to replicate physically. In these virtual simulations, operators can make mistakes and learn from them without the consequences of real-world incidents. By providing this "fail-safe" environment, VR equips operators with vital experience in navigating emergencies, ultimately reducing the risk of human error and ensuring quicker, more confident responses during actual crises.

Another significant benefit of VR-based training is its contribution to environmental sustainability. Traditional training often involves physical ship trials, which consume large amounts of fuel, produce emissions, and cause environmental wear and tear on equipment. In contrast, virtual simulations eliminate the need for such trials, drastically reducing the environmental footprint of training programs. Trainees can practice ship handling, maneuvering, and crisis management in detailed, lifelike environments without the carbon emissions or fuel costs associated with operating actual vessels. This transition to digital training not only conserves resources but aligns with the global maritime industry's goal to reduce its environmental impact.

One of the most crucial aspects of VR-based training is its ability to prepare operators for crisis management. Emergencies such as engine failure, navigation errors, or extreme weather are simulated to allow operators to practice responses under stressful conditions.

In these scenarios, users must think quickly and apply correct procedures, giving them the practical experience they might not gain through standard training methods. The VR simulator can present a range of realistic challenges, such as steering the ship through a storm, responding to mechanical malfunctions, or avoiding collisions. These high-pressure situations are difficult to replicate in physical settings, but VR offers a safe space for operators to practice them repeatedly, honing their ability to react swiftly and correctly during actual maritime crises.

Moreover, VR simulations can adapt dynamically to user performance, offering real-time feedback. If an operator fails to respond adequately to a simulated crisis, the system can guide them through corrective steps, reinforcing their learning and building their problem-solving capabilities. This type of real-time evaluation is crucial for ensuring that trainees not only understand the theoretical aspects of ship operations but also have practical, hands-on experience managing difficult situations.

In the long run, the advantages of VR-based training extend beyond immediate operational readiness. It contributes to the long-term safety culture within maritime companies by providing continuous learning opportunities. Operators can engage in regular refresher training without the logistical challenges of arranging physical ship trials or risking safety in a live environment. In turn, this fosters a higher level of preparedness and reduces the likelihood of accidents due to outdated knowledge or procedural lapses.

Overall, VR-based ship simulators represent a significant advancement in the maritime training landscape. By offering safe, environmentally friendly, and crisis-oriented training, these simulators prepare operators not just to perform everyday tasks but to respond effectively in the most challenging situations. As VR technology continues to evolve, its role in shaping competent, well-prepared maritime professionals is expected to grow, setting new standards for both safety and efficiency in the industry.

2.6 Game Engines

Game engines are specialized software frameworks designed to streamline the development process by offering developers a set of tools, libraries, and features for building interactive digital experiences. These engines simplify tasks such as rendering graphics, simulating physics, processing inputs, and more, enabling developers to focus on creating engaging and immersive content without needing to build these functionalities from scratch.

The concept of game engines originated in the early days of video game development in the 1970s and 1980s. During this time, game developers wrote all the code manually, including graphics, physics, and input processing. As games grew in complexity, developers realized the need for more efficient tools to handle these increasingly demanding tasks. This shift laid the foundation for what would become modern game engines.

In the 1990s, the rise of 3D graphics and the growing popularity of personal computers

and gaming consoles gave momentum to the evolution of game engines. Major gaming companies like id Software, Epic Games, and Valve Corporation developed proprietary engines for groundbreaking games such as Doom, Quake, and Half-Life [33]. These engines allowed developers to reuse game code, shortening development time while still producing high-quality, immersive games.

By the late 1990s and early 2000s, game engines became more widely available to third-party developers through licensed platforms such as Unreal Engine and CryEngine [34]. These engines provided comprehensive toolsets, robust graphics rendering, and advanced scripting capabilities, making it easier for developers to create high-quality games without starting from scratch.

Today, game engines have expanded far beyond traditional video games. They are now used in various industries, including virtual reality (VR), augmented reality (AR), simulations, and multimedia applications [35]. Modern engines feature real-time rendering, physics simulation, artificial intelligence, networking, and cross-platform support, making them indispensable tools for creating interactive content in industries like film, architecture, education, and healthcare.

In summary, game engines have evolved into powerful and versatile platforms that fuel a wide range of digital experiences. Their history reflects the rapid advancements in technology and the growing demand for immersive, interactive content across multiple platforms and industries.

2.6.1 Unity

Unity is recognized for its exceptional versatility, offering developers the capability to create applications that seamlessly transition across 2D, 3D, and virtual reality (VR) platforms [36]. This adaptability allows projects to evolve from simple 2D apps to more complex 3D environments or immersive VR experiences without requiring significant re-development. Such flexibility is crucial, enabling developers to experiment with different interaction paradigms while maintaining consistency within the same development environment [37]. Unity's powerful tools enable developers to use a unified codebase and asset pipeline, ensuring smooth transitions and cohesive app development for various platforms.

One of Unity's standout qualities is its user-friendly design, making it accessible to developers of all skill levels. With an intuitive interface and visual editor, Unity allows creators to quickly prototype and develop their ideas without needing extensive coding knowledge [36]. Its drag-and-drop functionality and robust scripting capabilities streamline the development process, allowing developers to focus on crafting engaging user experiences. Unity's large community and comprehensive documentation further support developers in overcoming challenges and optimizing workflows, making it a favorite among both beginners and experts.

In addition to its ease of use, Unity boasts a wide range of advanced features and customization options. These include high-quality graphics and rendering capabilities, realistic physics simulations, sophisticated audio effects, and robust animation tools [37]. Unity's extensible architecture also supports third-party plugins and assets, providing developers the flexibility to expand their projects and integrate additional functionality as needed. This combination of versatility, power, and ease of use has made Unity a go-to platform for developing everything from mobile apps to complex VR simulations [38].

In this thesis, Unity was chosen as the development platform for its robust capabilities, particularly in handling the physics and simulation aspects of the unmanned cargo ship simulator, ensuring an immersive and realistic experience for users.

Chapter 3

Technological Backround and Definitions

3.1 Introduction

In this chapter, the technological framework supporting the thesis will be thoroughly examined. A detailed exploration of the Unity game engine will be presented, highlighting its key features, including those used for VR integration in this project. Additionally, the chapter will delve into the physics principles applied, with a focus on the buoyancy equation and the interaction of resistances and propulsion.

3.2 Unity Structure and Architecture

Unity, as a game engine and development platform, provides a comprehensive toolkit for creating interactive 3D environments, including virtual reality (VR) applications. Understanding Unity's fundamental structure is essential for effectively navigating its development environment and building robust projects. In Unity, every creation starts with a "Project," which serves as the container for all the necessary elements used in development [39].

A Project is composed of various Assets, such as 3D models, textures, sounds, and scripts. Within a Project, there are one or more Scenes, which act as the individual stages or environments where the interaction takes place. Each Scene consists of GameObjects, which represent all the objects in the scene—whether they are characters, cameras, or environmental elements. Prefabs, a critical aspect of Unity's structure, are reusable templates that allow developers to replicate objects with consistent properties across different scenes.

Both GameObjects and Prefabs are extended with Components, which are the building blocks that define their behavior, such as physics properties, audio, and rendering. Custom Scripts can also be attached to these objects, providing control over their interactions and behavior within the scene.

By understanding this foundational structure—Projects, Assets, Scenes, GameObjects, Prefabs, Components, and Scripts—developers can efficiently manage complex

projects and create immersive VR applications with ease. The majority of the following subsections are based on the official unity documentation [40].

3.2.1 Unity Supported Pipelines

Unity offers a variety of rendering pipelines, each designed to suit different performance, quality, and platform requirements. The main pipelines include the Built-In Render Pipeline, the Universal Render Pipeline (URP), and the High Definition Render Pipeline (HDRP).

The Built-In Render Pipeline (also known as the Legacy Pipeline) is widely compatible across platforms and supports essential features such as dynamic lighting, shadows, and post-processing effects. It has been the default choice in Unity for years and is suitable for projects requiring general-purpose rendering. It's a robust and adaptable solution but lacks some of the advanced optimization and modern features available in other pipelines.

The Universal Render Pipeline (URP) is a more modern, lightweight option designed to offer better performance, particularly for mobile platforms, VR, and low-end hardware. While it supports many contemporary features like Shader Graph and forward or deferred rendering, it balances quality and performance for a wide range of devices. URP is highly customizable and offers improved rendering quality over the Built-In Pipeline, making it ideal for projects targeting multiple devices with resource constraints.

On the other end of the spectrum, the High Definition Render Pipeline (HDRP) is built for high-end platforms and applications that demand realistic, cutting-edge visuals. It includes advanced features such as physically based rendering (PBR), high-quality lighting, shadows, volumetric effects, and advanced post-processing options. HDRP is tailored for projects requiring high-fidelity visuals, such as AAA games or simulations, where realism and graphical detail are critical. However, its high demands on computing resources often make it unsuitable for lower-end devices or real-time applications on mobile.

In this thesis, the Built-In Render Pipeline was utilized to maintain compatibility across platforms while ensuring manageable performance. Although the Universal Render Pipeline was considered for its efficiency and versatility, it was ultimately dismissed due to the project's focus on higher-quality visuals, which URP could not provide at the desired level. Furthermore, while the High Definition Render Pipeline was initially appealing for its advanced graphical capabilities and visual fidelity, its significant computational demands and potential performance drawbacks led to it being deemed impractical for this particular project. The Built-In Render Pipeline provided the optimal balance between visual quality and performance, ensuring that the project remained stable across various platforms without sacrificing too much graphical detail or system efficiency.

3.2.2 Assets

Assets in Unity encompass all the files and resources used to build a project, such as 3D models, textures, audio clips, animations, scripts, and more. They act as the foundational elements of the virtual environment, shaping the visual, auditory, and interactive aspects of the project. By combining these resources, developers create the immersive experiences that define Unity projects. Assets are managed within the Unity editor and can be easily imported or created directly within the platform, ensuring seamless integration and customization for specific project needs.

3.2.3 Scenes

Scenes in Unity serve as distinct levels or environments within a project. Each scene is an organized collection of objects, characters, and interactive elements that define a specific part of the virtual world. By structuring a project into multiple scenes, developers can focus on designing and developing individual sections or stages independently. This organization also helps in managing complex projects by allowing for a modular approach, where different scenes can be loaded or transitioned between during gameplay. Whether it's a main menu, gameplay level, or cinematic cutscene, each scene in Unity contributes to the overall project by housing the assets and GameObjects that bring the virtual environment to life.

3.2.4 GameObjects

GameObjects form the foundation of Unity's scene hierarchy, serving as the containers for all elements within a virtual environment. They can represent a wide range of objects, including characters, props, lights, cameras, and more. While GameObjects themselves are blank entities, they gain functionality through the attachment of Components. These Components define the object's visual appearance, behavior, physics, and interactions within the scene. By customizing and manipulating these Components, developers can tailor GameObjects to fulfill specific roles in a game or application.

3.2.5 Components

Components in Unity are reusable pieces of functionality that can be attached to GameObjects to define their behavior and properties. They are the building blocks that give GameObjects their functionality, ranging from physics interactions to sound playback. Examples include Colliders, which detect collisions, Rigidbodies for simulating physical interactions, custom Scripts that define specific behaviors, and Audio Sources to manage sound within the scene. Components allow developers to extend the capabilities of

GameObjects without needing to write all behavior from scratch, making development more efficient and modular.

3.2.6 Scripts

Scripts in Unity are written primarily in C sharp and are used to define custom behaviors and logic for GameObjects and other elements within a project. By attaching scripts to GameObjects, developers can access and modify object properties, respond to user input, and interact with other components and systems within Unity's environment. These scripts enable the implementation of complex game mechanics, application logic, and interactive elements. They provide flexibility and power in creating dynamic, responsive, and interactive experiences in Unity projects, making them a crucial tool for development.

3.2.7 Coordinates- Transform

In Unity, the coordinate system forms the foundation for positioning, rotating, and scaling objects in a 3D space. It operates on a Cartesian coordinate grid with three axes: X, Y, and Z. The X-axis typically represents horizontal movement, with positive values extending to the right and negative values to the left. The Y-axis indicates vertical movement, where positive values go upward and negative values downward. The Z-axis controls depth, with positive values extending forward and negative values extending backward. Together, these axes create a right-handed coordinate system, adhering to the right-hand rule for consistent rotation and orientation within the scene.

The Transform component is attached to every object in a Unity scene and governs its spatial properties, including position, rotation, and scale. Through this component, developers can precisely control an object's placement in the 3D environment. The Position values define an object's location relative to the scene's origin (0, 0, 0). The Rotation values specify the object's orientation along the X, Y, and Z axes, determining its angular position. The Scale values adjust the object's size along each axis, either uniformly or non-uniformly, allowing objects to be resized while retaining their proportions or altered for specific effects.

3.2.8 Materials, Textures, Shaders, Lighting

Materials Materials in Unity define the surface properties of objects, such as color, reflectivity, transparency, and glossiness. They determine how light interacts with the surface and are key to giving objects their visual appearance. By using different material properties like metallic or specular maps, developers can simulate effects like roughness or shininess. Unity provides several built-in materials, but custom materials can also be created to meet the specific visual needs of a project.

Textures Textures are 2D images applied to the surfaces of 3D objects to add detailed visual features such as patterns, colors, or depth. They enhance realism by simulating surface characteristics like wood grain, metal rust, or fabric patterns. In Unity, textures are typically combined with materials, providing an extra layer of detail to the object. Textures can also be used for normal mapping, bump mapping, or height maps to create the illusion of depth on flat surfaces.

Shaders Shaders are small programs that define how materials interact with light to create various visual effects. In Unity, shaders are written in shading languages like Cg/HLSL, and they control the rendering behavior of materials, including lighting, shadows, and reflections. Unity supports different shader types, including surface shaders for high-level material effects and vertex/fragment shaders for more complex custom visuals. This allows developers to create advanced graphical effects, such as dynamic lighting or water simulations.

Lighting Lighting in Unity is controlled through various systems and components, such as:

Light Sources: Unity offers several types of light sources, including directional, point, and spotlights, each affecting the scene differently. These lights control how objects are illuminated and cast shadows. Light Probes: These capture environmental lighting and apply it to dynamic objects, ensuring consistency in how light interacts with moving elements. Global Illumination (GI): GI simulates indirect lighting caused by light bouncing off surfaces, adding realism to scenes by replicating natural light behavior. Lightmaps: Precomputed textures that store the lighting information for static objects, helping improve rendering performance while maintaining realistic lighting. Light Cookies: Textures applied to light sources to create customized light patterns, such as simulating shadows from tree leaves or window panes.

3.2.9 User Interface (UI)

In Unity, the User Interface (UI) system provides a comprehensive set of components for creating interactive elements like buttons, sliders, text fields, images, and panels. These components can be arranged and styled to design custom interfaces tailored to the needs of the application.

Unity's built-in UI editor allows developers to create, configure, and position UI elements directly within the Unity Editor, enabling rapid prototyping and iteration. The UI system is highly flexible, supporting interaction through various input methods such as mouse, keyboard, touch, and VR controllers. This makes it adaptable for a wide range of platforms and devices, from traditional PC applications to mobile and virtual reality environments.

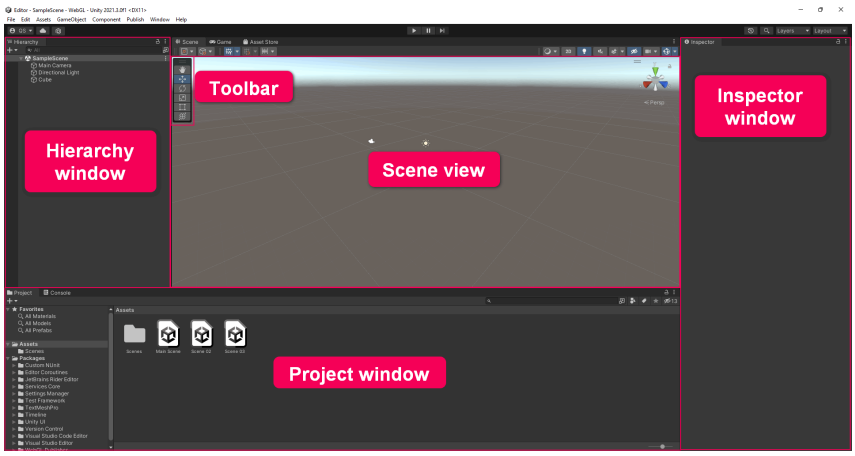

Figure 3.1: Unity Editor

3.2.10 AI and Navigation

Unity provides robust tools for implementing artificial intelligence (AI) and navigation systems within games and simulations. One of the core components for navigation is the NavMesh, a navigational mesh that allows AI agents to move intelligently through a scene by calculating paths around obstacles. Unity's NavMesh system can dynamically adjust for changes in the environment, ensuring that AI can react to real-time events or obstacles in a scene.

AI behaviors in Unity can be customized through NavMesh Agents, which are attached to GameObjects. These agents control the movement and pathfinding of characters or vehicles, helping them navigate complex environments. In addition to basic pathfinding, developers can implement features like dynamic obstacle avoidance, off-mesh links for traversing difficult terrain, and agent coordination for group movement.

By using Unity's AI and Navigation tools, developers can create responsive and adaptive environments where characters or objects can interact realistically with their surroundings, improving the immersion and complexity of gameplay or simulations. This system is highly scalable, making it suitable for various applications, from simple AI movement to complex, multi-agent interactions in real-time environments.

3.2.11 VFX Graph in Unity

Unity's Visual Effect Graph (VFX Graph) is a powerful tool designed for creating high-performance, visually complex effects such as explosions, smoke, fire, and other particle-based simulations. Using a node-based interface, VFX Graph allows developers to design intricate visual effects in real-time without needing to write complex code, offering flexibility and scalability for effects in both 2D and 3D environments.

The VFX Graph is GPU-based, which allows for the processing of millions of particles simultaneously, making it ideal for large-scale simulations or effects that require significant

performance, especially in VR or AR applications. Effects are generated procedurally and can be fully customized with a wide range of parameters, allowing developers to control the appearance, movement, and interaction of particles with the environment.

With its integration into Unity's rendering pipeline, VFX Graph also supports features such as lighting, shadows, and depth interactions, allowing effects to blend seamlessly into the scene for increased realism. This makes it a crucial tool for creating immersive environments, particularly in projects like virtual reality, where dynamic and responsive visual effects are key to enhancing user experience.

3.2.12 Animation System

Unity's Animation System provides a versatile framework for animating objects, characters, and environments. At its core is the Animator Controller, which allows developers to create and manage complex animation states and transitions for characters and objects. The system supports both skeletal animation, commonly used for character rigs, and keyframe animation, which can be applied to any object within the scene.

Unity also features Blend Trees, enabling smooth transitions between animations based on parameters such as speed or direction. This is particularly useful for character movement, where actions like walking, running, or jumping can be blended seamlessly depending on the player's input. For more dynamic control, Unity's animation system supports Inverse Kinematics (IK), which helps characters interact naturally with the environment, such as placing hands on objects or feet on uneven terrain.

In combination with timeline-based tools like Unity's Timeline, the animation system allows for easy sequencing of complex animation events, making it ideal for creating cutscenes, choreographed actions, or complex behavior-driven animations in both games and simulations.

3.3 Unity VR Structure

In Unity, transitioning from a desktop project to a virtual reality (VR) experience requires integrating specialized libraries, tools, and components designed for immersive interactions. Unlike traditional applications, VR projects demand additional elements to handle input from VR devices, manage rendering, and ensure smooth, real-time performance in 3D space. Unity provides a range of VR-specific libraries and plugins, such as the XR Interaction Toolkit, which enables developers to integrate VR headsets, controllers, and spatial interactions seamlessly.

To convert a desktop project into VR, developers must incorporate components tailored to VR environments, including headset tracking, hand controllers, and UI systems optimized for stereoscopic displays. These components manage crucial VR functionalities

such as head and hand movement tracking, physics interactions in virtual environments, and rendering optimizations for high-performance output. Special attention must also be given to maintaining frame rates and avoiding motion sickness, which is critical for a smooth and comfortable VR experience.

In this project, the virtual environment was developed using a combination of VR frameworks, plugins, and rendering techniques to ensure optimal performance and fidelity in the immersive simulation of the Maersk Honam unmanned ship.

3.3.1 XR System

Unity's XR (Cross-Reality) System is a comprehensive framework that supports the development of immersive experiences across a variety of reality platforms, including virtual reality (VR), augmented reality (AR), and mixed reality (MR). This system provides developers with a unified interface for building applications that can easily transition between different reality modes, allowing for seamless adaptability across devices and platforms.

One of the key benefits of Unity's XR System is its ability to abstract hardware complexities, enabling developers to focus on creating engaging content without needing to address the intricacies of platform-specific optimizations. This makes cross-reality development more accessible, streamlining the process of building experiences that can function across multiple devices. However, this convenience can also come with challenges, such as performance overhead and compatibility issues when targeting multiple platforms with varying capabilities and requirements. Despite this, the XR System remains a valuable tool for creating versatile, immersive applications.

3.3.2 XR Origin

The XR Origin is a crucial element in transitioning from desktop to VR environments within Unity. It acts as the anchor or reference frame for all XR interactions, determining the position and orientation of the virtual camera relative to the physical world or play space. In VR, the XR Origin ensures that a user's real-world movements, such as walking, turning, or crouching, are accurately mapped to the virtual environment, creating a seamless and immersive experience.

By aligning the XR Origin with the user's physical location and orientation, developers can minimize motion sickness and enhance the naturalness of movement in VR. This transformation enables more responsive and intuitive interactions, allowing users to feel more connected to the virtual world. The accurate mapping of physical movements to the virtual space is essential for creating a fluid and immersive VR experience, making the XR Origin a key component in any XR project.

3.3.3 XR Interaction ToolKit

Unity's XR Interaction Toolkit is a framework designed to simplify the development of interactive VR experiences. It provides developers with a broad set of tools and components that are essential for implementing a wide range of VR interactions. From basic object manipulation and teleportation to more complex tasks such as menu navigation and initiating animations, the toolkit offers a versatile and user-friendly system.

Developers can integrate features like object interaction, hand gesture recognition, locomotion, and user interface navigation into their VR projects using the prefabs, scripts, and resources provided by the toolkit. This streamlined approach allows for the quick implementation of interactive elements without the need for extensive custom coding, making the XR Interaction Toolkit an invaluable resource for creating immersive and responsive VR environments.

3.3.4 XR Plug-in Management

Unity's XR Plug-in Management system allows developers to efficiently manage and integrate external XR (Extended Reality) plug-ins into their projects. These plug-ins enable Unity applications to support a variety of VR, AR, and MR devices and platforms beyond Unity's default XR framework. The system provides a unified interface for managing plug-ins, simplifying the process of adding support for multiple XR ecosystems.

One of the key advantages of XR Plug-in Management is its flexibility, allowing developers to easily switch between different XR plug-ins depending on the target device or platform. This ensures broader compatibility across a wide range of hardware and software, offering support for platforms like Oculus, SteamVR, or HoloLens. Additionally, the system includes tools for configuring input mappings, optimizing rendering settings, and adjusting performance parameters, enabling developers to deliver an optimal XR experience tailored to the specific needs of various devices.

3.3.5 Open XR

OpenXR, developed by the Khronos Group, is an open standard designed to unify the development of virtual reality (VR) and augmented reality (AR) applications across multiple hardware platforms and software ecosystems. It provides a common API (Application Programming Interface) that allows developers to write XR applications once and deploy them seamlessly on a wide variety of XR devices, including VR headsets, AR glasses, and other immersive technologies.

By adopting OpenXR, developers can avoid the complexities of platform-specific code modifications, enabling broader compatibility across different XR hardware. OpenXR standardizes the handling of key XR features such as tracking, input management, render-

ing, and device interactions, abstracting the differences between devices. This promotes interoperability, making it easier for device manufacturers, platform developers, and content creators to collaborate within the XR ecosystem. The unified framework provided by OpenXR ensures consistent user experiences across diverse devices and platforms, facilitating the growth of a more cohesive and diverse XR environment.

3.4 Unity Physics

Unity's physics system offers a robust suite of tools for simulating realistic physical interactions in 2D and 3D environments, making it essential for game development and simulations. Unity provides both 2D Physics and 3D Physics engines, powered by Box2D for 2D physics and NVIDIA's PhysX engine for 3D physics, which handle collisions, forces, and rigidbody dynamics.

Rigidbodies: Rigidbodies are the core objects that allow physics-based movement. By adding a Rigidbody component to a GameObject, it can respond to gravity, forces, and collisions. Rigidbodies can be either dynamic, static, or kinematic, each affecting how they interact with forces and other objects in the environment.

Colliders: Colliders define the physical boundaries of an object. Unity supports different types of colliders such as Box, Sphere, Capsule, and Mesh Colliders (for complex shapes). These colliders allow objects to interact with each other by detecting collisions and triggering events.

Joints: Unity supports various joint types like Fixed, Hinge, Spring, and Configurable Joints, which help connect objects together and simulate physical constraints such as swinging doors, springs, or connected bodies.

Raycasting: Raycasting is used to detect objects along a specified direction, often used for line-of-sight checks, shooting mechanics, or surface detection. Raycasting can be done in both 2D and 3D, making it a versatile tool for various gameplay mechanics.

Gravity and Forces: Unity provides control over gravity and allows the application of forces like Impulse, Velocity, and Acceleration to simulate real-world physics behavior. This is particularly useful for scenarios such as launching objects, simulating wind, or adding friction and drag.

Collision Detection: Unity supports different types of collision detection, such as discrete and continuous collision detection. This is critical for handling fast-moving objects and ensuring that collisions are accurately detected, which is particularly important in VR or high-speed environments.

Character Controllers: For player movement, Unity provides a Character Controller component that offers built-in collision and movement handling. This is useful for games where the physics system needs to handle player movement in complex environments.

Cloth Physics and Soft Bodies: Unity includes a cloth physics system, allowing the simulation of realistic fabrics or soft bodies. This is often used for character clothing, flags, or other dynamic elements that react naturally to movement and environmental forces.

Ragdoll Physics: Ragdoll physics can be applied to characters for realistic, physics-based movement after a character loses control or during death animations. Ragdolls consist of multiple colliders connected by joints that move naturally when influenced by external forces.

Trigger Events: Unity offers the ability to create trigger zones, which can detect when objects enter or exit a specific area. This feature is useful for interactions, such as picking up items, triggering animations, or initiating gameplay mechanics when an object enters a defined space.

3.5 Buoyancy

As mentioned in the previous chapter, buoyancy is the upward force exerted by a fluid that opposes the weight of an object immersed in it. This principle is governed by Archimedes' Principle, which states:

> "An object submerged in a fluid experiences an upward buoyant force equal to the weight of the fluid displaced by the object."

This force determines whether an object floats, sinks, or remains neutrally buoyant. The buoyant force F_b can be expressed mathematically as we see in [41] as:

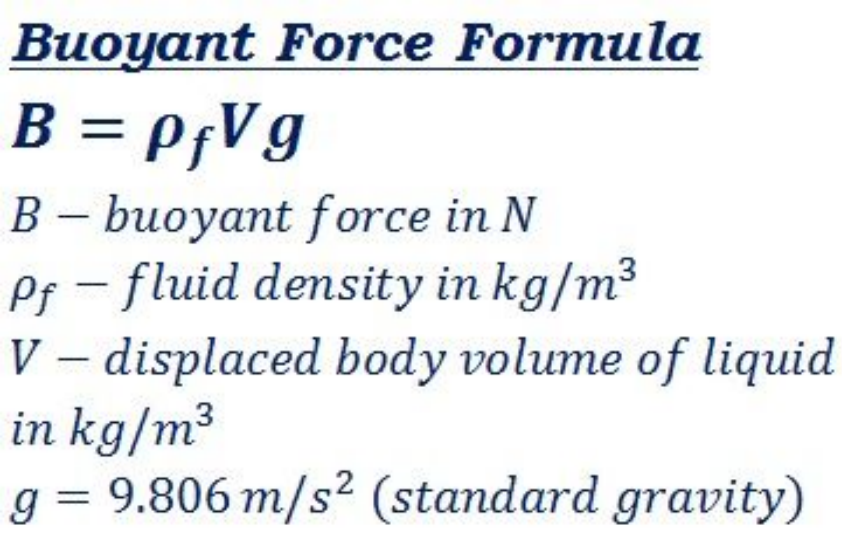

Figure 3.2: Buoyancy Force

For an object floating or submerged in water, the relationship between buoyancy and the object's weight determines its equilibrium state:

- **Object floats** when the buoyant force equals the object's weight:

$$F_b = W$$

- **Object sinks** if the weight exceeds the buoyant force:

$$W > F_b$$

- **Object remains neutrally buoyant** when the buoyant force and weight are equal, and the object is fully submerged without sinking or rising.

3.6 Resistances

In ship propulsion, resistance refers to the forces working against the forward motion of a vessel. The total resistance (R_T) is the sum of three primary components: **frictional resistance**, **residual resistance**, and **air resistance** as we see in [42].

Frictional Resistance (R_F) Frictional resistance is caused by the friction between the ship's hull and water. It is proportional to the wetted surface area of the hull and increases with the ship's speed. The frictional resistance can be calculated as:

$$R_F = C_F \cdot K$$

where C_F is the frictional resistance coefficient, and K is the reference force calculated as:

$$K = \frac{1}{2} \cdot \rho \cdot V^2 \cdot A_S$$

Here, ρ is the density of water, V is the ship's velocity, and A_S is the wetted surface area of the hull.

To determine the frictional resistance coefficient C_F, the Reynolds number R_n must first be calculated:

$$R_n = \frac{V \cdot L}{\nu}$$

where: - V is the speed of the ship, - L is the length of the submerged part of the ship, - ν is the kinematic viscosity of the fluid.

Once R_n is known, C_F can be calculated using the ITTC 1957 friction line formula:

$$C_F = \frac{0.075}{(\log_{10}(R_n) - 2)^2}$$

Residual Resistance (R_R) Residual resistance is a combination of wave-making

resistance and eddy resistance. Wave-making resistance arises from the energy lost to the waves generated by the ship, while eddy resistance is caused by turbulence and flow separation around the hull.

Calculation of the Residual Resistance Coefficient (C_R): To calculate the residual resistance coefficient, the following steps are applied:

Froude Number (F_n) The first step is to calculate the Froude number F_n, a dimensionless parameter that relates the ship's speed to the gravitational force acting on it. It is calculated as:

$$F_n = \frac{V}{\sqrt{g \cdot L}}$$

where:

- V is the speed of the ship (m/s),
- g is the gravitational acceleration ($9.81\,\mathrm{m/s^2}$),
- L is the length of the submerged part of the ship (m).

Prismatic Coefficient (C_P) and Block Coefficient (C_B)

The prismatic coefficient C_P and the block coefficient C_B describe the hull geometry. These coefficients are important in determining the hull's shape and its contribution to wave-making resistance.

- C_P: Measures the concentration of the ship's volume around its center.
- C_B: Describes the fullness of the hull shape and is the ratio of the submerged volume to the product of the ship's length, breadth, and draft.

Derived Constants (E, G, H, and K)

Using the Froude number and the geometric coefficients C_P and C_B, the constants E, G, H, and K can be determined. These constants adjust for the effects of the hull's shape on the residual resistance and are derived either from empirical formulas or through model testing.

Residual Resistance Coefficient: The Residual Resistance Coefficient C_R is then determined through empirical formulas based on the Froude number and the geometric coefficients. Once C_R is known, the residual resistance is calculated as:

$$R_R = C_R \cdot K$$

where K is the reference force, calculated as:

$$K = \frac{1}{2} \cdot \rho \cdot V^2 \cdot A_S$$

where:

- ρ is the density of water (kg/m^3),
- V is the speed of the ship (m/s),
- A_S is the wetted surface area of the hull (m^2).

Air Resistance (R_A) Air resistance refers to the resistance caused by air flowing over the part of the ship above the waterline. It is typically smaller than the other two resistances and is calculated as:

$$R_A = C_A \cdot K$$

where C_A is the air resistance coefficient.

Total Resistance The total resistance acting on the ship is the sum of all three components:

$$R_T = R_F + R_R + R_A$$

The power required to overcome the resistance at a specific speed V, known as **the effective towing power** (P_E), is given by:

$$P_E = R_T \cdot V$$

Chapter 4

Users View

4.1 Introduction

In this chapter, we will explore the user's experience while interacting with the application, focusing on the various controls and functionalities provided through its user interfaces (UIs). The chapter will detail how users navigate through the Main Menu and interact with key canvases such as the Choose Mode Canvas and the Choose Conditions Canvas. Each UI component will be accompanied by use case diagrams to illustrate the possible actions and user interactions.

We will also examine the Flow of the Application, highlighting the sequence of events from the user's perspective, from setting parameters to completing a session. Finally, we will present the Users Point of View, providing insight into how the application is designed to enhance user engagement and interaction. The functionalities and the use case diagramms will also be covered to ensure a comprehensive overview of the user journey.

4.2 Controls

The application leverages the capabilities of the Meta Quest Pro, offering an immersive and intuitive user experience. Instead of relying on traditional controllers, all interactions are conducted using the user's hands. This approach was chosen to enhance realism and create a more natural, seamless interface. By incorporating hand-tracking technology, users can perform actions such as selecting options, manipulating objects, and navigating menus directly through hand gestures. This design not only adds a layer of immersion but also simulates real-world interactions, making the experience more engaging.

The hand-tracking system detects various gestures, allowing users to grab, point, swipe, pinch and press with precision. The absence of physical controllers eliminates the need for additional equipment, enabling users to focus entirely on the virtual environment. This contributes to a more immersive and accessible experience, as users can interact with the system naturally without learning complex controller mechanics.

The decision to use hand-tracking was made to replicate real-world ship control tasks, aligning with the goal of providing a realistic training simulation for maritime operations.

4.3 Main Menu

The Main Menu canvas is the first canvas the user sees in the application. It contains a scenery with the ocean and 4 buttons the player can select, the start, credits, settings and exit buttons. When the player clicks the start button, the function canvas switch is called and the canvas changes to Choose Mode Canvas. If it clicks the credits button, then the credits panel opens. If it clicks the settings button then the canvas switch activates the settings canvas and if the exit button is pressed the application closes.

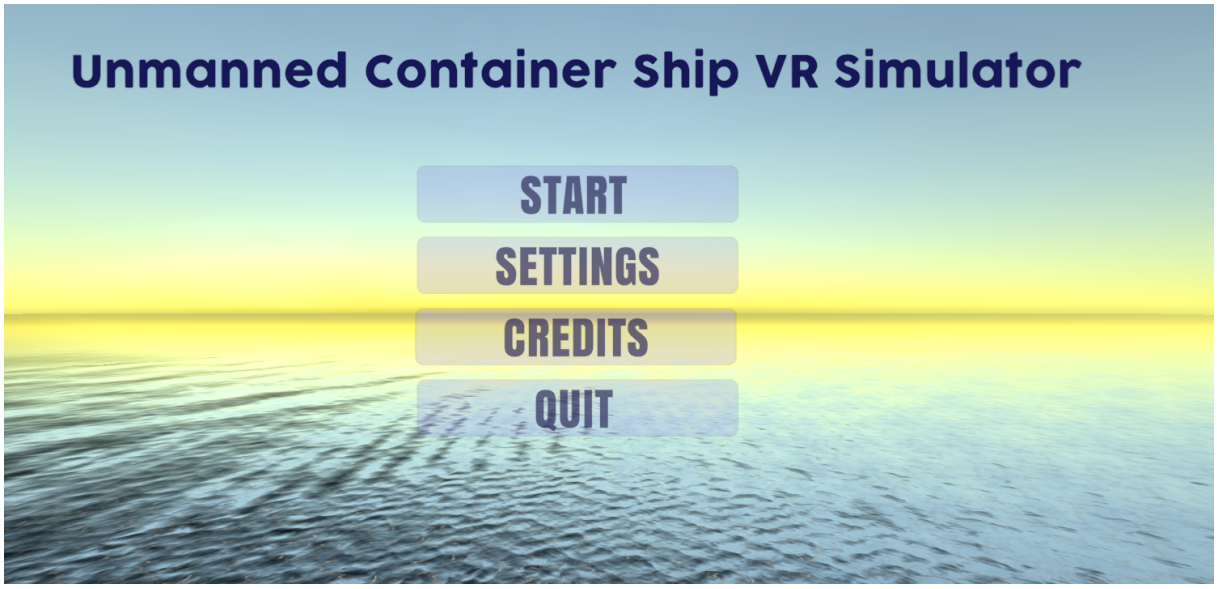

Figure 4.1: Canvas Main Menu

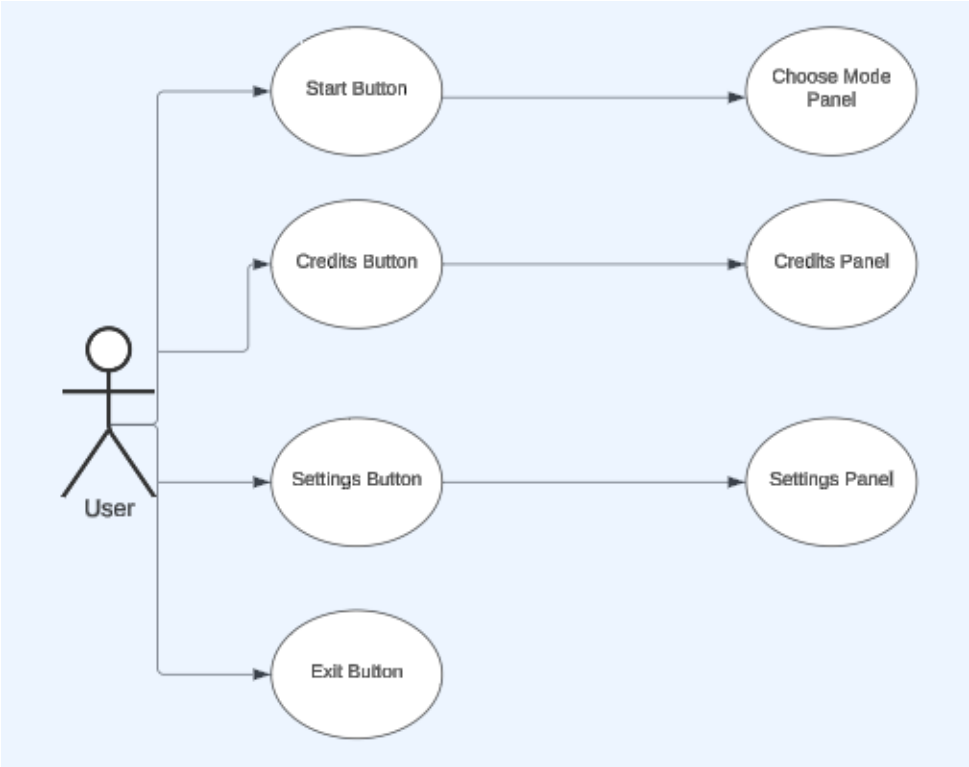

Figure 4.2: Canvas Main Menu Use Case

4.3.1 Choose Mode Canvas

In this panel, the user is presented with two distinct modes to choose from, each designed to offer a different experience within the application.

The first option is Exploration Mode, which allows the user to freely navigate through the virtual environment without any specific objectives. This mode is intended for users to familiarize themselves with the system and the maritime environment, offering a relaxed experience where they can explore the ocean, observe the ship's movement, and get accustomed to the controls and settings.

The second option is Scenario Mode, where the user is tasked with transporting containers from one port to another within a set time limit. In this mode, the user must navigate efficiently and manage the ship's operations under time pressure, simulating a real-world cargo transport scenario. This mode introduces an element of challenge, requiring users to apply their skills and knowledge to complete the mission successfully.

By offering these two modes, the application provides both a learning environment for newcomers and a more structured, goal-oriented experience for those ready to tackle specific tasks. This flexibility enhances user engagement, catering to different learning curves and preferences.

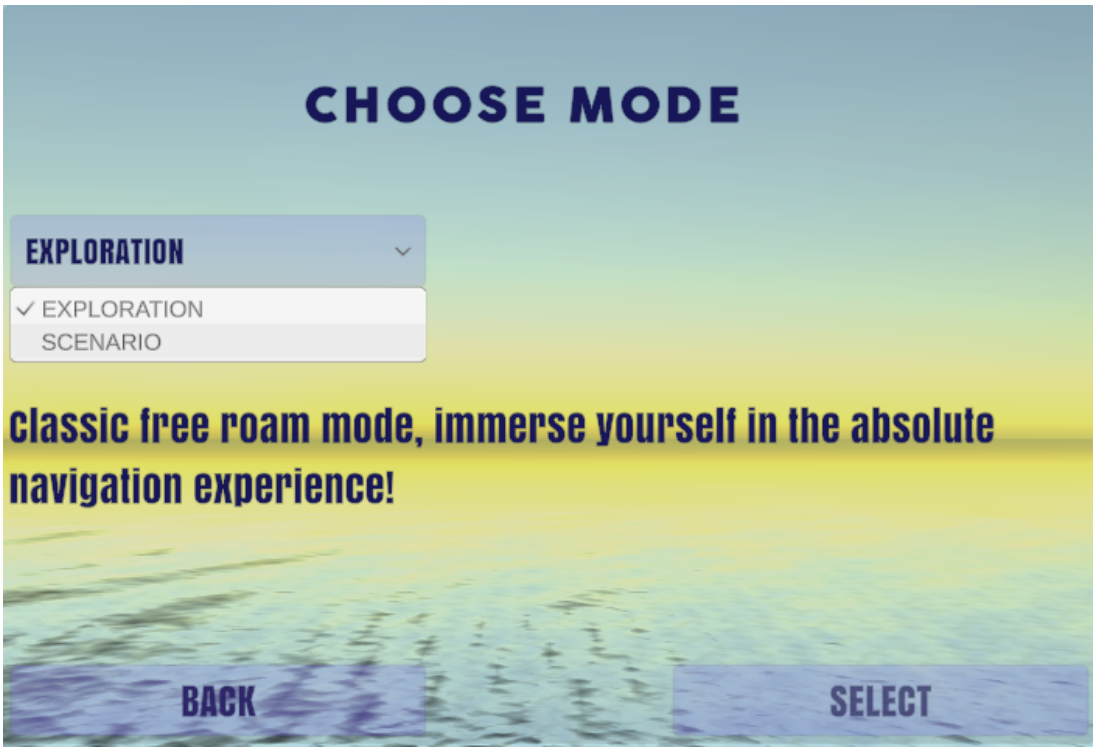

Figure 4.3: Choose Mode Canvas

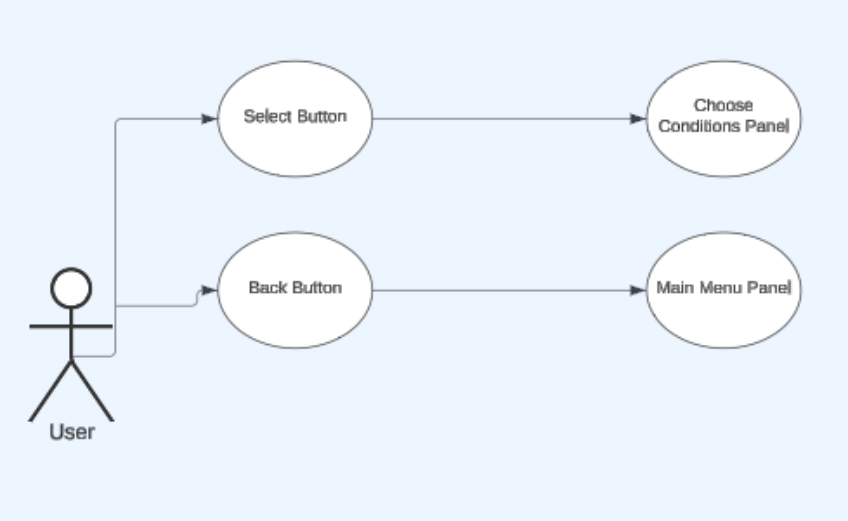

Figure 4.4: Canvas Choose Mode Use Case

4.3.2 Choose Conditions Canvas

The Choose Conditions Canvas allows the user to set various environmental and time-based parameters before starting the simulation. This interface offers a range of adjustable conditions to create a customized experience based on real-world factors. The user can select:

Date and Time: Users can choose the exact date and time of their simulation, with a convenient date picker and a slider for setting the time in UTC format.

Wind Direction: A slider is available to adjust the wind direction in degrees, offering precise control over this environmental variable.

Sea Waves: Users can adjust the intensity of the sea waves with a simple slider, simulating calm or rough sea conditions as per their preference.

Sky Conditions: Three options are provided for the sky: Clear, Cloudy, or Rainy, allowing the user to define the weather setting for their simulation.

Rain: A slider enables the user to adjust the level of rainfall, from light drizzle to heavy rain, further enhancing the realism of the weather conditions.

Visibility: Visibility can also be adjusted with a slider, allowing the user to simulate clear or foggy conditions that may affect navigation.

With these options, the user has full control over the environmental conditions, making each simulation unique and tailored to different training or exploration scenarios.

Figure 4.5: Choose Conditions Canvas

4.3.3 Flow of the Application

The flow varies slightly depending on whether the user selects Free Roam Mode or Scenario Mode, with different panels and events shaping the user's experience.

Free Roam Mode When the user selects Free Roam Mode, they are free to explore the environment without any specific objectives. The user can navigate the ocean and interact with the environment at their leisure. However, if the user collides with land masses or other ships, the application halts and restarts. Upon collision, a Crash Panel will appear, informing the user about the nature of the collision and notifying them that

the game will restart. This panel helps users understand what went wrong and prepares them for a fresh start.

Figure 4.6: Crash Panel

Scenario Mode In Scenario Mode, the user is tasked with completing a specific objective—transporting containers from one port to another within a given time frame. The flow in this mode is more structured and goal-oriented.

At the beginning of the scenario, a Mission Panel is presented to the user, outlining the objectives they need to achieve to successfully complete the scenario. This panel clearly states the task of loading the cargo and safely transporting it to the destination. After the cargo is loaded onto the ship, another Cargo Loaded Panel appears, providing instructions for the next step in the scenario, such as setting the correct course and navigating toward the destination.

As the user progresses, they must carefully navigate the ocean and manage the ship's operations. Similar to Free Roam Mode, if the user collides with land or another ship, the same Crash Panels will appear, informing the user of the type of collision and restarting the game.

If the user fails to complete the task within the given time, a Time Elapsed Panel will be displayed, notifying them that the time has run out and the scenario is incomplete. This panel provides feedback and encourages the user to retry the scenario.

On successful completion of the scenario—if the user manages to transport the containers within the designated time without any crashes—a final Completion Panel appears. This panel congratulates the user for finishing the scenario successfully and informs them that they completed the task on time.

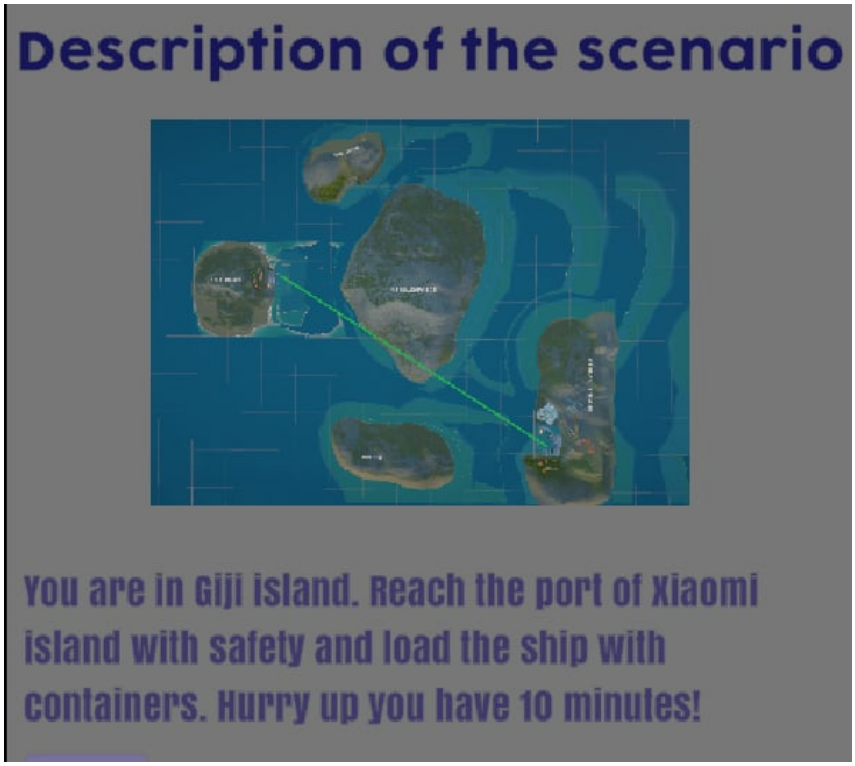

Figure 4.7: Mission Panel Canvas

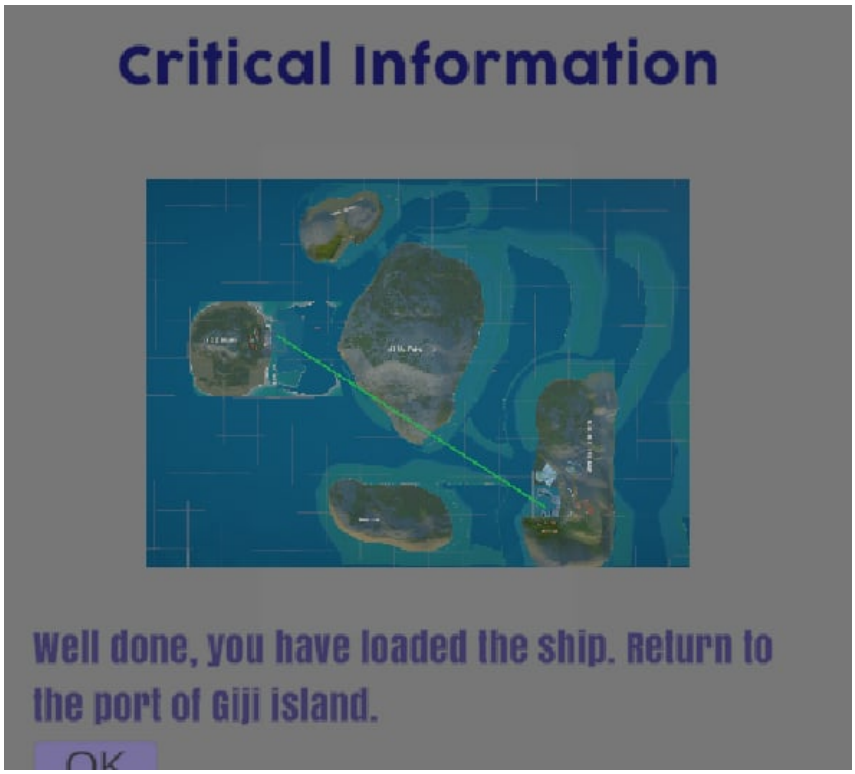

Figure 4.8: Choose Conditions Canvas

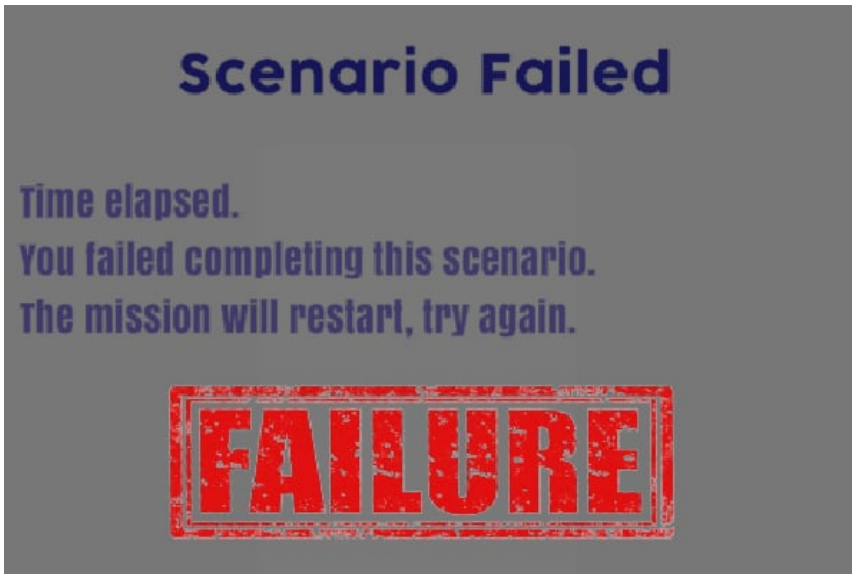

Figure 4.9: Time Elapsed Panel

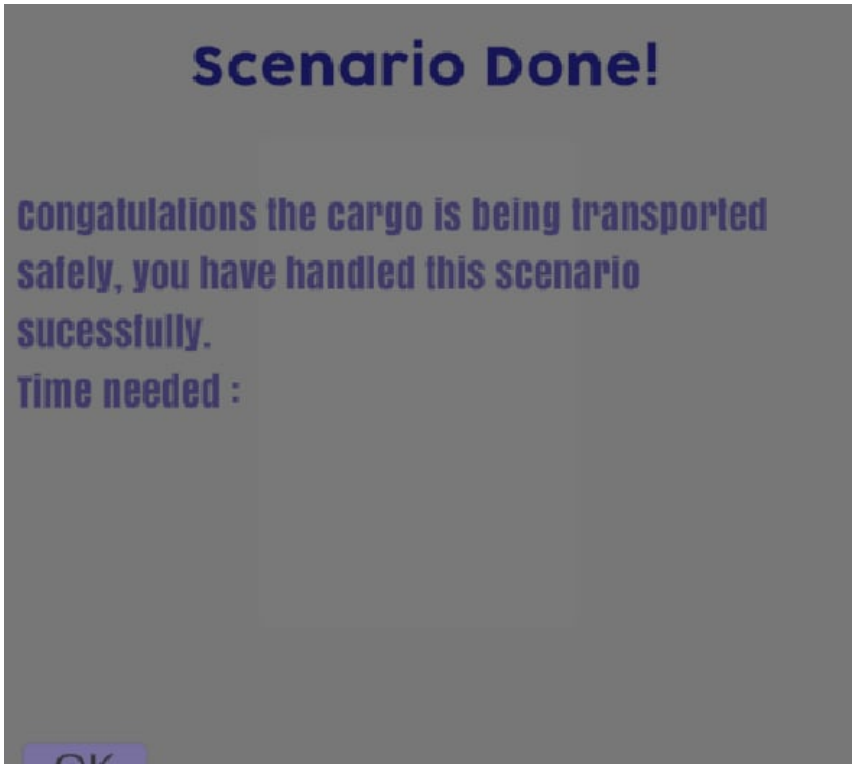

Figure 4.10: Completion Panel

4.4 Users Point Of View

The virtual control room, as shown in the image below, serves as the central hub for user interactions with the unmanned cargo ship system. This immersive environment replicates the layout of a real-world maritime control room, providing users with a variety of technologies and displays to manage and monitor the ship's operations. Through hand gestures and natural interactions, the user can control each element of the room, enhancing both realism and immersion.

Figure 4.11: The Virtual Control Room

1. Radar Display The Radar Display is located on the left side of the room and provides real-time tracking of nearby ships obstacles. The radar sweeps continuously, offering updates on the vessel's surroundings. Users can interact with the radar using hand gestures, increasing the range it covers and the speed of the sweep so how fast it is scanning. This interaction is essential for maintaining situational awareness, especially in challenging environments like ours in which ships are moving through the ocean.

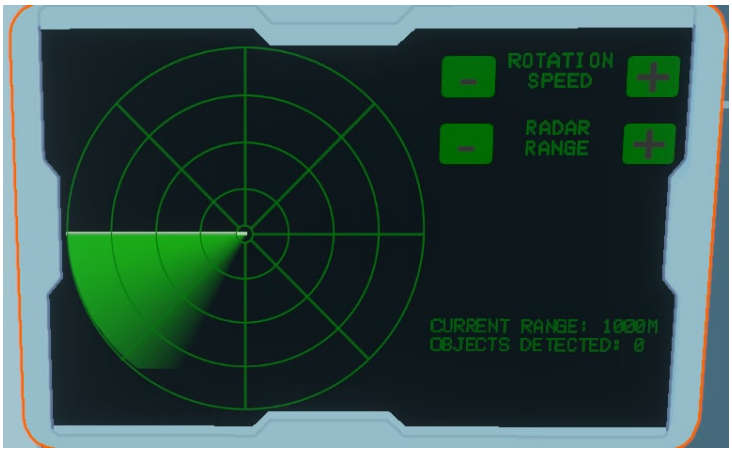

Figure 4.12: Radar

2. Inertial Measurement Unit (IMU) and Sonar System Adjacent to the radar is the Inertial Measurement Unit (IMU) and Sonar System, which tracks the ship's movement and provides data on its orientation. The IMU displays critical information about the ship's pitch, surge and the depth of the sea below the ship giving in this way vital information that the remote operator should have in mind while controlling the ship.

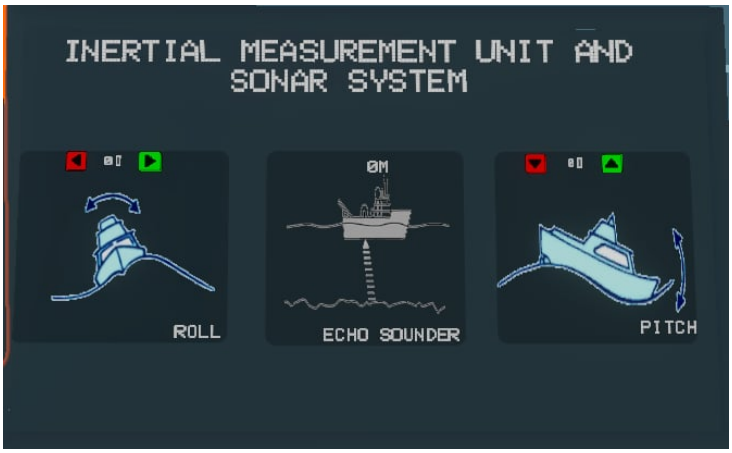

Figure 4.13: IMU and Sonar System

3. Main View Monitors The three Main View Monitors provide a first-person perspective from different angles around the ship. Users can switch between these views, enabling them to observe the surroundings from multiple perspectives. These monitors also feature a Night Vision option, which can be toggled on when operating in low-visibility conditions. Hand interactions allow users to switch views seamlessly, enhancing their control over the ship's external environment.

Figure 4.14: Main View Monitors

4.The Bathymetric LIDAR Display, located to the left of the minimap, provides a visual representation of the underwater terrain. The system simulates the real-world functionality of LIDAR technology by emitting pulses that detect underwater landmasses. When these landmasses are detected, the screen displays small blue circles, as shown in the image, indicating the presence of submerged obstacles or terrain features.

This information is critical for avoiding potential hazards below the surface and ensuring safe navigation in shallow or unknown waters. The user can monitor this display to make necessary adjustments to the ship's course, further enhancing situational awareness and preventing collisions with underwater obstacles.

Figure 4.15: Bathymetric LIDAR

5. Navigation Map - GPS System On the right side, the Navigation Map provides a top-down view of the ship's current location, surrounding terrain, and destinations. This map is essential for planning routes and monitoring the ship's progress. Users can interact with the map by zooming in and out, setting way points and tracking their journey across the ocean. In addition this screen has an autopilot mode with which the user by setting a waypoint on the minimap, the ship will find the shortest and safest route to reach it. The user can also interact and change the focus of the gps either on the ship or free to be moved.

Figure 4.16: GPS System

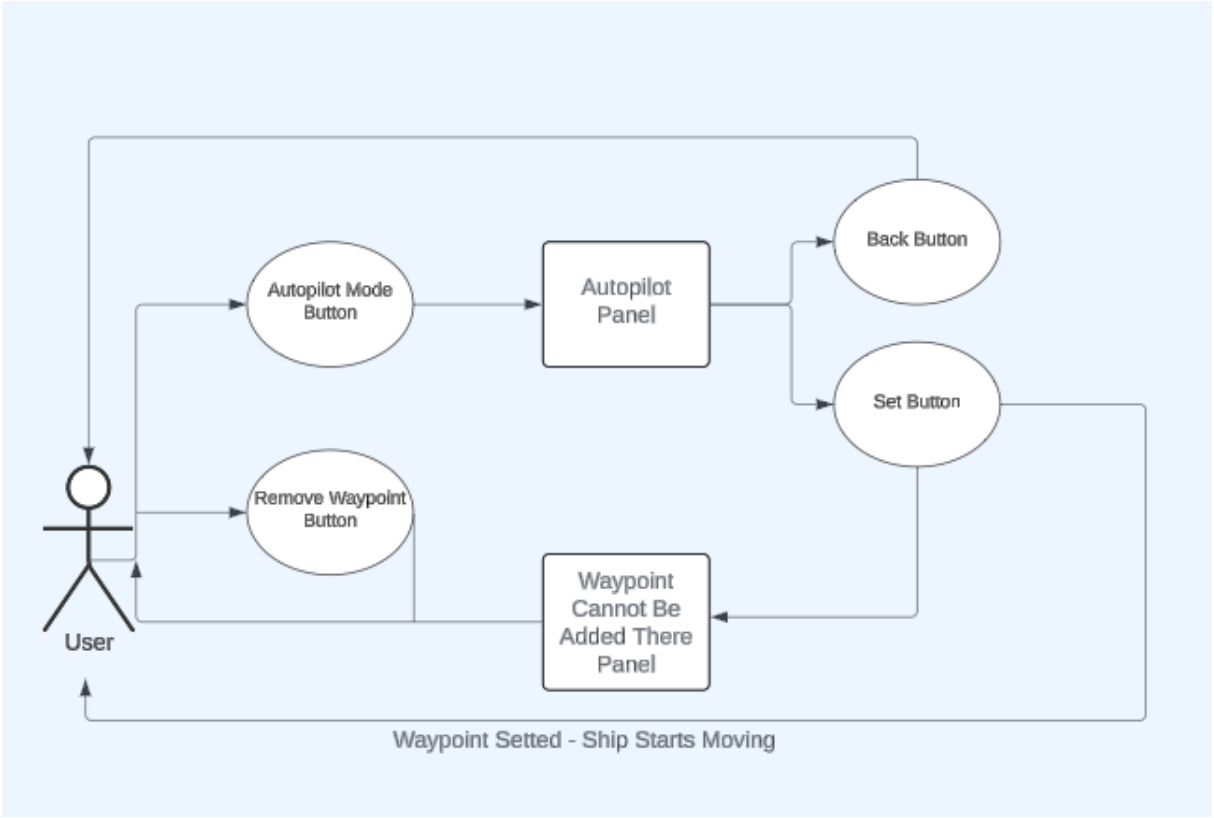

Figure 4.17: Autopilot Mode Use Case

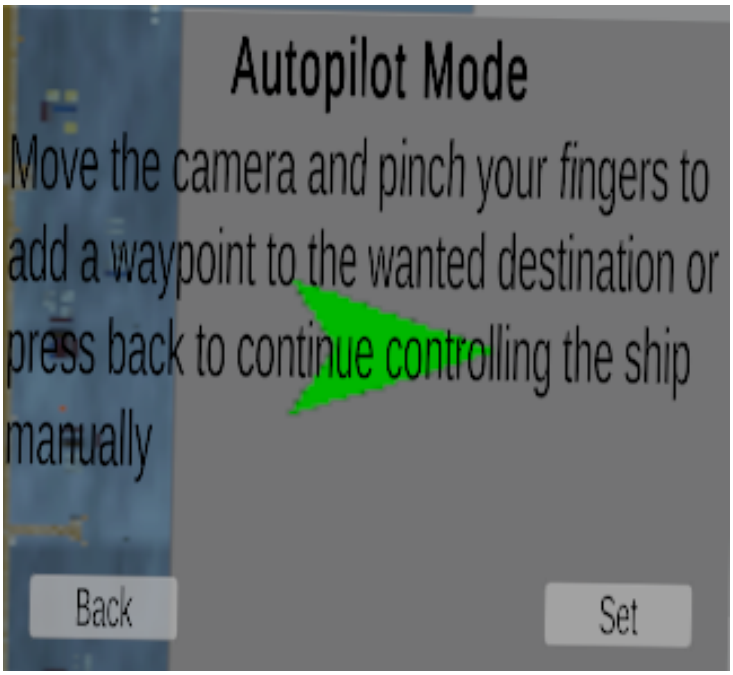

Figure 4.18: Autopilot Panel

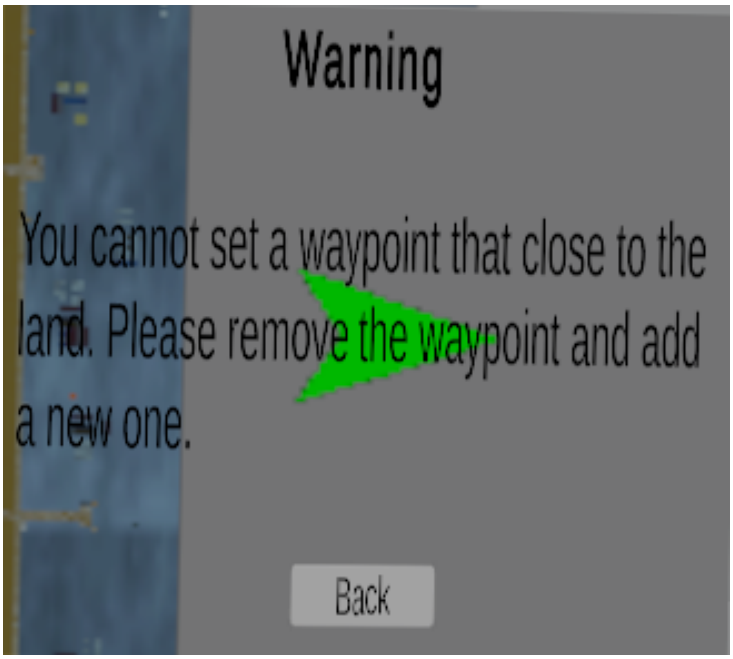

Figure 4.19: Warning Panel

6.The Gyroscope Display, located on the rightmost screen, informs the operator of the ship's current heading in degrees. This real-time data is essential when steering, allowing the user to make precise adjustments to the ship's course. As the ship turns, the display updates the heading, ensuring the operator maintains full control over the vessel's direction. Below of the gyroscope is the Wave Direction Panel, which provides the operator with information on the degrees of the incoming wave direction. This data

helps the operator understand how the waves are impacting the ship, enabling them to adjust the course or speed accordingly.

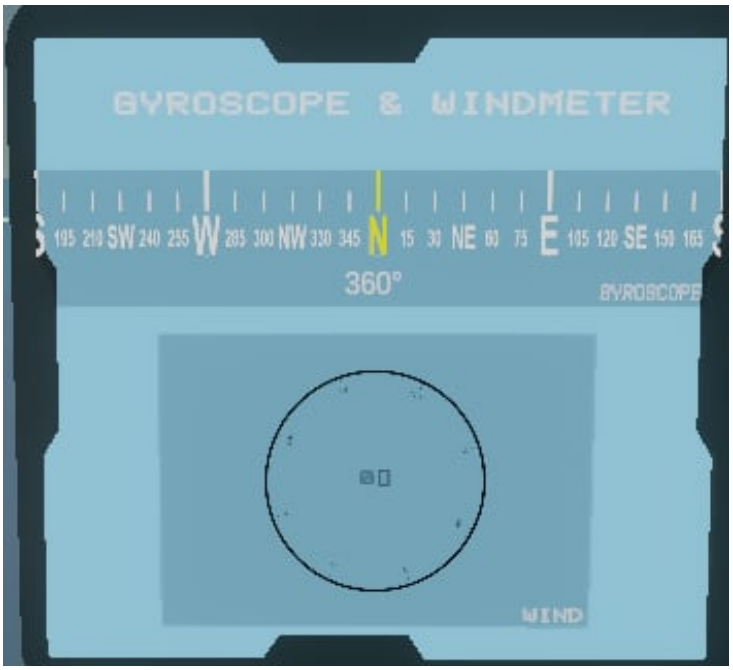

Figure 4.20: Gyroscope and Windmeter

7. At the center of the control room is the Steering Wheel and Engine Control System. These serve as the primary interaction points for navigating the ship. The user can steer the virtual steering wheel using hand gestures, simulating real-life ship steering, and adjust the engine speed through intuitive throttle controls. These manual controls are vital in both Free Roam and Scenario modes, allowing users to perform precision maneuvers and respond quickly to changes in the environment. Around the thrust lever there are buttons for controlling the anchor, the engine power, the astern mode, the lights of the ship and its intensity and lastly the horn with a guide indicating some of the most important horn signals the user should learn.

The virtual control room integrates all these technologies into a cohesive system that mirrors the complexity of a real-world maritime control center. By using their hands to interact with each system, users are offered an unparalleled level of control and immersion, making this an effective training tool for real-world ship navigation and unmanned vessel management.

Chapter 5

Implementation

5.1 Introduction

This chapter outlines the technical implementation of the project, detailing how various components were developed and integrated to create the final application. The focus is on constructing an immersive maritime training simulation using the Meta Quest Pro, leveraging advanced features such as hand tracking, realistic control environments, and the interaction of various ship systems.

The development process was divided into key areas: User Interface Integration, Virtual Control Room Design, Hand-Tracking Interaction, and Simulation Logic. Each area plays a critical role in creating a seamless, hands-on experience for users as they navigate through free roam and scenario modes.

All elements of the project were implemented using the Unity game engine, with C sharp scripts controlling the behavior, physics, and interactions within the application. From the integration of various control panels (e.g., radar, sonar, LIDAR, gyroscope) to managing real-time environmental effects such as weather conditions and collisions, the implementation relied heavily on precise scripting to ensure both realism and functionality. Furthermore, specialized scripts were written to manage the physics and mathematical models needed for accurate ship behavior simulation.

By breaking down the implementation into these core areas, this chapter provides a comprehensive overview of the technical efforts that brought the virtual maritime environment to life. Each section will explore the methods, tools, and approaches used to create an immersive and educational experience for users.

5.2 Ship Movement

The movement of the ship within the simulation was carefully modeled to reflect the complex physics that govern real-world maritime navigation. A core aspect of this system is the implementation of a buoyancy simulation, which mimics the forces that allow the ship to float and remain stable in water. This was crucial in creating a realistic experience, as buoyancy plays a significant role in the behavior of any vessel, especially an unmanned cargo ship.

In addition to buoyancy, resistance forces were incorporated to simulate the various challenges the ship encounters as it moves through the water. These resistances, which include drag from the water, wave resistance, and friction, all contribute to making the movement more realistic and responsive to environmental conditions. This means that the ship will not move effortlessly; rather, its speed and maneuverability are influenced by these opposing forces, requiring careful navigation from the user.

Lastly, a propulsion force was added to simulate the engine's output, allowing the ship to move forward. This force interacts dynamically with the resistance forces, meaning that as the ship increases speed, it experiences greater resistance from the water, which in turn affects how much power is required to maintain or adjust course.

The combined effect of these forces—buoyancy, resistance, and propulsion—impacts not only the ship's speed and movement but also its orientation. The ship's roll, pitch, and yaw are influenced by the forces acting on it, making steering more complex in turbulent conditions. For example, as waves apply resistance from varying directions, the ship's roll (side-to-side tilt) and pitch (up-and-down tilt) will be affected, requiring constant adjustments by the user to maintain stability and course.

This realistic simulation of forces makes navigating the ship in the simulation a challenging task, especially in the face of natural resistances and environmental conditions. By incorporating these forces, the movement of the ship mirrors real-world scenarios, adding depth and complexity to the training experience.

5.2.1 Ship Dimensions and Values - Physics Simplifications

In order to simulate a ship as accurately as possible, I chose to base my model on the Maersk Honam, a large container vessel. Realistic dimensions are critical for simulating not only how the ship moves through water but also how external forces like resistance, buoyancy, and propulsion affect its performance.

Dimensions of Maersk Honam : The actual dimensions of the Maersk Honam, which are reflected in the simulation, are as follows:

- Length: 353 meters
- Beam (Width): 53.5 meters
- Draft: 15 meters
- Depth: 29.9 meters [43].

These parameters are crucial in determining how the ship interacts with water (e.g., how deep it sits and how much drag it experiences), directly influencing how buoyancy and resistance forces are calculated.

Simplification of Ship Weight :

One of the key simplifications made in the simulation was regarding the ship's weight. The real deadweight tonnage (DWT) of the Maersk Honam is approximately 162,051 tons [43]. However, I set the weight in the simulation to 550,000 kg (or 550 tons). This is far from the actual weight, and the reasoning for this adjustment is tied to the collider used in the simulation.

In Unity, I employed a box collider for the ship to simplify collision detection and movement. However, since the box collider doesn't perfectly match the ship's hull shape, it effectively increases the surface area of the ship, leading to exaggerated resistance forces. This increased resistance results in the ship encountering more drag than it would with a more accurate collider.

Balancing the Forces with Weight Adjustment

To balance out the increased resistance caused by the box collider, I reduced the ship's weight in the simulation. According to *Newton's Second Law* $F = m \times a$, reducing the mass allows the ship to move more freely, despite the extra resistance. This ensures the ship can still achieve its intended speed of 24 knots, corresponding to the real Maersk Honam's service speed [43], without increasing the propulsion force.

If the actual weight of the ship had been used, the propulsion force would have been insufficient due to the excessive drag caused by the simplified collider. By reducing the weight, the balance is maintained without exceeding the calculated propulsion force of 4468432 Newtons, derived from the equation $T = \frac{75 \times Thp}{V(m/s)}$.

Propulsion and Engine Power

The Maersk Honam is powered by a 54,940 kW engine [43]. This engine power was the basis for calculating the propulsion force in the simulation. The ship uses a *single-shaft, fixed-pitch propeller* system, ensuring steady propulsion through the water. In the simulation, the propulsion force was set to 4468432 Newtons, ensuring that the ship reaches its maximum speed of 24 knots without overwhelming the system with excessive force.

Collider Simplification

Using a box collider was another necessary simplification for performance reasons. While more complex colliders could have matched the exact contours of the ship, this would have significantly increased computational demands. The box collider simplifies the process of collision detection and movement, ensuring efficient performance. However, it does contribute to the increased resistance forces during the ship's movement, which is why the weight adjustment was necessary to maintain a balanced and realistic simulation.

In summary, through these simplifications and adjustments, I was able to maintain a balance between realism and computational efficiency, ensuring that the Maersk Honam behaves realistically in the simulation environment while keeping the system performance optimal.

5.2.2 Buoyancy

To accurately simulate the buoyancy of the ship, I used the Physical Voxel method from buoyancy script. This method calculates buoyancy by dividing the ship into small voxel units, each of which interacts with the water individually, allowing for a detailed and physically accurate buoyancy calculation. Below, I will explain how each relevant function works, particularly focusing on how buoyancy is calculated and applied to the ship. The mathematical equations were presented in Chapter3.

Init() The initialization function Init() sets up the necessary components for calculating buoyancy based on the selected type (PhysicalVoxel in this case). In particular, it calls four key methods:

```
private void Init()
{
    _voxels = null;

    switch (_buoyancyType)
    {

        case BuoyancyType.PhysicalVoxel:
            SetupColliders();
            SetupVoxels();
            SetupData();
            SetupPhysical();
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }
}
```

Figure 5.1: Init()

SetupColliders() Ensures that the ship has colliders (physical boundaries) that can be used to divide the ship into voxels. If no colliders are found, a basic BoxCollider is added as a fallback.

SetupVoxels() The most important method for voxelization, this divides the ship's body into smaller cubic sections (voxels) using the SliceIntoVoxels() method. These voxels represent individual units of the ship's volume, and the number of voxels depends on the voxelResolution (in my case, set to 10). Each voxel can independently interact with water, allowing for a more granular buoyancy calculation.

```
private void SliceIntoVoxels()
{
    var t = transform;
    var rot = t.rotation;
    var pos = t.position;
    var size = t.localScale;
    t.SetPositionAndRotation(Vector3.zero, Quaternion.identity);
    t.localScale = Vector3.one;

    _voxels = null;
    var points = new List<Vector3>();

    var rawBounds = VoxelBounds();
    _voxelBounds = rawBounds;
    _voxelBounds.size = RoundVector(rawBounds.size, voxelResolution);
    for (var ix = -_voxelBounds.extents.x; ix < _voxelBounds.extents.x; ix += voxelResolution)
    {
        for (var iy = -_voxelBounds.extents.y; iy < _voxelBounds.extents.y; iy += voxelResolution)
        {
            for (var iz = -_voxelBounds.extents.z; iz < _voxelBounds.extents.z; iz += voxelResolution)
            {
                var x = (voxelResolution * 0.5f) + ix;
                var y = (voxelResolution * 0.5f) + iy;
                var z = (voxelResolution * 0.5f) + iz;

                var p = new Vector3(x, y, z) + _voxelBounds.center;

                var inside = false;
                foreach (var t1 in colliders)
                {
                    if (PointIsInsideCollider(t1, p))
                    {
                        inside = true;
                    }
                }
                if (inside)
                    points.Add(p);
            }
        }
    }

    _voxels = points.ToArray();
    t.SetPositionAndRotation(pos, rot);
    t.localScale = size;
    density = gameObject.GetComponent<Rigidbody>().mass / volume;
}
```

Figure 5.2: SliceIntoVoxels()

SetupData() Initializes the arrays that store information about each voxel's position, water height, and forces acting on them. These arrays are used later to track how submerged each voxel is and how it affects the ship's overall buoyancy.

SetupPhysical() Sets up the ship's physics by ensuring it has a Rigidbody component, which allows the ship to interact with the Unity physics engine. It also pre-calculates the Archimedes Force for each voxel based on water density and voxel size. This is the force responsible for making the ship float.

```
private void SetupPhysical()
{
    if (!TryGetComponent(out _rb))
    {
        _rb = gameObject.AddComponent<Rigidbody>();
        Debug.LogError($"Buoyancy:Object \"{name}\" had no Rigidbody. Rigidbody has been added.");
    }

    var archimedesForceMagnitude = WaterDensity * Mathf.Abs(Physics.gravity.y) * volume;
    _localArchimedesForce = new float3(0, archimedesForceMagnitude, 0) / _voxels.Length;
    LocalToWorldJob.SetupJob(_guid, _voxels, ref _samplePoints);
}
```

Figure 5.3: SetupPhysical()

FixedUpdate() method, the script tracks each voxel's movement and calculates its buoyant force at every physics step.

```
private void FixedUpdate()
{
    var submergedAmount = 0f;

    switch (_buoyancyType)
    {
        case BuoyancyType.PhysicalVoxel:
            {
                LocalToWorldJob.CompleteJob(_guid);
                //Debug.Log("new pass: " + gameObject.name);
                Physics.autoSyncTransforms = false;

                for (var i = 0; i < _voxels.Length; i++)
                {
                    //Debug.Log(submergedAmount);
                    BuoyancyForce(_samplePoints[i], _velocity[i], Heights[i].y + waterLevelOffset, ref submergedAmount, ref _debugInfo[i]);
                }


                volumeSubmerged = submergedAmount * volume; //submerged amount how much of the ship is sumerged

                break;
            }
```

Figure 5.4: FixedUpdate()

BuoyancyForce() The Physical Voxelization method applies Archimedes' Principle to each voxel. Archimedes' Principle states that the buoyant force acting on an object is equal to the weight of the water displaced by that object. In this case, each voxel is considered a small part of the ship, and its buoyancy is calculated based on how submerged it is in water.

k: Represents how much of the voxel is submerged, ranging from 0 (not submerged) to 1 (fully submerged).

```
private void BuoyancyForce(Vector3 position, float3 velocity, float waterHeight, ref float submergedAmount, ref DebugDrawing debug)
{
    debug.Position = position;
    debug.WaterHeight = waterHeight;
    debug.Force = Vector3.zero;

    if (!(position.y - voxelResolution < waterHeight)) return; //check that the voxel is underwater

    var k = math.clamp(waterHeight - (position.y - voxelResolution), 0f, 1f);

    submergedAmount += k / _voxels.Length;


    var force = math.sqrt(k) * _localArchimedesForce;
    _rb.AddForceAtPosition(force, position);

    debug.Force = force; // For drawing force Gizmos

}
```

Figure 5.5: BuoyancyForce()

5.2.3 Resistances

Residual Resistance Residual resistance as we mentioned earlier, is the force acting against the ship due to wave-making and the shape of the hull. This force primarily increases as the ship gains speed and begins to create larger waves in the water. The ResidualResistanceCoefficient method calculates the coefficient Cr, which is then used to determine the magnitude of the residual resistance.

```
private void CalculateResidualResistance(float hullWetterArea, float waterDensity)
{
    Rigidbody rigidbody = GetComponent<Rigidbody>();
    float Cr = (float)ResidualResistanceCoefficient(rigidbody.velocity.magnitude, loa); //residualResistanceCoeff


    Vector3 velocity = rigidbody.velocity;
    float residualResistanceMagn = Cr * 1 / 2 * waterDensity * velocity.magnitude * velocity.magnitude * hullWetterArea;
    Vector3 residualResistance  = residualResistanceMagn * (-rigidbody.velocity.normalized);

    if (velocity.magnitude < 1.5)
    {
        residualResistance = new Vector3(0, 0, 0);
    }
    rigidbody.AddForce(residualResistance);
}
```

Figure 5.6: CalculateResidualResistance()

```
public static double ResidualResistanceCoefficient(float velocity, float length)
{
    //Needed for the calculation of the residual coefficient
    float M = 0.65f; //Prismatic Coefficient
    float Cp = 0.7f; //Block Coefficient
    double froudeNumber;

    //Debug.Log(velocity);

    double E, G, H, K, Cr;
    float g = 9.81f; //g =10m/s^2


        froudeNumber = velocity / Math.Pow(g * length, 1.0/2);
        //Debug.Log(froudeNumber);

        //E component
        float A0 = 1.35f - 0.23f * M + 0.012f * M * M;
        double A1 = 0.0011f * Math.Pow(M, 9.1f);
        float N1 = 2 * M - 3.7f;

        E = (A0 + 1.5f * Math.Pow(froudeNumber, 1.8f) + A1 * Math.Pow(froudeNumber, N1) * (0.98 + 2.5f / (M - 2) + Math.Pow(M - 5, 4) * Math.Pow(froudeNumber - 0.1, 4)));

        //Debug.Log(E);
        //G component
        double B1 = 7 - 0.09 * M * M;
        double B2 = Math.Pow(5 * Cp - 2.5f, 2);
        double B3 = Math.Pow(600 * Math.Pow(froudeNumber - 0.315f, 2) + 1, 1.5f);

        G = B1 * B2 / B3;
        //Debug.Log(G);

        //H component
        H = Math.Exp(80 * (froudeNumber - (0.04f + 0.59f * Cp) - 0.015f * (M - 5)));
        //Debug.Log(H);

        //K component
        K = (180 * Math.Pow(froudeNumber, 3.7f) * Math.Exp(20 * Cp - 16));
        //Debug.Log(K);

        //Final Caclulation
        Cr = (E + G + H + K) / 1000;
        //Debug.Log(Cr);

    return Cr;

}
```

Figure 5.7: ResidualResistanceCoefficient()

The residual resistance coefficient Cr is calculated using empirical formulas based on Froude numbers and hull characteristics: The Froude number compares the ship's speed to the wave propagation speed, which gives an indication of the ship's efficiency in creating waves. The coefficients E,G,H,K are empirically derived and depend on the shape of the ship (through prismatic and block coefficients) and the ship's speed. This method simulates how real ships generate waves, with residual resistance increasing significantly as the ship moves faster.

Frictional Resistance Frictional resistance arises from the interaction between the hull and the water, often caused by the water's viscosity. This resistance depends on the wetted area of the hull and is proportional to the square of the ship's velocity.

```
private void CalculateFrictionalResistance(float hullWetterArea,float waterDensity)
{
    Rigidbody rigidbody = GetComponent<Rigidbody>();
    float Cf = FrictionalResistanceCoefficient(buObj.density,rigidbody.velocity.magnitude,loa); //frictionallResistanceCoeff



    Vector3 velocity = rigidbody.velocity;
    //Debug.Log(velocity);
    float frictionalResistanceMagn = Cf * 1 / 2 * waterDensity * velocity.magnitude * velocity.magnitude * hullWetterArea;
    Vector3 frictionalResistance = frictionalResistanceMagn * (-rigidbody.velocity.normalized);

    if (velocity.magnitude < 1.5 ) //if the ship is not moving
    {
        frictionalResistance = new Vector3(0,0,0);
    }

    //Debug.Log(frictionalResistance.magnitude);

    rigidbody.AddForce(frictionalResistance);
}
```

Figure 5.8: CalculateFrictionalResistance()

```
public static float FrictionalResistanceCoefficient(float rho, float velocity, float length)
{
    //Reynolds number

    // Rn = (V * L) / nu
    // V - speed of the body
    // L - length of the sumbmerged body
    // nu - viscosity of the fluid [m^2 / s]

    //Viscocity depends on the temperature, but at 20 degrees celcius:
    float nu = 0.000001f;
    //At 30 degrees celcius: nu = 0.0000008f; so no big difference

    //Reynolds number
    float Rn = (velocity * length) / nu;

    //The resistance coefficient
    float Cf = 0.075f / Mathf.Pow((Mathf.Log10(Rn) - 2f), 2f);

    return Cf;
}
```

Figure 5.9: FrictionalResistanceCoefficient()

Here, frictional resistance is calculated similarly to residual resistance but uses a different coefficient, Cf, the frictional resistance coefficient. The frictional resistance coefficient Cf is calculated using the Reynolds number (Rn).The Reynolds number describes the ratio of inertial forces to viscous forces and helps in calculating the drag based on flow characteristics. The frictional resistance coefficient Cf is an empirical formula based on Rn. It decreases as Rn increases, meaning that as the ship moves faster, the frictional resistance grows.

Air Resistance Air resistance is calculated but not applied because the contribution of air resistance is small for large cargo ships like the Maersk Honam. In reality, air resistance contributes only a minor percentage to the total drag on a ship because most of the ship's resistance comes from the interaction between the hull and the water. For ships

like the Maersk Honam, air resistance accounts for only about 2-3% of the total resistance, making it negligible compared to water resistance. Thus, omitting air resistance is a reasonable simplification in this context, as the ship's mass and hydrodynamic resistance are far more influential than air drag.

Calculation of the wetted area

The wetted area, which represents the portion of the ship's hull submerged in water, changes continuously due to wave motion and ship movements. Therefore, it was essential to create an approach that would dynamically adjust the wetted area frame by frame.

To achieve this, I used the voxel-based approach. The ship's collider as we said was divided into voxels, that represent discrete sections of the hull. In every frame, I determined which of these voxels are below the waterline (i.e., submerged). By isolating the submerged voxels, I could then calculate the wetted area.

The process for calculating the wetted area for each frame involved the following steps:

- Voxels Submerged Under the Waterline: First, I identified which voxels of the ship were submerged by comparing their positions to the water surface. These submerged voxels are critical for calculating the wetted area, as they represent the portions of the hull in contact with the water.

- Boundary Voxels: Once the submerged voxels were identified, I further filtered the voxels to obtain those that represent the boundaries of the submerged portion of the hull. Specifically, I identified the lowest submerged voxels (bottom boundary), the leftmost, rightmost, frontmost, and backmost voxels. These boundary voxels were used to calculate the projected surface areas.

- Projection and Area Calculation: For each set of boundary voxels (bottom, left, right, front, back), I projected them onto a plane perpendicular to the direction in question (e.g., projecting the frontmost voxels onto a plane to calculate the front wetted area). I calculated the dimensions of each projection by finding the minimum and maximum bounds of the voxels in the respective direction. By multiplying the projection dimensions (e.g., width and height), I estimated the surface area for each side of the ship.

- Summing the Areas: After estimating the areas for each side (bottom, front, back, left, right), I summed them together to get the total wetted area for the current frame. Since the ship's motion changes dynamically, this calculation was performed for each frame, ensuring that the wetted area was updated continuously as the ship interacted with waves.

```
public float EstimateWettedArea(List<Vector3> layerVoxels, Vector3 direction)
{
    // Calculate the bounds of the layer voxels
    Vector3 minBounds = Vector3.positiveInfinity;
    Vector3 maxBounds = Vector3.negativeInfinity;

    foreach (Vector3 voxel in layerVoxels)
    {
        minBounds = Vector3.Min(minBounds, voxel);
        maxBounds = Vector3.Max(maxBounds, voxel);
    }

    // Calculate the dimensions of the projection
    float projectionDimension1 = 0f;
    float projectionDimension2 = 0f;



    if (direction == Vector3.forward || direction == Vector3.back) //back - front wetted area
    {

        projectionDimension1 = maxBounds.y - minBounds.y;
        projectionDimension2 = maxBounds.z - minBounds.z;


    }
    else if (direction == Vector3.left || direction == Vector3.right) //left - right wetted area
    {
        projectionDimension1 = maxBounds.y - minBounds.y;
        projectionDimension2 = maxBounds.x - minBounds.x;
    }

    else //bottom wetted area
    {
        projectionDimension1 = maxBounds.x - minBounds.x;
        projectionDimension2 = maxBounds.z - minBounds.z;
    }



    // Calculate and return the wetted area
    return projectionDimension1 * projectionDimension2;
}
```

Figure 5.10: EstimateWettedArea()

5.2.4 Propulsion - Steering

In this section, I describe the propulsion and turning system used for controlling the vessel's movement. This system, uses physics-based interactions to allow the ship to accelerate and steer dynamically in response to the environment. The PropulsionSystem script is responsible for the movement and turning of the vessel. It works with the vessel's Rigidbody component, which handles all physical interactions like acceleration, torque, and forces exerted on the ship. The system ensures that the vessel moves forward or backward, depending on the power applied, and can also steer left or right using torque.

Acceleration: The system calculates the ship's speed and applies forces to propel it forward. The propulsion force is applied based on engine power and other variables such as the ship's speed and direction.

Turning: Steering is managed by applying torque to the ship's rigidbody. The turning force is calculated based on input or automated commands, allowing the vessel to change direction smoothly.

Both the propulsion and steering mechanisms are tied to the ship's interaction with water dynamics, allowing the vessel to react realistically to environmental changes like

waves. The system also checks whether the engine is submerged and active before applying any forces, ensuring the ship behaves naturally in all conditions.

This dynamic and physics-based approach enables smooth and realistic movement and turning for the vessel, creating an immersive simulation of real-world maritime operations.

5.3 Electronic Systems

We now proceed to the implementation of the electronic systems within the application, which represents the second major objective of this thesis. A significant focus was placed on developing and simulating real-world technologies that a remote operator would typically interact with.

5.3.1 Radar

The radar system in this project was implemented to provide real-time detection of objects around the vessel, simulating how actual radar systems work. The radar continuously scans the environment, detects nearby objects, mainly ships, and updates the user with relevant information. More specifically it generates pings on a UI indicating the position of the obstacle it detected. The script manages the radar's rotation, range, and object detection, while also providing visual feedback through a radar interface.

Radar Sweep and Rotation: The radar system rotates at a defined speed, simulating the sweep motion of a real radar. The angle of rotation is calculated based on time and a specified rotationSpeed, allowing the radar to continuously scan the environment. The rotation is visualized using a sweep graphic, which updates every frame to reflect the radar's current orientation.

Object Detection: Using Unity's RaycastAll function, the radar system sends out raycasts in the direction of the radar sweep. If the raycast hits an object within the radar's defined range, the system registers the object's position and adds it to a list of detected colliders. This list is reset after each half-sweep to ensure that only current objects are detected.

Visual Feedback: For each detected object, a radar ping is generated on the radar's UI. The position of each object relative to the ship is calculated and transformed from 3D space into 2D coordinates for display on the radar interface. Objects are represented as icons (pings) on the radar screen, with the color and size of the pings adjusted based on proximity to the vessel.

User Control: The radar's range and rotation speed can be adjusted dynamically by the user. Increasing or decreasing the radar's range allows the operator to scan a wider or narrower area, while adjusting the rotation speed controls how quickly the radar completes a full sweep.

Audio Cues: Audio feedback is provided to signal the detection of objects, with different sounds triggered depending on how close an object is to the vessel. This adds an immersive layer to the radar system, alerting the user when an object is detected nearby.

```
private void DetectObjects()
{
    RaycastHit[] hits = Physics.RaycastAll(ship.transform.position, finalRotation, radarRange, radarLayerMask);

    if (hits.Length > 0) //if we hit an object
    {
        foreach (RaycastHit hit in hits)  //for the case of hitting two objects in the same direction
        {
            if (hit.collider != null)
            {
                if (!colliders.Contains(hit.collider))
                {
                    //hit this object for the first time
                    colliders.Add(hit.collider);

                    // If the ray hits something, you can get information about the hit object here
                    GameObject hitObject = hit.collider.gameObject;

                    distance = GetDistanceToShip(hitObject.transform.position);

                    radarSize = GetRadarUISize();

                    var scale = radarSize / radarRange;

                    distance *= scale;

                    // Get the forward vector of the ship projected on the xz plane(direction)
                    var shipForwardDirectionXZ = Vector3.ProjectOnPlane(ship.transform.forward, Vector3.up);

                    // Create a roation from the direction. Create a rotation that points in the forward direction provided.
                    var rotation = Quaternion.LookRotation(shipForwardDirectionXZ);

                    // Mirror y rotation
                    var euler = rotation.eulerAngles;
                    euler.y = -euler.y;
                    rotation.eulerAngles = euler;

                    // Rotate the icon location in 3D space
                    var rotatedIconLocation = rotation * new Vector3(distance.x, 0.0f, distance.y);

                    // Convert from 3D to 2D
                    distance = new Vector2(rotatedIconLocation.x, rotatedIconLocation.z);

                    RadarPing ping = Instantiate(pfRadarPing).GetComponent<RadarPing>();
                    ping.transform.SetParent(iconContainer.transform, false);
                    ping.GetComponent<RectTransform>().anchoredPosition = distance;
                    //sound of ping


                    ping.SetColor(new Color(1, 0, 0));


                    ping.SetDisappearTimer(360f / rotationSpeed);

                    if(distance.magnitude < 50)
                    {
                        FindObjectOfType<AudioManager>().Play("RadarPingClose");

                    } else
                    {
                      FindObjectOfType<AudioManager>().Play("RadarPingFar");

                    }

                }
```

Figure 5.11: DetectObjects()

5.3.2 Gyroscope

The gyroscope system in this project was implemented to provide real-time orientation feedback to the operator, simulating the functionality of a physical compass combined with a gyroscope. This system calculates the vessel's heading and displays the direction, along with the current angle, to help with navigation

Compass Image Rotation: The compass image is updated continuously to reflect the vessel's heading in real-time. As the ship rotates, the system calculates the ship's local y angle and adjusts the compass image's UV coordinates to simulate the rotation of the compass. This provides a visual representation of the ship's orientation relative to the

cardinal directions (North, East, South, and West).

Heading Calculation: The forward vector of the vessel is used to calculate its heading direction. By zeroing out the y component, the system ensures that only the X and Z plane is considered when determining the compass direction. The resulting angle is then clamped to 5-degree increments, ensuring smoother transitions between displayed angles.

Display of Compass Degrees: The calculated heading angle is rounded to the nearest 5 degrees and displayed to the user. Depending on the value of the heading angle, the system determines whether to display a specific direction (e.g., N for North, E for East, etc.) or the precise degree value. A switch statement handles this, converting specific angles into cardinal directions (N, NE, E, etc.) while displaying numerical degrees for all other angles.

Real-Time Updates: The compass-gyroscope system runs continuously, updating both the compass image and the degree text in real-time as the ship moves. This ensures that the user always has an accurate sense of the ship's current heading, allowing for precise navigation.

```
private void Update()
{
    //Get a handle on the Image's uvRect
    compassImage.uvRect = new Rect(ship.localEulerAngles.y/360,0,1,1);

    // Get a copy of your forward vector
    forward = ship.transform.forward;


    // Zero out the y component of your forward vector to only get the direction in the X,Z plane
    forward.y = 0;

    //Clamp our angles to only 5 degree increments
    headingAngle = Quaternion.LookRotation(forward).eulerAngles.y;
    headingAngle = 5 * (Mathf.RoundToInt(headingAngle / 5.0f));

    //Convert float to int for switch

    displayAngle = Mathf.RoundToInt(headingAngle);

    //Set the text of Compass Degree Text to the clamped value, but change it to the letter if it is a True direction
    switch (displayAngle)
    {
        case 0:
            //Do this
            compassDegreesText.text = "N";
            break;
        case 360:
            //Do this
            compassDegreesText.text = "N";
            break;
        case 45:
            //Do this
            compassDegreesText.text = "NE";
            break;
        case 90:
            //Do this
            compassDegreesText.text = "E";
            break;
        case 130:
            //Do this
            compassDegreesText.text = "SE";
            break;
        case 180:
            //Do this
            compassDegreesText.text = "S";
            break;
        case 225:
            //Do this
            compassDegreesText.text = "SW";
            break;
        case 270:
            //Do this
            compassDegreesText.text = "W";
            break;
        default:
            compassDegreesText.text = headingAngle.ToString();
            break;
    }

}
```

Figure 5.12: Compass Script

5.3.3 Bathymetric Lidar

The LiDAR simulation system in this project replicates the functionality of a real-world LiDAR sensor, which is used to scan the environment and gather spatial data. More specifically ours is detecting underwater obstacles. This system emits rays in a spherical pattern to detect objects within a given range, generating a detailed 3D map of the surroundings. The following explains how the LiDAR simulation was implemented.

Raycasting for Environmental Scanning: The system performs multiple raycasts from the vessel's position in random directions within a defined spherical area. For each raycast, if an object is detected within the LiDAR's range, the hit point is recorded and added to a list of detected positions. These positions represent the 3D coordinates of objects in the environment, simulating how real LiDAR captures the shape and distance of surrounding objects.

Visualization of Detected Points: Each detected point is visualized using a LineRenderer, which displays the ray from the LiDAR's origin to the detected object. This provides real-time visual feedback to the user, showing the direction and distance of each detected object in the environment. The line color varies based on whether the ray hits an object (green) or misses (red), helping to distinguish between successful and unsuccessful scans.

Handling of Particle Data: The detected points are stored in a Texture2D object, which acts as a data map for the positions of objects in the environment. The texture is then passed to a Visual Effect (VFX) system, which is responsible for displaying the LiDAR particles in a visual simulation. Each detected point is translated into a particle in the VFX system, forming a dynamic representation of the environment as the LiDAR scans.

Dynamic VFX Creation: To ensure smooth operation and avoid overloading the system, the LiDAR simulation creates new VFX objects only when necessary. The system has a built-in limit on the number of VFX objects that can be active at once. If the limit is reached, the oldest VFX is destroyed and replaced by a new one. This ensures that the simulation can continue without performance issues.

Adjustable Parameters: The simulation allows for several adjustable parameters, such as the radius of the scan, the number of points per scan, and the range of the LiDAR. This flexibility makes the system adaptable to different scanning scenarios, providing a balance between performance and accuracy.

Application of Results: After each scan, the detected points are applied to the VFX system, updating the visual simulation of the environment. The LiDAR continues scanning and updating in real-time, providing a continuous stream of data that reflects the changing environment as the vessel moves.

```
    private void Scan()
    {


        for (int i = 0; i < _pointsPerScan; i++)
        {
            // generate random point
            Vector3 randomPoint = Random.insideUnitSphere * _radius;
            randomPoint += _castPoint.position;

            // calculate direction to random point
            Vector3 dir = (randomPoint - transform.position).normalized;

            // cast ray
            if (Physics.Raycast(transform.position, dir, out RaycastHit hit, _range, _layerMask))
            {
                Debug.DrawRay(transform.position, dir * hit.distance, Color.green);
                // only add point if the particle count limit is not reached
                if (_positionsList.Count < resolution * resolution)
                {
                    //if (hit.collider.CompareTag(REJECT_LAYER_NAME)) continue;
                    _positionsList.Add(hit.point);
                    _lineRenderer.enabled = true;
                    _lineRenderer.SetPositions(new[]
                    {
                            transform.position,
                            hit.point
                        });



                //_particleAmount++;
                //_currentVFX.SetInt(PARTICLE_AMOUNT_PARAMETER_NAME, _particleAmount);
            }
            // create new VFX if the particle count limit is reached
            else
                {
                    _createNewVFX = true;
                    CreateNewVisualEffect();
                    break;
                }
            } // raycast
            else
            {
                Debug.DrawRay(transform.position, dir * _range, Color.red);
            }
        } // for loop
        ApplyPositions();

        _lineRenderer.enabled = false;

    }
}
```

Figure 5.13: Scan()

5.3.4 Rendering Cameras

In the control room of the vessel, the screens displaying security feeds and GPS data are rendered using a dynamic texture system. This system ensures that the visual data from various cameras, including the security cameras and GPS, are properly displayed on their respective screens without unnecessary performance overhead.

Rendering to a Texture: The visual data from the security cameras and GPS systems are first rendered onto a Render Texture. This texture acts as a live feed of the camera's

view, capturing what the camera sees in real-time. Instead of rendering the camera's view directly to the screen, it is first captured in this texture.

Applying the Texture to a Material: The Render Texture is then applied to a material that is mapped to the corresponding screens in the control room. This ensures that the camera's output is displayed on the correct screen, whether it's a security feed or the GPS interface.

Displaying the Material on the Screen: Once the material is created with the updated texture, it is applied to the in-game screen objects in the control room. This allows the camera's perspective to be projected onto the control screens, giving the user a real-time view of what the cameras or GPS systems are capturing.

Optimizing Frame Rendering: To enhance performance and avoid rendering unnecessary frames, I implemented a script that controls the frequency of the camera rendering. The RenderTextureUpdater script ensures that the camera only renders every few frames (based on a configurable interval). This approach helps save valuable frames per second (FPS) while maintaining a smooth user experience, as the slower update rate is not noticeable to the user in practice.

The script works by enabling the camera only every few frames, controlled by the frameInterval variable. By skipping the rendering of unnecessary frames, it prevents excessive strain on the system's resources, allowing the game to run more efficiently.

5.3.5 Autopilot

The autopilot system implemented in this project is designed to guide the vessel autonomously using Unity's NavMesh package. This system calculates a path for the ship and moves it through various waypoints by controlling its steering and acceleration based on the calculated path. The key difference from typical NavMesh implementations is that instead of using NavMeshAgent.SetDestination, the script manually follows the calculated path and controls the ship's movement through custom steering and propulsion functions.

NavMesh Agent and Surface: The NavMeshAgent represents the ship in the autopilot system, and it moves along a path defined by the NavMesh Surface, which is essentially a map of the navigable water surface for the agent. The bake procedure is used to generate this surface, allowing the ship to determine valid paths across the environment.

Path Calculation: The script calculates a path from the ship's current position to a destination using the NavMeshAgent.CalculatePath() function. This path is represented as a series of waypoints (called corners) that guide the ship from its current location to the target. If the destination or path is invalid, an event is triggered to handle the error, ensuring the system doesn't fail unexpectedly.

Custom Movement: Instead of allowing the NavMeshAgent to move the ship directly, the system retrieves the path and processes it manually. The ship's steering and accelera-

tion are controlled through custom functions (Turn and Accelerate) based on the direction and distance to the next waypoint in the path.

- Steering: The ship calculates the angle between its current direction and the next waypoint, then adjusts its direction by turning either left or right using the Turn() function from the PropulsionSystem script. This function determines the amount of steering needed and applies it dynamically until the ship faces the waypoint.
- Acceleration: Once the ship is aligned with the next waypoint, it accelerates toward it using the Accelerate() function, propelling itself along the calculated path. This process is repeated for each waypoint until the ship reaches its destination.

Waypoint Handling: The ship navigates from waypoint to waypoint by checking the distance to each corner of the path. When it gets close to a waypoint (within a certain threshold), the script moves on to the next waypoint, adjusting the steering and acceleration accordingly. This ensures a smooth transition from one point to another along the path.

Optimized Performance: The system enables and disables the NavMeshAgent appropriately, ensuring it is only active when necessary. Additionally, visual debugging is available through the DrawPath() function, which renders the calculated path in the scene view to help visualize the ship's route.

```
void CalculatePathToDestination()
{
    // Find the destination waypoint
    GameObject waypoint = GameObject.Find("waypointForNavMesh");
    if (waypoint != null)
    {
        destination = waypoint.transform;
    }

    // Initialize the path
    path = new NavMeshPath();

    // Calculate the path to the destination
    if (destination != null)
    {
        bool pathFound = navAgent.CalculatePath(destination.position, path);
        if (pathFound)
        {
            Debug.Log("Path found!");
            // Draw the path for debugging
            DrawPath(path);
        }
        else
        {
            Debug.LogError("No path found to the destination.");
            OnPathNotFound?.Invoke(); //Fire an event to handle when we dont have a path to the destination
        }
    }
    else
    {
        Debug.LogError("Destination is null!");
    }
}
```

Figure 5.14: CalculatePathToDestination()

```
void MoveAlongPath()
{

    //Move towards the current waypoint
    if (currentWaypointIndex +1< path.corners.Length) //+1 because the first waypoint is on our ship
        {

            Vector3 targetPosition = path.corners[currentWaypointIndex+1];
            Vector3 directionToWaypoint = (targetPosition - transform.position).normalized;

            //Calculate the angle between the ship's forward direction and the direction to the waypoint
            float angleToWaypoint = Vector3.SignedAngle(-transform.right, directionToWaypoint, Vector3.up);

            //Determine the turn direction: 1 for right, -1 for left
            float turnModifier = angleToWaypoint > 0 ? 1f : -1f;

        float currentValue = angleToWaypoint;


        if (stValue) //only for the initialisation of previousValue
        {
            previousValue = angleToWaypoint > 0 ? 1f : -1f;
            stValue = false;
        }

        if (!((previousValue >= 0 && currentValue < 0) || (previousValue < 0 && currentValue >= 0)) && !turnEnded) //turn towards the waypoint
        {
            //Debug.Log("Turn");
            previousValue = currentValue;
            propulsionSystem.Turn(turnModifier, 0.15f);
            }
            else //If the ship is facing the waypoint, accelerate
            {
            turnEnded = true;
            //Debug.Log("Accelerate");

            propulsionSystem.Accelerate(1, 2234216f); //Adjust the horsepower as needed
            }


        // If close to the waypoint, move to the next one
        if (Vector3.Distance(navAgent.transform.position, targetPosition) < 50f)
            {
            Debug.Log("reached waypoint");

            currentWaypointIndex++;
            turnEnded = false;
            stValue = true;
            }
        }
        else
        {
            Debug.Log("Reached the destination.");

        }
    }
```

Figure 5.15: MoveAlongPath()

5.4 Vr Integration in Unity

In this project, the integration of Virtual Reality (VR) plays a crucial role in creating an immersive and interactive experience. The process involved setting up the VR environment within Unity by installing and configuring several key packages. The first step was installing Unity's XR Interaction Toolkit and OpenXR Plugin, which provide the foundational tools for VR functionality, such as head tracking, input controls, and rendering. Additionally, the Meta Quest SDK was integrated to ensure compatibility with the Meta Quest Pro headset, which was used for this project.

With these packages installed, the next step was configuring the XR Rig, which serves as the user's presence in the virtual environment. The rig includes a camera that tracks the user's head movements, ensuring that their view adjusts according to their real-world motions. The rig was further customized to incorporate hand-tracking controls instead of traditional VR controllers, allowing for a more natural and intuitive interaction with the virtual space.

The VR camera was then carefully configured to provide the correct field of view and smooth head tracking, ensuring that the user experiences an immersive perspective when navigating through the virtual environment. This setup allows the user to look around freely and interact with objects using hand gestures, enhancing the realism and engagement of the simulation.

By following these steps, the VR integration in Unity was successfully established, creating a seamless bridge between the user's physical movements and their virtual interactions. In the subsequent sections, we will explore in detail the camera setup and hand-based controls that make this experience possible.

5.4.1 Camera

The first step in creating the virtual environment was to configure the rendering camera for VR. With the XR Interaction Toolkit installed, Unity provides the option to create an object called XR Origin, which represents the player within the virtual world when the application is running.

Within the XR Origin, there is a camera that is specifically set up for stereoscopic rendering, allowing the VR experience to be immersive and realistic. This camera is essential for achieving the depth perception that makes VR environments feel three-dimensional. Additionally, the XR Origin includes objects for the controllers, or in this case, hand tracking, to allow natural interaction with the environment.

The XR Origin's camera is closely integrated with the headset (HMD) tracking system. This setup ensures that the camera mirrors the user's head movements in real-time, allowing the virtual environment to move fluidly in sync with the user. As the user navigates through the virtual space, the camera updates instantly, creating a seamless and immersive experience that responds accurately to the user's physical movements. This real-time tracking enhances the sense of presence within the virtual environment, making the VR simulation more engaging and realistic.

5.4.2 Controls

The interaction system in the VR environment was implemented using XR Hands, enabling natural and intuitive hand-based controls. By integrating the XR Hands package, we replaced traditional controllers with virtual hands, allowing users to interact directly with the virtual world through their hand movements.

With XR Hands, we captured the precise position and orientation of the user's hands, which are rendered in real-time within the VR environment. These hands are used to perform actions such as grabbing, pushing, or manipulating objects. The hand tracking is mapped to various gestures, like pinching or pointing, which correspond to specific interactions within the virtual space.

The XR Hands system detects these movements through the headset's sensors and translates them into real-time inputs. For instance, users can reach out and "grab" virtual objects by bringing their fingers together, simulating the gesture of holding something. This provides a highly immersive experience by eliminating the need for external controllers, allowing users to interact with the environment as they would in real life.

The integration of XR Hands created a fluid and natural user experience, enabling direct, gesture-based interaction with objects and interfaces, enhancing the realism and engagement of the virtual environment.

5.4.3 Interactions

5.4.4 UI Interaction

In the VR environment, interacting with the User Interface (UI) was a crucial aspect of creating an absorbing and functional experience. With the integration of XR Hands, the traditional method of using controllers for UI interaction was replaced by hand gestures, allowing users to naturally engage with the interface.

To enable VR interactions with the UI, the XR Interaction Toolkit was configured to recognize hand gestures and map them to actions within the UI elements. For example, pointing at a button or icon with a finger allows users to highlight the UI component, while a pinching gesture or hand tap can be used to select or activate that element. The UI system is designed to detect proximity between the virtual hands and the UI elements, making it responsive to natural hand movements.

The UI elements themselves were modified to be VR-compatible, ensuring that buttons, sliders, and other interactive components are placed in 3D space and can be interacted with from different angles. This setup allows the user to reach out and interact with menus, controls, and settings as if they were physically present in the virtual world. The Tracked Device Graphic Raycaster script from the XR interaction toolkit is applied.

Additionally, hover effects, visual feedback, and sounds were added to the UI to provide clear interaction cues. These features enhance the user experience by providing real-time feedback when an item is being selected or manipulated, ensuring that interactions feel responsive and intuitive in the VR setting.

By leveraging XR Hands for hand-based interactions, the UI in the VR environment became a seamless extension of the user's natural motions, making navigation and control feel fluid and captivating.

5.4.5 3D Interaction

To facilitate natural and immersive 3D interactions within the VR environment, I utilized several pre-built scripts from the XR Interaction Toolkit, which greatly simplified the

process of integrating hand-based controls. These scripts provided out-of-the-box functionality for common interactive elements such as buttons, sliders, and levers, all of which were operated using XR Hands.

For instance, the XR Grip Button was used for buttons that required gripping motions, while the XR Knob was implemented for turning controls like dials and the steering wheel. The XR Slider allowed for sliding interactions, enabling smooth movement across a range. Additionally, the XR Poke Filter and XR Simple Interactable were employed for buttons that required poking or tapping gestures, making those interactions intuitive and responsive. The XR Lever was specifically used for the thrust lever, simulating real-world throttle controls by allowing users to move the lever up or down with a hand gesture.

These scripts provided a high level of flexibility and responsiveness, allowing the hand-tracking system to accurately detect and respond to user inputs. This significantly streamlined the development of hand-based interactions, enabling precise and realistic manipulation of objects and controls within the virtual environment. By leveraging these tools, I was able to create a fluid and absorbing interaction system that made use of natural hand movements, enhancing the overall user experience in VR.

5.4.6 Buttons - Lever - Wheel

In this section, I created three custom scripts—WheelRotator, ThrustLever, and Button3D—to integrate 3D models with the XR interaction system for realistic and intuitive user interactions within the VR environment. These scripts work in conjunction with XR Knob, XR Lever, and XRSimpleInteractable components, allowing the virtual hands to manipulate the ship's controls accurately. Each script processes inputs from the corresponding XR component and translates them into real-time control actions for the ship's propulsion, steering, and systems.

WheelRotator Script The WheelRotator script manages the functionality of the ship's steering wheel using the XR Knob to detect the current wheel rotation. The script continuously reads the XR Knob's value—a normalized value from 0 to 1—and maps it to a specific range of wheel rotations between -145° and 145°. This mapped value is then used to calculate the desired heading for the ship, ranging from 135° to 225°.

The script also calculates the amount of torque required to steer the ship based on how far the wheel is turned from its neutral position (180°). If the wheel is turned significantly, the PropulsionSystem's Turn() function is triggered, adjusting the ship's direction accordingly. The rotation speed is determined by how far the wheel is turned, with the turning force gradually increasing as the wheel moves closer to its maximum values. This allows for smooth and precise control over the ship's steering, simulating a realistic navigation experience.

```
void Update()
{
    //Get the current rotation from the XR Knob script
    float currentWheelValue = xrKnobScript.value;


currentRotation = MapValue(currentWheelValue, 0, 1, wheelMinRotation, wheelMaxRotation);


    //Clamp the current rotation to the allowed range
    //currentRotation = Mathf.Clamp(currentRotation, wheelMinRotation, wheelMaxRotation);

    //Map the current rotation to the ship's rotation range
    shipWantedDeg = MapValue(currentRotation, wheelMinRotation, wheelMaxRotation, shipMinRotation, shipMaxRotation);

    //Calculate the steering based on the desired ship angle
    CalculateSteering();

    //Apply propulsion system turn logic if needed
    if (shipWantedDeg < 175f || shipWantedDeg > 185f)
    {
        propulsionSystem.Turn(modifier, shipSteeringTorque);
    }
}


private void CalculateSteering()
{
    float middlePoint = (shipMinRotation + shipMaxRotation) / 2;

    if (shipWantedDeg < middlePoint)
    {
        modifier = thrustLever.modifier == -1 ? 1 : -1; // turn left
        shipSteeringTorque = MapValue(shipWantedDeg, shipMinRotation, middlePoint, 0.25f, 0f); // map to find how fast we turn
    }
    else if (shipWantedDeg > middlePoint)
    {
        modifier = thrustLever.modifier == -1 ? -1 : 1; // turn right
        shipSteeringTorque = MapValue(shipWantedDeg, middlePoint, shipMaxRotation, 0f, 0.25f); // map to find how fast we turn
    }
    else
    {
        modifier = 0; // we don't turn
    }
}


private float MapValue(float value, float oldMin, float oldMax, float newMin, float newMax)
{
    return ((value - oldMin) / (oldMax - oldMin)) * (newMax - newMin) + newMin;
}
```

Figure 5.16: WheelRotator script

ThrustLever Script

The ThrustLever script controls the ship's thrust level by using the XR Lever to detect the lever's current position. The lever's rotation angle is mapped to a range of engine thrust values, from 0 to 4,468,432 Newtons, representing the real force exerted our , Maersk Honam, vessel's engine. As the lever moves, the script continuously updates the target thrust and smoothly adjusts the engine's power output to match, simulating gradual acceleration or deceleration.

To provide additional realism, the script also handles audio feedback by adjusting engine sounds depending on the thrust level. Low engine power triggers a quiet hum, while high thrust levels increase the volume and intensity of the sound. The ThrustLever script also updates the control panel with real-time data, displaying the ship's speed in knots, RPMs, and the engine's status. This script integrates closely with the PropulsionSystem to ensure smooth acceleration and deceleration while maintaining realistic feedback for

the user.

```
void FixedUpdate()
{
    // Get the current angle from the XRLever component
    float currentLeverAngle = leverStick.localEulerAngles.x;

    if (currentLeverAngle < 360f && currentLeverAngle > 279f)
        currentLeverAngle = leverStick.localEulerAngles.x - 360;


    // Map the lever angle to the target newtons value
    targetNewtons = MapValue(-currentLeverAngle, minRotation, maxRotation, minEngineNewtons, maxEngineNewtons);


    // Smoothly interpolate the current newtons towards the target newtons
    currentNewtons = Mathf.MoveTowards(currentNewtons, targetNewtons, accelerationRate * Time.deltaTime);




    if (currentNewtons > 0)
    {
        propulsionSystem.Accelerate(currentNewtons, modifier);
    }
```

Figure 5.17: Update() of Thruster script

Button3D Script The Button3D script is used for managing the various 3D buttons found on the ship's control panel. It utilizes the XRSimpleInteractable component to detect hand-based interactions when a button is pressed. Each button is assigned a specific function, such as toggling the engine, controlling the anchor, or managing the ship's lights. When the user presses a button, the script checks the button's tag to determine the associated action.

The Button3D script also includes sound effects for button presses and releases, as well as other ship systems like the horn or engine. It handles button state changes and ensures that each press corresponds to a specific ship function, allowing the user to interact with the ship's systems in a realistic and tactile way.

```
void OnButtonPressed(string buttonTag)
{
    FindObjectOfType<AudioManager>().Play("ButtonPress");
    switch (buttonTag)
    {
        case "Button1":

            if(!engine.GetComponent<PropulsionSystem>().engineOn)
            {
                FindObjectOfType<AudioManager>().Play("EngineOn");
            }else
            {

                FindObjectOfType<AudioManager>().Stop("EngineOn");
                FindObjectOfType<AudioManager>().Play("EngineOff");
            }
            engine.GetComponent<PropulsionSystem>().engineOn = !engine.GetComponent<PropulsionSystem>().engineOn;
            break;
        case "Button2": //Button for going Astern
            thrustLever.GetComponent<ThrustLever>().modifier *= -1;
            break;
        case "Button3": //Button for dropping and elevating anchors
            anchor.GetComponent<AnchorManager>().ToggleAnchor();
            FindObjectOfType<AudioManager>().Play("Anchor");
            break;
        case "Button4": //Button for dropping and elevating anchors
            mastHeadLight.gameObject.SetActive(!mastHeadLight.gameObject.activeSelf);
            break;
        case "Button5": //Button for lights
            if (pls.on) { //If the power is on
                sternLight1.gameObject.SetActive(!sternLight1.gameObject.activeSelf);
                sternLight2.gameObject.SetActive(!sternLight2.gameObject.activeSelf);
            }
            break;
        case "Button6": //Button for lights
            if (pls.on)//If the power is on
            { leftSideLight.gameObject.SetActive(!leftSideLight.gameObject.activeSelf); }
            break;
        case "Button7": //Button for lights
            if (pls.on)//If the power is on
            {
                aftMastHeadL1.gameObject.SetActive(!aftMastHeadL1.gameObject.activeSelf);
                aftMastHeadL2.gameObject.SetActive(!aftMastHeadL2.gameObject.activeSelf);
                aftMastHeadL3.gameObject.SetActive(!aftMastHeadL3.gameObject.activeSelf);
                aftMastHeadL4.gameObject.SetActive(!aftMastHeadL4.gameObject.activeSelf);
            }
            break;
        case "Button8": //Button for lights
            if (pls.on)//If the power is on
            {
                rightSideLight.gameObject.SetActive(!rightSideLight.gameObject.activeSelf);
            }
            break;
        default:
            Debug.Log("Unknown Button Pressed!");
            break;
    }
}
```

Figure 5.18: OnButtonPressed()

5.4.7 Vr Hand Gestures

To implement teleportation in the control room using hand gestures, we leveraged Unity's XR Interaction Toolkit and XR Hands. We began by enabling hand tracking, ensuring that both the XR Interaction Rig and the necessary packages for gesture detection were correctly set up. This rig supports both controllers and hand gestures, but for our purpose, we focused on hand-based teleportation.

We first ensured the basic functionality of teleportation using a ray-based interaction method. The teleportation system initially worked with controllers, where pressing the

touchpad would activate the teleportation ray, allowing the user to move to the designated area. However, we wanted to extend this functionality to hand gestures, making the interaction more immersive. This was achieved by defining a custom hand pose using the gesture detection features provided by the XR Hands package.

The custom hand gesture for teleportation involved pointing with the palm facing upward. This was achieved by creating a specific hand pose that recognized the correct curl of the little and middle fingers while ensuring the palm faced upward. Once this gesture was detected, the teleportation ray was activated, allowing the user to aim and teleport to the designated location. When the user performed the teleportation gesture, the system would enable the teleport interactor, allowing movement around the control room.

Additionally, we customized the visual feedback of the teleportation ray. The ray only appeared when the hand gesture was active, preventing unwanted visual clutter during regular interactions. The color of the ray was adjusted to provide clear feedback, turning blue when it was ready to interact with a teleportable surface and becoming invisible when no valid surface was detected.

To ensure that only one interaction (either the teleportation ray or the regular ray used for object interaction) was active at a time, we utilized the XR Interaction Group. This prevented the teleportation system from conflicting with other hand-based interactions, ensuring a seamless experience. The interaction group prioritized the teleportation ray over other rays, ensuring that when the teleport gesture was detected, the user could focus purely on movement.

Overall, this implementation allows the user to teleport within the control room using natural hand gestures, without needing controllers, enhancing both immersion and ease of use in the VR environment. By combining gesture recognition, visual feedback customization, and interaction prioritization, the user can navigate the space fluidly, making the VR experience both intuitive and immersive.

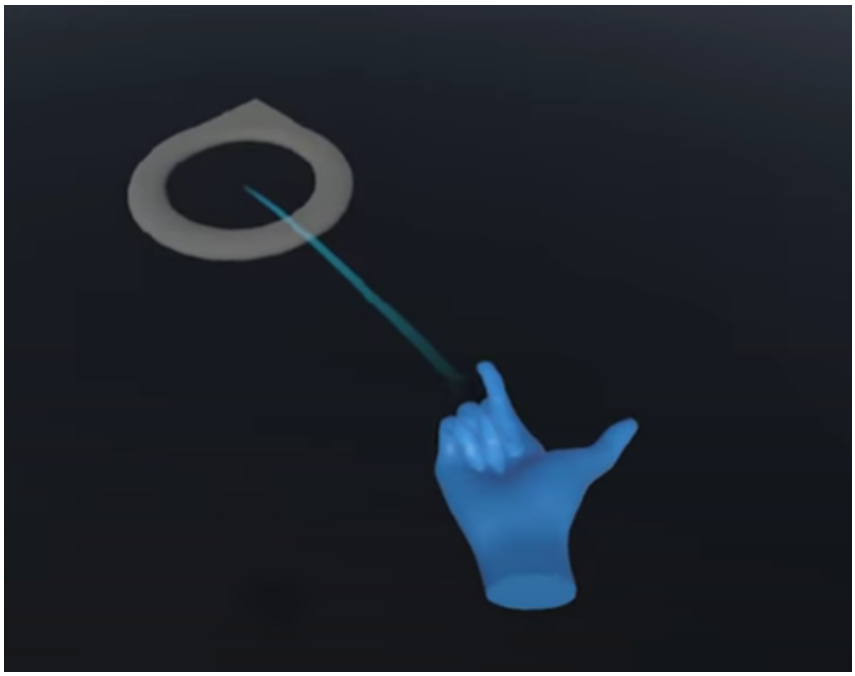

Figure 5.19: Teleportation Hand Gesture

5.5 Manager Scripts

In the application, several manager scripts were implemented using the singleton design pattern. The singleton class ensures that only one instance of the manager exists and provides a global access point for other scripts to retrieve that instance. This allows for centralized control and coordination of various systems within the application, ensuring consistency and ease of access across different components.

5.5.1 Game Manager

The GameManager script is a central component that handles the overall flow of the game and connects user preferences from the main menu to the main scene. Using the singleton pattern, it ensures there is only one instance of the GameManager, allowing other scripts to easily access and modify game states globally.

One of its primary functions is to apply the player's preferences, which were set in the main menu, to the main scene. These preferences include rain intensity, fog density, time of day, and skybox selection. Upon starting the game, the script reads these values from PlayerPrefs and adjusts the environment accordingly. For instance, it modifies the rain intensity across multiple rain systems, adjusts the fog density by setting the appropriate material properties, rotates the sun to simulate the selected time of day, and applies the selected skybox to change the atmosphere. Additionally, the game mode, such as Freeroam or Inform, is set based on the player's choice, dictating how the gameplay unfolds.

The GameManager is also responsible for managing the various game states, such as Freeroam, Inform, Load, Success, Crash, and Time Elapsed. Each state triggers specific actions or displays the appropriate UI panels to the user. For example, in the Freeroam state, the player can explore the scene freely, while in Inform, an informational panel is shown. The Load state presents a loading screen, and the Success state displays a panel with the timer once the player successfully completes an objective. In cases of failure, such as when the player crashes the ship or runs out of time, the game enters the Crash or Time Elapsed state, displaying the relevant UI panel and then restarting the scene after a short delay.

The script also listens for user actions, such as button presses or interactions, and updates the game state accordingly. For example, when the player finishes a task or crashes, the game state changes, and the appropriate UI panel is displayed. Depending on the situation, the game either restarts or progresses to the next state, ensuring a smooth transition through the game's phases.

```
void Awake()
{
    Instance = this;

    //Get ints from mainMenuController
    rainIntensity = PlayerPrefs.GetFloat("Rain");
    fogDensity = PlayerPrefs.GetFloat("Fog");
    sunRot = PlayerPrefs.GetFloat("Time");


    foreach (var rain in rains)
    {
        rain.RainIntensity = rainIntensity;
    }

    //Set the fog
    fog.material.SetFloat("FogDensity", fogDensity);

    //Set the time
    sun.transform.rotation = Quaternion.Euler(sunRot, 0f, 0f);


    //Set the skybox
    i = PlayerPrefs.GetInt("SelectedSkyboxIndex");
    UnityEngine.RenderSettings.skybox = skyboxes[i];

    //Set the mode of the game

    gameMode = PlayerPrefs.GetInt("GameMode");

   activateOnStart.SetActive(true);



    //Depends on the mode
    if (gameMode == 0)
        UpdateGameState(GameState.Freeroam);
    if (gameMode == 1)
        UpdateGameState(GameState.Inform);
}
```

Figure 5.20: Awake()

```
public void UpdateGameState(GameState newState)
{
    State = newState;

    switch (newState)
    {
        case GameState.Freeroam:
            break;
        case GameState.Inform:
            ShowPanel(0);
            break;
        //case GameState.GoLoad:
        //   break;
        case GameState.Load:
            ShowPanel(1);
            break;
        //case GameState.Return:
        //    break;
        case GameState.Sucess:
            ShowPanel(2);
            timerTextNeeded.text = timer.timerText.ToString();

            break;
        case GameState.Crash:
        case GameState.TimeElapsed:
            ShowPanel(newState == GameState.Crash ? 3 : 4);
            Restart(); // Restart after showing panel
            break;
        default:
            throw new System.ArgumentOutOfRangeException(nameof(newState),newState , null);
    }

    OnGameStateChanged?.Invoke(newState);
}
```

Figure 5.21: UpdateGameState()

5.5.2 Sound Manager

The AudioManager script is responsible for managing the audio in the application, using the singleton pattern to ensure there is only one instance of the AudioManager throughout the game. In the Awake() method, the script checks if an instance of the AudioManager already exists. If none exists, the current instance (this) is assigned as the main instance. If an instance already exists, the new one is destroyed to avoid having multiple audio managers. This ensures that the AudioManager is globally accessible and prevents conflicts from multiple instances.

Additionally, the DontDestroyOnLoad(gameObject) function is used to ensure that the AudioManager persists across scene changes, meaning that sounds and music continue to play even when switching between different scenes in the game. This is particularly useful for maintaining continuity in background music or persistent sound effects.

In the initialization process, the foreach loop iterates over an array of Sound objects, each representing a specific audio clip. For each sound, an AudioSource component is dynamically added to the AudioManager's gameObject, and properties such as volume, pitch, spatial blend, and looping behavior are set according to the parameters defined in each Sound object.

The script also includes Play() and Stop() methods for controlling audio. The Play(string name) method finds the desired sound by name in the sounds[] array and plays the associated audio clip by calling the Play() method of the sound's AudioSource. Similarly, the Stop(string name) method finds the sound by name and stops the audio by calling the Stop() method of the corresponding AudioSource.

```
public class AudioManager : MonoBehaviour
{
    public Sound[] sounds;

    public static AudioManager instance;

    private void Awake()
    {
        if (instance == null)
            instance = this;
        else
        {
            Destroy(gameObject);
        }

        DontDestroyOnLoad(gameObject);

        foreach (Sound s in sounds)
        {
            s.source = gameObject.AddComponent<AudioSource>();
            s.source.clip = s.clip;

            s.source.volume = s.volume;
            s.source.pitch = s.pitch;
            s.source.spatialBlend = s.spatialBlend;
            s.source.loop = s.loop;
        }
    }

        public void Play(string name)
        {
          Sound s = Array.Find(sounds, sound => sound.name == name);
          s.source.Play();
        }

    public void Stop(string name)
    {
        Sound s = Array.Find(sounds, sound => sound.name == name);
        s.source.Stop();

    }


}
```

Figure 5.22: Audio Manager Script

5.5.3 Menu Manager

The Menu Manager script is responsible for handling all the functionalities related to the in-game menus. It contains functions for every interactive element, such as buttons, sliders, toggles, or checkboxes, allowing the user to modify settings and preferences during gameplay. This script manages a wide range of interactions, from simple toggles to more complex sliders that alter the environment and gameplay experience.

For instance, the day-night slider adjusts the time of day in the game. By moving the slider, the user can change the scenery from a bright day to a dark night, directly affecting the lighting and atmosphere. Similarly, the rain slider enables the user to modify the intensity of the rain, ranging from light drizzle to heavy downpour, dynamically adjusting the environmental conditions based on the player's preference. Other examples include toggling environmental effects like fog density or switching between different skyboxes.

In addition to controlling these settings, the Menu Manager also ensures that the user's preferences are saved. Each time the player interacts with a menu element, the script stores the corresponding value using PlayerPrefs, so that these preferences are remembered when the game is restarted. For example, if the user sets the rain intensity to a specific level or chooses a particular skybox, these settings will persist across sessions, ensuring the player's customized experience is retained.

Overall, the Menu Manager script streamlines all user interactions with the game's menus, providing intuitive control over environmental settings and other gameplay parameters while ensuring that preferences are saved for future playthroughs.

```
void UpdateSunAndText(float value)
  {
      // Convert the slider value to represent hours of the day (0 to 24)
      float hoursOfDay = value / 15f; // Since your slider goes from 0 to 360

      // Calculate the rotation angle for the sun based on the hours of the day
      // Assuming 6:00 corresponds to 0 degrees rotation
      float sunRotation = (hoursOfDay - 6f) * 15f; // Each hour corresponds to 15 degrees rotation

      //for setting the sunRotation ingame
      PlayerPrefs.SetFloat("Time", sunRotation);

      // Update the rotation of the sun
      sun.transform.rotation = Quaternion.Euler(sunRotation, 0f, 0f);

      // Update the text to display the time
      int currentHour = Mathf.FloorToInt(hoursOfDay);
      int currentMinute = Mathf.FloorToInt((hoursOfDay - currentHour) * 60f);
      utcText.text = string.Format("{0:00}:{1:00} UTC", currentHour, currentMinute);
  }
```

Figure 5.23: A function in Menu Manager Script

5.6 Sounds

The Sounds segment in the application was implemented with a focus on both simple and complex audio management to enhance the immersive experience. Simple sounds, such as button clicks or engine start/stop effects, are handled using the AudioManager script. This script allows for centralized control of playing and stopping various sound effects triggered by user interactions or in-game events. The AudioManager uses the singleton pattern, ensuring that sound effects are managed consistently throughout the application.

For more complex sounds, such as the sound of waves and the sounds of the ship's engine, additional techniques were used to ensure smooth and realistic transitions. Instead of abruptly stopping and starting sounds when the intensity of the waves changes (e.g., from small ripples to large waves), the Mathf.Lerp function was employed. This function interpolates between two sound levels, allowing for gradual and natural mixing of audio clips. For instance, if the player moves the slider in the menu, from a calm sea to a stormy waves ocean, the Mathf.Lerp function smoothly transitions the sound from gentle waves to crashing waves, without any abrupt interruptions. This ensures that the wave sounds feel continuous and dynamic, adjusting seamlessly to changes in the environment.

The engine sounds were implemented with a focus on creating smooth and realistic audio transitions, ensuring that the sound of the engine accurately reflects the changes in power and speed during gameplay. For basic engine-related sounds, such as the engine starting or stopping, the AudioManager script was used to play and stop these sound effects as needed.

However, for more complex scenarios, such as changes in the engine's power output (e.g., from low RPMs to high RPMs), the Mathf.Lerp function was employed to smoothly transition between different engine sound intensities. Instead of abruptly stopping one sound and starting another, Mathf.Lerp allows the sound to gradually shift between levels, ensuring a continuous and realistic representation of engine power. As the player accelerates or decelerates, the sound smoothly transitions between low, medium, and high engine volumes and pitches, giving the impression of a dynamic, responsive engine.

```
private void UpdateEngineSounds(float leverValue)
{
    if (propulsionSystem.engineOn) //if the engine is on else no sound
    {

        // Smooth transition by adjusting volumes
        if (leverValue >= 27f)
        {
            engineLow.source.volume = Mathf.Lerp(engineLow.source.volume, 1f, Time.deltaTime);
            engineMedium.source.volume = Mathf.Lerp(engineMedium.source.volume, 0f, Time.deltaTime);
            engineHigh.source.volume = Mathf.Lerp(engineHigh.source.volume, 0f, Time.deltaTime);
        }
        else if (leverValue < 27f && leverValue >= -26f)
        {
            engineLow.source.volume = Mathf.Lerp(engineLow.source.volume, 0f, Time.deltaTime);
            engineMedium.source.volume = Mathf.Lerp(engineMedium.source.volume, 1f, Time.deltaTime);
            engineHigh.source.volume = Mathf.Lerp(engineHigh.source.volume, 0f, Time.deltaTime);
        }
        else if (leverValue < -26f)
        {
            engineLow.source.volume = Mathf.Lerp(engineLow.source.volume, 0f, Time.deltaTime);
            engineMedium.source.volume = Mathf.Lerp(engineMedium.source.volume, 0f, Time.deltaTime);
            engineHigh.source.volume = Mathf.Lerp(engineHigh.source.volume, 1f, Time.deltaTime);
        }
    }else //engine is off so no sound from propulsion
    {
        engineLow.source.volume = 0f;
        engineMedium.source.volume = 0f;
        engineHigh.source.volume = 0f;
    }

}
```

Figure 5.24: UpdateEngineSounds()

5.7 Thirt Party Assets

In this thesis, no 3D modeling software was used for creating original assets. Instead, all 3D models, skyboxes, and environmental elements were sourced from various online platforms, such as the Unity Asset Store and other free 3D model websites. For example, the control room, props, and objects used within the scene were downloaded and adapted from pre-made assets available online. The water system used was provided by Unity's built-in water system, and the rain effects were also sourced from an external creator. This approach allowed for the efficient creation of a detailed and functional environment without the need for custom modeling.

Chapter 6

Evaluation and Results

6.1 Evaluation - Results Analysis

Throughout the development process, the application was continuously tested by users, allowing for iterative improvements and refinements. The people participated in testing the simulator, which were around 15, provided valuable feedback that guided changes to enhance the user experience. The general response from testers was positive, particularly regarding the realism of the environment, which created a highly immersive experience. Users commented that the ship's controls and movement felt authentic, and the interaction system with hand gestures was intuitive after a brief learning period.

However, some negative feedback came from individuals who were unfamiliar with VR environments. These users reported a lack of clear instructions and faced challenges in controlling the ship initially. In response, adjustments were made, including fine-tuning the interaction sensitivity and improving the user interface to provide more guidance. Despite these early challenges, users quickly adapted to the controls and appreciated the accuracy and realism of the simulation.

As a result of user feedback, several tweaks were made to improve the overall experience. This included resizing certain elements for easier interaction and adjusting the responsiveness of the hand gestures. With these changes implemented, subsequent evaluations showed a marked improvement in usability, with fewer complaints and a more seamless navigation experience. The process of continuous testing and refinement ensured that the final application offered a polished and immersive training environment with only minimal defects.

6.2 Think Aloud Methodology

The Think Aloud Methodology is a user-centered testing approach where participants are encouraged to verbalize their thoughts, feelings, and decision-making processes while interacting with an application. This allows developers to gain insight into how users perceive the functionality and user interface, helping identify pain points and areas for improvement.

During the evaluation phase of our VR ship simulator, participants tested the appli-

cation using this methodology, providing real-time feedback on their experiences. Several negative points were noted. First, users highlighted imprecise interactions—they found it challenging to interact with certain UI's, UI buttons, particularly small elements, which made navigating some controls frustrating. Also some of the 3d levers and sliders was slightly diffuclt to interact with. This led to moments of confusion as users struggled to accurately select options. Additionally, those unfamiliar with virtual reality reported feeling motion sickness or nausea after extended use, a common issue for VR newcomers. The lack of clear instructions was also a recurring concern, with many first-time users expressing that they were unsure how to navigate or fully utilize the simulator's features, leading to initial frustration and a steeper learning curve.

On the positive side, users overwhelmingly praised the simulator for creating a realistic and immersive experience. The feeling of being aboard a control room of an uncrewed ship was convincing, with many commenting on how they felt completely submerged in the environment. The ship's handling and the detailed environment contributed to the authenticity, leading users to express excitement and a sense of accomplishment when completing tasks. They also found the application to be amusing and engaging, with several participants enjoying the unique concept of controlling a ship in a VR setting. The simulation sparked curiosity and interest, making users feel invested in the experience, even those who had no prior VR experience. The tactile interaction with the ship controls and the ability to explore the environment made the application feel like a unique and innovative approach to maritime training.

This feedback was invaluable in identifying areas where improvements were needed while also affirming the strengths of the application, particularly in its ability to immerse users in a realistic and engaging environment.

6.3 Number Of Restarts - Scenario Completion

The Number of Restarts - Scenario Completion is an evaluation approach that was used in my thesis measuring how often the application restarted and if the users did complete specific tasks or stages. In this method, users are observed during multiple sessions to track the number of restarts required when they encounter difficulties, as well as the total time taken to complete each task or goal.

In the context of the VR ship simulator, this method helped assess how well users adapted to the controls and gameplay mechanics over time. A higher number of restarts could indicate issues with clarity or difficulty in avoiding the obstacles while a shorter time to completion might suggest that users are becoming more comfortable with the controls and features as they progress. This evaluation method provided valuable insight into the learning curve of the application and allowed us to refine certain aspects to reduce user frustration and enhance the overall flow of the simulation.

By tracking both the restarts and completion features, the goal was to ensure that users could smoothly interact with the simulation and gradually improve their performance with each attempt, thereby demonstrating the simulator's effectiveness as a training tool.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we developed a physics-based virtual reality ship simulator designed to assist individuals in training for ship navigation and operations. The simulator includes realistic physics for ship movement, water dynamics, and environmental factors such as rain and fog, providing an immersive and safe environment for users to practice their maritime skills. Through the use of VR technology, this simulator offers a highly engaging and realistic training experience, allowing users to improve their ability to maneuver ships in a variety of challenging conditions without the risk of real-world consequences.

The simulator trains users to perform essential ship-handling tasks, such as navigating through rough waters, controlling speed and thrust, and adjusting to changing weather conditions. Evaluation and feedback from users showed that participants consistently improved their handling and performance over time. With repeated use, users were able to navigate more efficiently, improving their speed and accuracy in completing tasks such as docking or avoiding obstacles. This feedback highlights the effectiveness of the simulator in improving ship navigation skills.

While the simulator proved to be a success in helping users improve their performance, some participants, particularly those new to VR, experienced slight motion sickness. This is an area that requires further research to identify optimal settings or techniques that can reduce motion sickness and enhance comfort during extended use of the simulator.

In addition to the training experience, this thesis also explored how real-time environmental interactions, such as dynamic weather changes, influence the training and performance of ship operators. The results suggest that exposing users to these environmental conditions, such as heavy rain or strong winds, adds a layer of challenge that enhances their ability to adapt to real-world scenarios. This reinforces the value of the simulator as a comprehensive training tool that prepares users for a variety of real-world maritime challenges.

Overall, the physics-based VR ship simulator represents a significant advancement in the field of maritime training. It offers a versatile, safe, and immersive platform for improving human performance in ship navigation and operations. As VR technology continues to evolve, there is great potential for further enhancing this simulator to meet

the needs of both individual users and companies seeking to train ship operators in an increasingly complex maritime environment. This tool offers a promising solution for improving the effectiveness of maritime training programs, though further research is necessary to refine certain aspects, such as motion sickness reduction and the expansion of training features.

7.2 Future Work

While the VR ship simulator developed in this thesis provides a solid and immersive training platform, there are several areas where further research and development could significantly enhance its capabilities and expand its use cases. This application serves as a robust foundation that can be scaled and improved to offer even more advanced simulation experiences, particularly for maritime training. In the future, this simulator has the potential to become an indispensable tool for companies aiming to train ship operators in a variety of scenarios, providing a cost-effective and flexible alternative to traditional simulators. By building on the core functionality established here, it is possible to develop a far more sophisticated system that could serve both educational institutions and maritime corporations.

One key area for future development is the expansion of the simulator's functionalities to accommodate a wider variety of vessel types. Currently, the simulator is tailored to a specific type of ship, but by incorporating different classes of vessels, such as cargo ships, oil tankers, ferries, and military vessels, the simulator could become a versatile platform for training various professionals in the maritime industry. This would make the simulator more attractive to shipping companies, maritime schools, and military institutions that require specialized training for different types of vessels. Each vessel type has unique handling characteristics, propulsion systems, and navigation challenges, and simulating these would allow users to experience a broader range of maritime operations. For instance, future versions could simulate the complex handling of large container ships in narrow canals or oil tankers navigating hazardous waters, providing more diverse and challenging training scenarios.

Another significant area for improvement is the integration of real-time dynamic weather systems, which would significantly enhance the realism and variability of the training experience. Currently, the simulator offers basic environmental effects like rain and fog, but expanding this to include advanced weather systems would allow users to train under different and more complex maritime conditions. For instance, simulating real-time wind, storms, ocean currents, and wave heights could create high-stakes scenarios that better replicate the unpredictable conditions encountered at sea. For a more immersive experience, the simulator could integrate weather data from real-world sources, allowing users to train in environments that mirror actual maritime weather conditions.

Additionally, future iterations could model the effects of these environmental conditions on the vessel's stability, handling, and propulsion, making the training far more realistic. Accurate simulation of rough seas, crosswinds, and tidal effects would be particularly valuable for maritime pilots and captains who regularly deal with challenging weather conditions.

A particularly exciting future enhancement would be the development of a multiplayer mode that allows multiple users to interact in the same virtual environment. In a multiplayer setting, users could simulate complex maritime operations such as fleet movements, convoy navigation, search and rescue operations, or even competitive races between ships. This mode would foster teamwork and collaboration, which are essential skills in real-world maritime operations. Users could practice working together to solve problems in high-pressure situations, such as coordinating a rescue mission during a storm or managing the simultaneous navigation of several vessels in a busy port. Moreover, multiplayer interactions would introduce the element of unpredictability that comes from human opponents or teammates, providing an added layer of realism and training value. Companies could use this feature to train entire crews, preparing them for real-world maritime scenarios where communication and coordination are critical.

The inclusion of autonomous vessel operations could reflect the growing trend toward automation in the maritime industry. Future versions of the simulator could allow users to simulate autonomous vessels, equipping them with advanced AI systems that respond to environmental conditions and obstacles. This would not only prepare trainees for future autonomous ship operations but also make the simulator highly relevant to companies developing or operating such vessels. For example, users could practice monitoring autonomous ships, taking control in emergency situations, or operating in environments with mixed autonomous and human-piloted vessels. Furthermore, advanced AI could be integrated into the simulator to replicate real-world traffic, with AI-controlled ships responding dynamically to the player's actions. This could be particularly useful for simulating congested waterways or busy ports where human operators need to adapt quickly to the behavior of autonomous vessels.

There is also potential for further refinement of the simulator's physics model. While the current model simulates realistic ship handling, there is room for improvement in accurately representing forces such as drag, wind resistance, and wave interaction. Future work could involve a more granular approach to simulating how these forces act on the ship. For example, simulating the individual thrusts from each propeller or thruster, rather than relying on aggregate forces, would provide a higher degree of control and precision in ship handling. Moreover, future iterations could introduce failure scenarios, such as engine malfunctions or power loss, giving trainees the opportunity to practice emergency protocols. By offering deeper physics simulations, the simulator would provide a more comprehensive tool for training in advanced ship maneuvering and emergency

response scenarios.

To enhance the usability and accessibility of the simulator, future work could focus on optimizing the performance for standalone VR devices, such as the Meta Quest. While the current simulator is designed for high-performance PC-based VR systems, adapting it for standalone devices would make it more accessible to a wider audience, particularly for remote training programs or companies that want portable training solutions. Optimization techniques could focus on reducing motion sickness through smoother frame rates, adjusting the field of view, and refining user interface elements for standalone platforms. A study could also be conducted to determine the best settings for minimizing motion sickness, such as adjusting the frame rate, field of view, and latency on standalone VR headsets. Optimizing the simulator for these devices would make it more flexible and cost-effective, especially for training centers that may not have access to high-end VR equipment.

Finally, validation of the simulator's effectiveness as a training tool is a key area for future work. Conducting real-world studies that compare the performance of trainees who have used the VR simulator with those who have trained on actual ships would provide valuable insights into the effectiveness of the platform. This could involve tracking key performance metrics, such as navigation accuracy, handling efficiency, and response times, and analyzing how well VR-trained individuals perform under real-world conditions. Feedback from these studies could be used to further improve the realism and educational value of the simulator. Such validation would also help position the simulator as a trusted tool for certification programs and official maritime training curricula, boosting its credibility in the industry.

Overall, while the current VR ship simulator represents a significant step forward in maritime training, there is vast potential for future enhancements. By expanding the range of vessel types, integrating dynamic weather systems, introducing multiplayer and autonomous vessel operations, refining the physics model, optimizing for standalone VR, and validating its effectiveness, the simulator could become an even more powerful tool for training ship operators and preparing them for the challenges of real-world maritime environments.

Bibliography

[1] Steven C. Mallam, Salman Nazir, Sathiya Kumar Renganayagalu, Rethinking Maritime Education, Training, and Operations in the Digital Era: Applications for Emerging Immersive Technologies, J. Mar. Sci. Eng, 2019.

[2] Rob O' Dwyer, Class approval granted to remove onboard chief engineer, Smart Maritime Network, 2024.

[3] Sam Rutherford, Meta Quest Pro review: A next-gen headset for the VR faithful, Engadget, 2022.

[4] Unmanned Surface Vehicles, Wikipedia, `https://en.wikipedia.org/wiki/Unmanned_surface_vehicle`

[5] Kyle Mizokami, The Surprising History of Unmanned Navy Systems,U.S Naval Institute, June 2020.

[6] Uncrewed Surface Vessels, Ocean Exploration, `https://oceanexplorer.noaa.gov/technology/usv/usv.html`

[7] Brian Dunn, The Future for Unmanned Surface Vessels in the US Navy, Georgetown Security Studies Review, October 28 2020.

[8] Dr Maaten Furlong, Unmanned Surface Vehicles, The UK Marine Science and Technology Compedium.

[9] Eulalia Balestrieri, Pasquale Daponte, Luca De Vito, Francesco Lamonaca, Sensors and Measurements for Unmanned Systems: An Overview, Feature Papers in Physical Sensors Section, 22 February 2021.

[10] Gongxing Wu, Debiao Li, Hao Ding, Danda Shi, Bing Han, An overview of developments and challenges for unmanned surface vehicle autonomous berthing, Springer Link, 14 August 2023.

[11] Vessel Performance Monitoring System, `https://sbntech.com/software/vessel-performance-monitoring-system/`

[12] Inyeong Bae, Jungpyo Hong, Unmanned Surface Vehicles, Intelligent Sound Measurement Sensor and System 2022, 10 May 2023.

[13] Transforming fleet operations with advanced remote technology ready for autonomous shipping, https://www.vard.com/products-and-services/seaq-remote

[14] The editors of Encyclopedia Britannica, Weight and buoyancy in naval architecture, Encyclopedia Britannica, 13 September 2024.

[15] The editors of Encyclopedia Britannica, Resistance and propulsion in naval architecture, Encyclopedia Britannica, 13 September 2024.

[16] Basic Principles of Ship Propulsion, https://www.man-es.com/docs/default-source/marine/5510-0004-04_18-1021-basic-principles-of-ship-propulsion_web.pdf

[17] Joseph Flynt, The History of VR: When was it created and who invented it?, 3d Insider, August 12, 2019.

[18] Mehmet Ilker Berkman, History of Virtual Reality, Encyclopedia of Computer Graphics and Games. Springer, 2018.

[19] Virtual Reality Society, History of Virtual Reality, https://www.vrs.org.uk/virtual-reality/history.html, accessed September 2024.

[20] Hamad, A., Jia, B., How Virtual Reality Technology Has Changed Our Lives: An Overview of the Current and Potential Applications and Limitations, 2022.

[21] Virtual Reality Applications - Wikipedia, https://en.wikipedia.org/wiki/Virtual_reality_applications, September 2024.

[22] Virtual Reality Society, Head-Mounted Displays in Virtual Reality, https://www.vrs.org.uk/virtual-reality/head-mounted-displays.html.

[23] Wikipedia: Head-mounted Display, https://en.wikipedia.org/wiki/Head-mounted_display.

[24] Petar Franček, Kristian Jambrošić, Marko Horvat, Vedran Planinec, The Performance of Inertial Measurement Unit Sensors on Various Hardware Platforms for Binaural Head-Tracking Applications, 12 January 2023.

[25] Gordon Wetzstein, Course Notes: 3-DOF Orientation Tracking with IMUs, EE 267 Virtual Reality, https://web.stanford.edu/class/ee267/notes/ee267_notes_imu.pdf.

[26] Pedro Manuel Santos Ribeiro, Ana Clara Matos, Pedro Henrique Santos, Jaime S. Cardoso, Machine Learning Improvements to Human Motion Tracking with IMUs, Physical Sensors, 9 November 2020.

[27] Explained: Inside-Out Tracking, Physical Sensors, 03 Aug 2017, https://blog.dimitridiakopoulos.com/2017/08/03/inside-out-tracking/.

[28] Abdenour Amamra, Smooth head tracking for virtual reality applications, SIViP 11, 27 Oct 2021.

[29] Yu Lei, Lin Dong, Xiaohui Li, Xiangnan Li, Zhi Su, A Novel Sensor Fusion Approach for Precise Hand Tracking in Virtual Reality-Based Human—Computer Interaction, Computer-Aided Biomimetics, Physical Sensors, 22 July 2023.

[30] Georgios Papadopoulos, Alexandros Doumanoglou, Dimitrios Zarpalas, VRGestures: Controller and Hand Gesture Datasets for Virtual Reality, 22 January 2024.

[31] Virtual Reality Training Lab, https://www.nasa.gov/virtual-reality-lab-doug/.

[32] Osso Vr,https://www.ossovr.com/.

[33] Henry Lowood , Game Engines and Game History, Kinephanos - Médias et culture populaire, Media and popular culture, January 2014.

[34] Piotr Bajda, From Unreal Engine to Unity: the most important game engines ever created, May 14 2020.

[35] Wikipedia, Game engine, https://en.wikipedia.org/wiki/Game_engine.

[36] Scott Tykoski, Mastering Game Design with Unity 2021: Immersive Workflows, Visual Scripting, Physics Engine, GameObjects, Player Progression, Publishing, and a Lot More , November 15, 2022.

[37] Sue Blackman ,Beginning 3D Game Development with Unity: All-in-one, multiplatform game development, May 25, 2011.

[38] John P. Doran, Alan Zucconi, Unity 2018 Shaders and Effects Cookbook - Third Edition, Jun 2018.

[39] Unity Game engine, https://unity.com/how-to/organizing-your-project.

[40] Unity Game engine Documentation, https://docs.unity3d.com/Manual/index.html.

[41] Sergio Casas, Silvia Rueda, José V. Riera and Marcos Fernández, On the real-time physics simulation of a speed-boat motion, Institute of Robotics, University of Valencia, January 2012.

[42] Man B and W, Basics of propulsion systems, https://www.dieselduck.info/machine/02%20propulsion/2005%20MAN%20B&W%20Basics%20of%20propulsion%20systems.pdf.

[43] Wikipedia Maersk Honam, https://en.wikipedia.org/wiki/Maersk_Honam.