TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Head-Worn Augmented Reality for Real-Time Navigation Assistance and Event Forecasting in Maritime Operations

Author:
Georgios LAMPRINAKIS

Committee:
Aikaterini MANIA
Antonis DELIGIANNAKIS
Nikos GIATRAKOS

A thesis submitted in fulfillment of the requirements for the degree of Electrical and Computer Engineering

June 13, 2025

TECHNICAL UNIVERSITY OF CRETE

Abstract

Electrical and Computer Engineering

Augmented Reality Application for Real-time Decision-Making in Maritime

by Georgios LAMPRINAKIS

Head-worn augmented reality (AR) interfaces are becoming increasingly vital in maritime operations, where real-time decision-making and situational awareness are critical to safety and efficiency. Incorporating dynamic, real-time navigational and environmental data, including hazard forecasts and uncertainty visualizations, into a hands-free AR system without obstructing the user's field of view is essential for effective maritime navigation.

This thesis presents an innovative head-worn AR navigation and event forecasting system specifically developed to enhance situational awareness and decision-making for ship captains during maritime operations. The system dynamically integrates GPS and Automatic Identification System (AIS) data to visualize navigational routes, collision predictions, hazard zones, and critical points of interest directly into the captain's field of view. Uncertainty cones, inspired by hurricane prediction models, represent potential deviations in the predicted ship trajectories, providing crucial decision-making context for captains.

The AR interface employs intuitive gaze and gesture-based interactions, enabling hands-free operation that is essential in complex maritime scenarios. Calibration processes tailored for varied ship structures ensure precise and flexible placement of AR elements. The prototype was evaluated in sea trials conducted onboard a 44-meter cruise ship and a 30-meter tug boat under realistic maritime conditions. Feedback from expert maritime users highlighted significant improvements in navigational safety, quicker decision-making, and enhanced operational efficiency, despite challenges such as visibility under bright lighting and depth perception.

This research advances the field of maritime navigation by delivering a practical, real-time AR solution that effectively supports ship captains in high-stakes decision-making scenarios, ultimately contributing to safer and more efficient maritime operations.

Acknowledgements

I extend my heartfelt gratitude to all who have played a pivotal role in the successful culmination of this thesis.

First and foremost, I am deeply grateful to my thesis supervisor, Aikaterini MANIA, whose unwavering support, expert guidance, and insightful feedback have been instrumental throughout this research endeavor. Her expertise and insights have shaped the direction of this work and provided invaluable motivation at every stage.

My sincere thanks go to the entire SURREAL lab for their collaboration and assistance. Their camaraderie and dedication created an inspiring environment that greatly contributed to the success of this project. I am especially thankful to my fellow students and colleagues who offered daily encouragement and made this journey more enjoyable.

I would also like to express my appreciation for the opportunity provided by the CREXDATA project. Working on this project allowed me to develop my application and thesis simultaneously, offering a unique platform to contribute to innovative research in augmented reality and maritime navigation.

Finally, I wish to express my heartfelt appreciation to my family for their unwavering support and encouragement. Their love and confidence in me have been a constant source of motivation throughout my academic journey. Last but not least, I am grateful for the support and camaraderie of my friends, who have made this journey more rewarding and memorable.

This thesis would not have been possible without the collective support, mentorship, and contributions of all those mentioned above.

Georgios Lamprinakis
June 13, 2025

Contents

List of Figures

Chapter 1

Introduction

1.1 Brief Overview

Head-worn augmented reality (AR) interfaces are transforming maritime operations by significantly enhancing situational awareness(Odd Sveinung Hareide, 2017) and supporting real-time decision-making processes. AR technology finds diverse applications in the maritime sector, from providing real-time navigational assistance and hazard visualization to training through realistic scenario simulations(Muhammad Raziq Kazura, 2024; Steven C. Mallam, 2019). Additionally, AR systems have become essential for maintenance((2016), 2016) and repair tasks(2018), 2018, facilitating remote guidance from onshore experts to onboard crew members, thereby reducing operational downtime and improving service quality.

The first Maritime AR Systems (MARS) aimed at streamlining navigation by consolidating various ship instruments into a single AR-enhanced interface. These early implementations effectively improved situational awareness and reduced cognitive load among users, marking a notable advancement in maritime operations(Teo Chee Hong and Kenny, 2015). Subsequent systems have evolved by integrating multiple data sources, including the Automatic Identification System (AIS), radar, and navigational route information, through hybrid interfaces that merge contextual world-referenced 3D visualizations with non-contextual data(Jung Min Lee, 2016). Beyond open-sea navigation, AR technology has also shown promise in harbor maneuvers, where smart glasses provide maritime pilots with essential navigational cues, supporting precision navigation and enhancing situational awareness in confined environments(Marie-Christin Ostendorp, 2015).

Historically, maritime AR implementations predominantly relied on monitor-based solutions to present AR content(Tadatsugi Okazaki, 2017; Jung Min Lee, 2016; Byeong-Wook Nam, 2017). Recent technological advancements, however, have shifted focus towards wearable headsets and smart glasses, enabling hands-free operation and directly projecting essential information into the user's field of vision(Frydenberg and Eikenes, 2018; Gerald Moulis, 2015; Oever, 2022). Despite these advancements, significant research challenges persist, particularly concerning the visualization and interaction with real-time data. Specifically, managing the delicate balance of delivering vital navigational and environmental forecasts while ensuring minimal obstruction of the user's field of view remains a crucial safety and usability consideration.

This thesis presents an innovative head-worn AR navigation and event forecasting system specifically developed to enhance wayfinding and situational awareness for

ship captains during maritime operations. The system incorporates a range of intuitive interaction techniques, such as gaze and gesture-based interactions, alongside dynamic visual aids like a mini-map highlighting essential points of interest (POIs) and uncertainty cones for projected navigational paths, thereby offering captains a comprehensive, real-time overview of their operational environment.

FIGURE 1.1: Testing the AR system on the field.

The specific contributions of this thesis include:

- **Real-Time AR Overlay System:**
  Development of an AR system that integrates a server with head-worn AR devices to dynamically overlay safe routes, ship trajectories, and hazard zones directly into the captain's field of view, thereby enhancing situational awareness and navigation efficiency.
- **Integration of GPS and AIS Data:**
  Seamless incorporation of GPS and Automatic Identification System (AIS) data with AR to visualize critical Points of Interest (POIs), such as ports and restricted zones, on the horizon, as well as hazards near the ship's current location.
- **Uncertainty Visualization and Interaction Techniques:**
  Implementation of uncertainty visualization for predicting ship trajectories over time, coupled with novel gaze and gesture-based interaction techniques. This includes the placement of visual information tailored to any vessel's structure and toggle controls that allow for unobstructed viewing.

- **Proof-of-Concept Testing in Simulated Environments:**
  Conducting proof-of-concept evaluations in simulated maritime settings to validate the system's functionality and usability prior to real-world deployment.
- **Expert User Evaluation:**
  Comprehensive evaluation of the AR system by expert users in scenarios involving collision avoidance demonstrating its potential to enhance navigation practices and decision-making processes in high-stakes maritime operations.

Feedback from the trials with ship captains underscored the system's ability to significantly enhance operational efficiency, improve situational awareness, and accelerate real-time decision-making. The system's capability to provide continuous updates on ship trajectories, collision risks, and environmental hazards establishes it as a valuable tool for maritime navigation. By delivering real-time, dynamic information directly into the captain's field of view, the system empowers captains to make better-informed decisions in complex and high-stakes maritime environments.

1.2 Thesis Structure

Chapter 1: Introduction This chapter outlines the core motivation, objectives, and research scope of the AR-based maritime navigation system. It highlights the real-world challenges that necessitated the creation of a head-worn AR solution for ship captains, and sets the context for the subsequent chapters.

Chapter 2: Foundations and State-of-the-Art in Augmented Reality In this part, fundamental concepts and existing literature on maritime protocols, services, and sensor data integration are presented. The chapter also provides an overview of Augmented Reality (AR) technology, tracing its evolution, practical applications, and the specifics of applying AR to navigation and hazard assessment in maritime domains.

Chapter 3: Technological Foundations and Platform Overview This section introduces the major tools, frameworks, and technologies employed in developing the AR application. Topics include the Unity game engine, the Mixed Reality Toolkit (MRTK3), and the Microsoft HoloLens 2, along with external services such as Mapbox for location-based data. The chapter explains how these elements converge to form the backbone of the system.

Chapter 4: System Architecture and Implementation As the most detailed segment, this chapter focuses on the technical realization of the AR solution. It details the design of the user interface, the functionalities for rendering Points of Interest (POIs), map integration, real-time route visualization, and the novel interaction techniques implemented. Core implementation challenges, including calibration processes, handling of prediction data, and how the system deals with 3D placement in a maritime environment, are examined in depth.

Chapter 5: Evaluation This chapter outlines the sea trials and pilot tests conducted to assess the system's performance under realistic maritime conditions. It includes user feedback from experienced captains or maritime professionals, showcasing the system's usability, effectiveness, and areas for improvement. Observations related to hardware constraints, feedback on AR interactions, and insights on potential enhancements are discussed.

Chapter 6: Conclusions and Future Work The final chapter summarizes the principal findings and broader impact of the work, reflecting on how the AR system contributes to enhanced situational awareness and decision-making within maritime operations. It also highlights possible avenues for further research and development, such as expanded sensor integration, more advanced predictive analytics, or improved resilience in challenging weather conditions.

Bibliography This concluding section contains a comprehensive list of references and sources cited throughout the thesis, ensuring proper academic attribution and offering readers guidance for additional reading.

Chapter 2

Foundations and State-of-the-Art in Augmented Reality

2.1 Augmented Reality: Principles, History, and Applications

Augmented Reality (AR) is a transformative technology that overlays digital information—such as images, data visualizations, annotations, or 3D objects—directly onto a user's perception of the physical world. Unlike Virtual Reality (VR), which immerses users in entirely synthetic environments, AR enhances real-world experiences by introducing virtual content that is spatially and contextually aligned with physical surroundings. The convergence of sensor technologies, computer vision, and real-time rendering has established AR as a key driver of innovation across industries ranging from entertainment and education to healthcare and industrial design.

2.1.1 Defining Augmented Reality

At its core, AR is defined by the seamless integration of digital elements with the user's real-world environment in real time. Three core characteristics that distinguish AR systems:

1. **Combination of Real and Virtual:** The system blends computer-generated content with live views of the physical world.
2. **Real-Time Interactivity:** The augmentation is interactive, responding dynamically to user actions and environmental changes.
3. **Registration in 3D:** Virtual elements are registered in three dimensions, so they appear correctly aligned with real objects as the user moves or shifts perspective.

These criteria distinguish AR from related concepts such as VR (full immersion in a virtual environment) and simple heads-up displays (HUDs) that do not offer contextual, spatial registration.

2.1.2 A Brief History of Augmented Reality

The conceptual origins of AR can be traced to the 1960s, with Ivan Sutherland's development of the first head-mounted display (HMD) system, often referred to as the "Sword of Damocles". This pioneering device rendered simple wireframe graphics over a user's real-world view, establishing foundational ideas for future AR systems.

The term "Augmented Reality" was first introduced by Tom Caudell and David Mizell in the early 1990s to describe a digital overlay system for assisting aircraft assembly workers at Boeing . In subsequent decades, AR research flourished within academic and industrial labs, supported by advances in computer graphics, mobile computing, and sensor technologies. Notable milestones include the development of early mobile AR systems in the 2000s, the introduction of marker-based AR toolkits, and the widespread adoption of smartphone AR applications in the 2010s.

Recent years have seen the emergence of robust, wearable AR devices such as Microsoft HoloLens, Magic Leap, and enterprise-focused smart glasses, which offer hands-free, spatially aware augmentation in real-world environments.

2.1.3 The Reality-Virtuality Continuum

Understanding AR requires situating it within the broader spectrum of mediated reality. Paul Milgram and Fumio Kishino proposed the "Reality-Virtuality Continuum," with the real environment (RE) at one end, fully immersive virtual environments (VR) at the other, and various forms of AR and Mixed Reality (MR) occupying the intermediate space. This continuum reflects the range of possible experiences, from unmediated physical reality through to environments dominated by digital content.

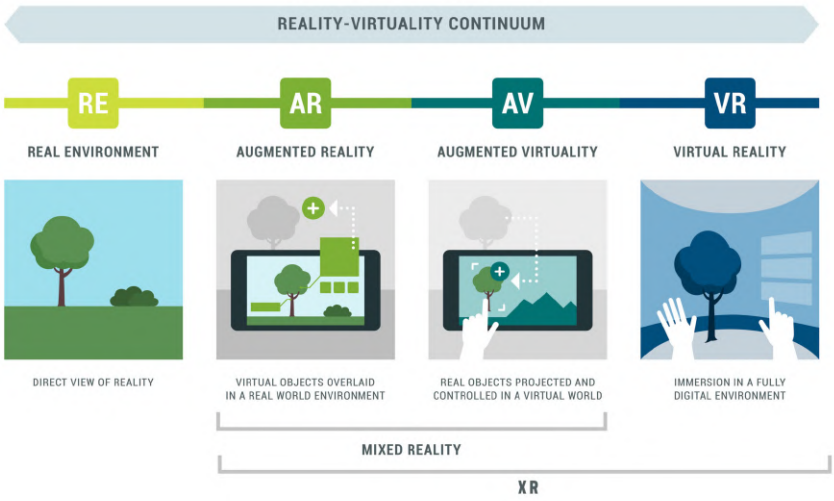


FIGURE 2.1: The Reality-Virtuality Continuum, illustrating the position of Augmented Reality between the real environment and fully virtual experiences.

2.1.4 Core Technologies Enabling AR

Modern AR systems are built upon a convergence of multiple hardware and software technologies, including:

- **Displays:** AR experiences can be delivered via smartphones, tablets, head-worn displays (e.g., HoloLens, Magic Leap), or projection systems. Transparent and semi-transparent displays allow digital content to appear within the user's real-world field of view.

- **Sensing and Tracking:** Cameras, inertial measurement units (IMUs), depth sensors, and GPS enable the system to localize the user and track environmental features, ensuring accurate registration of virtual objects.
- **Computer Vision:** Algorithms for feature detection, marker tracking, and simultaneous localization and mapping (SLAM) are critical for robust, markerless AR experiences in uncontrolled environments.
- **Rendering:** Real-time 3D rendering engines (e.g., Unity, Unreal Engine) generate visually compelling, interactive digital content responsive to both user input and environmental context.
- **Interaction:** AR interfaces increasingly support multimodal input, including touch, voice, gesture, gaze, and spatial manipulation, enabling intuitive and hands-free interactions.

2.1.5 Types of Augmentation and Interface

AR systems vary widely in their approach to augmentation and user interaction:

- **Marker-Based vs. Markerless:** Early AR often relied on visual markers (fiducials or QR codes) to anchor virtual content. Contemporary systems leverage natural feature recognition or environmental understanding for markerless operation.
- **Spatial vs. Non-Spatial Augmentation:** Spatial AR aligns digital content precisely with physical objects or locations (e.g., annotating machinery), while non-spatial AR overlays information in a fixed screen space (e.g., HUDs or dashboards).
- **2D vs. 3D Content:** Some AR applications deliver 2D annotations or visual cues, while others present fully interactive 3D models, virtual interfaces, or immersive simulations.

2.1.6 Applications of Augmented Reality

AR has seen adoption and experimentation across a wide range of domains:

- **Entertainment and Gaming:** Titles such as Pokémon GO and Minecraft Earth popularized mobile AR gaming, merging digital play with real-world exploration.
- **Education and Training:** AR is used to visualize abstract concepts, support hands-on learning, and provide interactive simulations for technical training in fields like medicine, engineering, and aviation.
- **Industrial and Enterprise:** AR assists workers in complex assembly, maintenance, and inspection tasks by overlaying step-by-step instructions, schematics, and sensor data onto physical equipment.
- **Navigation and Wayfinding:** AR enhances personal navigation by superimposing routes, landmarks, and contextual alerts onto the user's environment.
- **Healthcare:** Applications include surgical navigation, anatomy visualization, and rehabilitation, leveraging AR to improve accuracy, safety, and patient outcomes.

- **Retail and E-Commerce:** AR enables virtual product visualization in real-world settings, supporting "try-before-you-buy" experiences and personalized advertising.
- **Research and Visualization:** Scientists and engineers use AR to explore complex data sets, visualize simulations, and facilitate remote collaboration.

(A) Education (B) Healthcare (C) Manufacturing

FIGURE 2.2: **Sample Applications of AR.** From classrooms and operating rooms to factory floors, AR is enabling new ways to interact with complex information.

2.1.7 Emerging Trends and Research Challenges in AR

As Augmented Reality technologies continue to mature, several key trends are shaping both research and industry adoption. One notable direction is the emergence of the *AR Cloud*, which enables persistent, shared digital content to be anchored across users and devices, supporting collaborative and large-scale experiences. Advances in artificial intelligence (AI), particularly in computer vision and scene understanding, are making AR systems more robust, context-aware, and capable of dynamic semantic mapping of real-world environments.

The proliferation of web-based AR (*WebAR*) and cross-platform frameworks like *OpenXR* and *WebXR* has greatly expanded accessibility, allowing users to experience AR content directly through browsers without the need for specialized applications. Meanwhile, cloud/edge processing is alleviating device constraints, enabling real-time, high-fidelity rendering and computation beyond local hardware limits.

Major Industry Players and Platforms Recent years have seen the entry of major technology companies, driving commercial progress and standardization in the AR ecosystem. Key platforms and devices include:

- **Apple Vision Pro** and **ARKit** (Apple)
- **Google ARCore** (Android)
- **Meta Quest Pro** (Meta/Facebook)
- **Microsoft HoloLens 2** (Microsoft)
- **Magic Leap 2** (Magic Leap)
- **Snap AR**, **Niantic Lightship**, and **Vuzix Blade**

These platforms are fostering a rich development ecosystem and enabling new forms of AR experiences for both consumers and industry.

Current Challenges and Open Problems Despite substantial progress, AR remains an active area of research with significant open challenges:

- **Robust Registration and Tracking:** Achieving drift-free, accurate spatial alignment in dynamic, cluttered, or outdoor environments.
- **Visual Realism and Occlusion:** Seamlessly blending virtual and real-world elements, including handling of occlusion, lighting, and shadows.
- **Human Factors:** Addressing visual fatigue, ergonomic design, limited field of view, and cognitive overload, especially for wearable devices.
- **Power Efficiency:** Balancing device miniaturization with the processing demands of real-time AR.
- **Privacy and Security:** Managing sensitive data, persistent environmental mapping, and the ethical implications of ubiquitous sensors and always-on cameras.
- **Accessibility and Inclusion:** Designing AR experiences that are usable by people with diverse abilities and avoiding the reinforcement of digital divides.

TABLE 2.1: Comparison of Representative AR Devices and Platforms (2025)

Device/Platform	Display Type	FOV (Diagonal)	Tracking	Input Modalities
Microsoft HoloLens 2	Waveguide Optical	∼52°	Inside-out (SLAM)	Hand, gaze, voice
Magic Leap 2	Waveguide Optical	∼70°	Inside-out (SLAM)	Hand, controller, voice
Apple Vision Pro	Micro-OLED	∼100°	Inside-out (SLAM)	Eye, hand, voice
Meta Quest Pro	Pancake Optics	∼106°	Inside-out (SLAM)	Hand, controller, voice
Vuzix Blade	Waveguide Optical	∼20°	3DOF, GPS	Touchpad, voice
Smartphones (ARKit/ARCore)	LCD/OLED	Screen-limited	Camera, IMU, GPS	Touch, gesture, voice

Standardization and Interoperability Efforts by organizations such as the Khronos Group (*OpenXR*, *WebXR*) and the Open Geospatial Consortium (*ARML*) are promoting interoperability and cross-platform content delivery. These standards facilitate the creation of robust, long-lived AR experiences that are accessible across a range of hardware and software environments.

2.1.8 From General Applications to Maritime Context

While AR has demonstrated transformative potential across numerous domains, the maritime sector presents a uniquely challenging environment characterized by dynamic conditions, safety-critical operations, and complex sensor integration. These characteristics make it both a demanding and promising domain for AR-driven innovation. The following sections review how AR technologies have been tailored for maritime use, highlight existing systems and limitations, and motivate the novel contributions of this thesis.

2.1.9 Applications of AR in Maritime Operations

Head-worn augmented reality (AR) interfaces are rapidly transforming maritime operations by significantly enhancing situational awareness and facilitating real-time decision-making, as highlighted by recent maritime eye-tracking studies (Odd Sveinung Hareide, 2017). AR technologies have been effectively utilized in various maritime domains, including navigation support through the visualization of

real-time route information and environmental hazards, training simulations via realistic scenarios (Muhammad Raziq Kazura, 2024; Steven C. Mallam, 2019), and critical maintenance and repair tasks guided remotely by on-shore experts ((2016), 2016; (2018), 2018).

Initial Maritime AR Systems (MARS) focused on consolidating diverse ship-instrument data into a singular AR interface. Even these early prototypes demonstrated considerable improvements in situational awareness and reductions in cognitive load (Teo Chee Hong and Kenny, 2015). Subsequent systems have evolved to integrate Automatic Identification System (AIS), radar, and navigational routes, presenting hybrid interfaces combining world-referenced 3D visualizations with essential contextual information (Jung Min Lee, 2016). AR technology has also shown particular efficacy in assisting maritime pilots during complex harbor maneuvers, employing smart glasses to deliver critical navigation data and enhance situational awareness in confined and congested environments (Marie-Christin Ostendorp, 2015).

While earlier maritime AR solutions primarily relied on monitor-based displays (Tadatsugi Okazaki, 2017; Jung Min Lee, 2016; Byeong-Wook Nam, 2017), recent developments have shifted towards wearable devices such as headsets and smart glasses (Frydenberg and Eikenes, 2018; Gerald Moulis, 2015). These head-worn AR solutions provide the advantage of hands-free interaction and deliver data directly into the user's line of sight, significantly improving operational convenience (Oever, 2022). Nevertheless, a critical ongoing research challenge is to enable interactive visualizations of real-time data, including forecast hazards and uncertainty indicators, without obstructing the operator's natural field of view (Safranoglou et al., 2024; Koulieris et al., 2019).

To tackle these challenges, this paper introduces a novel head-worn AR navigation system designed explicitly for enhancing wayfinding and situational awareness among ship captains at sea. The system integrates a suite of advanced interaction techniques, featuring an interactive mini-map that clearly delineates points of interest (POIs) and uncertainty cones that visually communicate potential navigational deviations. These elements are combined to deliver a comprehensive, real-time overview of the vessel's surroundings, directly enhancing navigational safety and efficiency. Moreover, the system seamlessly integrates GPS and AIS data, dynamically overlaying critical navigational information and event forecasts directly into the captain's field of view. This is achieved while ensuring minimal visual occlusion, thereby maintaining unobstructed observation of the maritime environment.

AR Systems for Maritime Navigation

AR systems for maritime navigation have been extensively explored in both commercial deployments and academic research. For example, the commercially available Furuno Envision System (Furuno Product Solutions, 2019) provides a monoscopic AR solution, overlaying navigational data such as AIS information, routes, and waypoints onto a live video feed from a bow-mounted camera. Although it successfully enhances situational awareness, the system does not significantly reduce head-down time, limiting its overall operational effectiveness (Odd Sveinung Hareide, 2017). Furthermore, the absence of documented operational deployment or formally assigned technology readiness levels leaves its practical utility uncertain (Grabowski, 2015). Our approach differs substantially by emphasizing interactive

event forecasting, innovative uncertainty visualization, and real-time navigation assistance.

Oh et al. (Jaeyong OH, 2016) developed a monoscopic AR prototype visualizing various navigational elements, including heading, routes, and traffic data through a combination of 2D and 3D graphics. Despite validation in real-world maritime settings, the prototype exhibited critical usability challenges, particularly information overload and overlapping visuals, leading to reduced clarity and heightened user-perceived workload. Similarly, Lee et al. (Jung Min Lee, 2016) proposed a computer-vision-based system for vessel detection, incorporating 2D AR visualizations of AIS and ship data. However, the absence of an integrated framework for AR element management severely constrained its practical scalability.

In contrast, Leite et al. (Leite et al., 2022) explored obstacle detection and situational awareness enhancement using AR grid overlays and zoomed visualizations. Extending these concepts, Frydenberg et al. (Frydenberg and Eikenes, 2018) introduced stereoscopic AR prototypes, notably through the SEDNA project aimed at Arctic navigation. The resulting design guidelines (Kjetil Nordby, 2020) established foundational principles for stereoscopic AR application components and effective display zoning. Subsequent field trials with Arctic icebreaker crews underscored AR's effectiveness in improving communication and situational awareness, especially for high-stakes human-robot teamwork (Frydenberg et al., 2021).

Our proposed AR navigation system significantly advances beyond existing solutions by delivering a tailored stereoscopic interface adaptable to diverse ship infrastructures. It dynamically visualizes safe routes, highlights critical hazards, and explicitly addresses positional uncertainties in real-time, effectively enhancing maritime decision-making and navigational safety.

Uncertainty Visualization in Navigation Systems

Visualizing uncertainty is crucial for effective data representation, significantly influencing user comprehension of data reliability and variability. Effective uncertainty visualization can substantially improve decision-making by clearly communicating potential data variability (Potter, Rosen, and Johnson, 2012). However, its application in maritime contexts remains relatively unexplored, with most research traditionally focusing on historical data and adjacent fields. For instance, Plumejeaud-Perreau and Marzagalli (Christine Plumejeaud-Perreau, 2022) introduced visual tools for historical maritime route uncertainty, employing color-coded maps and schematic dashed lines to indicate certainty levels and inferred segments. Their interactive querying functionalities greatly simplified complex historical data, aiding decision-making processes.

Riveiro et al. (Riveiro et al., 2014) investigated uncertainty visualization methods in an air-defense scenario, employing semi-transparent overlays and varying line thickness to communicate positional uncertainty and sensor accuracy. Although such visualizations fostered more cautious and effective threat prioritization, overall task performance and user confidence were largely unchanged. Similarly, Broad et al. (Kenneth Broad and Steketee, 2007) identified significant misinterpretations of the "cone of uncertainty" visualization commonly used for hurricane path predictions. These findings emphasize the critical need for clear, intuitive uncertainty representations to avoid ambiguity, particularly relevant in maritime contexts where misinterpretation can lead to severe operational consequences.

Padilla et al. (Padilla, Kay, and Hullman, 2021) demonstrated ensemble displays and hypothetical outcome plots as effective means to enhance risk assessments by clearly representing potential data variations, an approach directly applicable to maritime trajectory predictions. Extending this notion, Liu et al. (Liu et al., 2015) proposed visualizations of hurricane predictions based on storm path ensembles, leveraging radial basis functions and simplicial depth to estimate uncertainty intuitively. Such visualization methods, presenting overlapping confidence intervals, significantly enhance the intuitive understanding of positional risks.

Incorporating these principles, our AR system features uncertainty cones to visualize potential trajectory deviations dynamically. This cone's gradient varies according to GPS accuracy and environmental factors, effectively communicating navigational uncertainties to the captain. Integrating color-coded schemes and semi-transparent overlays, our approach ensures clarity and reduces cognitive load, significantly enhancing navigational safety in complex maritime environments.

Chapter 3

Technological Foundations and Platform Overview

3.1 Rationale for Technology Choices

The technology stack adopted in this thesis was carefully curated to meet the demanding requirements of real-time, spatially-aware maritime augmented reality. Key selection criteria included platform maturity, cross-platform deployment capabilities, active support communities, extensibility, and robust track records in both industrial and research deployments.

Unity was chosen as the foundational engine for its well-established AR/MR ecosystem, intuitive scene editor, and powerful C# scripting environment. Its broad device compatibility and mature support for Universal Windows Platform (UWP) builds enable seamless deployment to HoloLens 2 and future AR hardware. Furthermore, Unity's large Asset Store ecosystem accelerates prototyping and reduces development overhead—crucial in research timelines.

MRTK 3 (Mixed Reality Toolkit) was selected to provide spatial interaction paradigms, ergonomic user interface (UI) components, and cross-device abstractions for hands-free, gaze-based, and gesture-based control. MRTK 3's modular architecture enables rapid integration of new features while maintaining robust support for the unique interaction modalities of HoloLens 2. Its open-source nature and large developer community ensure long-term sustainability and flexibility.

Microsoft HoloLens 2 serves as the primary device platform. Its untethered operation, precise spatial mapping, and advanced hand/eye tracking capabilities make it uniquely suited for dynamic maritime environments, where hands-free operation and spatial awareness are paramount. The device's proven deployment in industrial and safety-critical settings further supports its selection.

Mapbox was integrated to address the mapping and geospatial visualization requirements of the project. Mapbox's Unity SDK offers real-time rendering of custom basemaps and overlays, with support for rapid updates from live ship sensors and navigation data. Its flexibility in styling, support for both raster and vector layers, and strong developer community make it a robust choice for maritime navigation scenarios.

TABLE 3.1: Core Technology Platform Overview

Platform	Primary Role	Key Features
Unity	Core development	Cross-platform, AR/MR support, C#
MRTK 3	Interaction/UI	Spatial UI, hand/eye/gaze, extensibility
HoloLens 2	Device	Untethered AR, hand tracking, field deployable
Mapbox	Mapping/Geo	Custom basemaps, overlays, Unity integration

A variety of alternative solutions were assessed prior to final selection. **Unreal Engine**, while offering exceptional photorealistic graphics, lacks Unity's mature support for rapid AR prototyping, C# scripting, and out-of-the-box HoloLens 2 deployment. For spatial UI and interaction, **AR Foundation** and other toolkits were evaluated, but none matched the breadth of interaction patterns, extensibility, and device abstraction provided by MRTK 3, especially for head-worn, hands-free scenarios. In terms of geospatial visualization, **Cesium** and **HERE Maps** offer strong mapping capabilities, but Mapbox's superior Unity integration, customizability, and real-time update performance made it preferable for the rapid demands of maritime navigation.

By integrating these platforms, the project leverages cutting-edge AR development tools, ensuring a balance between rapid prototyping, operational robustness, and the capacity to extend or adapt the system for future research and commercial applications.

3.2 Unity: A Comprehensive Platform for Real-Time 3D Development

Unity is a cross-platform engine and integrated development environment (IDE) designed for the creation of interactive, real-time 2D and 3D content. Originally developed by Unity Technologies and released in 2005, Unity has evolved from a niche game development tool into a versatile and widely adopted platform underpinning diverse fields, from entertainment to engineering, architecture, simulation, education, and research.

FIGURE 3.1: Unity

3.2.1 Core Architecture and Functionality

At its foundation, Unity provides a component-based architecture. Each object within a scene, referred to as a *GameObject*, can be extended with a range of modular *components* that define its appearance, behavior, and interactivity. This architecture encourages code reusability, scalability, and a clean separation of concerns, facilitating the management of complex scenes and interactive logic.

Unity's scripting system is primarily based on C#, a modern, object-oriented programming language. Developers can write scripts to define custom behaviors, manipulate assets, manage game logic, and interact with external systems. The scripting API is extensive, offering fine-grained control over physics, animation, input, networking, rendering, and much more.

3.2.2 Editor and Workflow

One of Unity's distinguishing features is its graphical Editor, which allows developers to assemble scenes visually, adjust parameters in real time, and preview content across a variety of devices and platforms. The Editor integrates asset management, scene composition, prefab instantiation, animation, UI design, and debugging tools within a single interface, significantly streamlining the iterative development process.

Unity supports a broad range of asset types—including 3D models, textures, audio, and video—enabling seamless import from popular digital content creation tools. Through its Asset Store, Unity also fosters a large ecosystem of reusable assets, scripts, and extensions, accelerating prototyping and reducing development overhead.

3.2.3 Rendering and Physics

Unity's rendering pipeline is highly flexible. Developers can choose between the Built-In Render Pipeline, the Universal Render Pipeline (URP), or the High Definition Render Pipeline (HDRP), depending on the target device and visual fidelity requirements. These pipelines support advanced features such as physically based rendering (PBR), real-time global illumination, post-processing effects, and shader customization.

For real-time physics simulation, Unity integrates the NVIDIA PhysX engine for 3D physics, and a robust 2D physics system, both providing collision detection, rigidbody dynamics, joints, triggers, and complex constraint systems. These features are crucial for interactive and realistic simulation in both gaming and non-gaming contexts.

3.2.4 Cross-Platform Deployment

Unity's cross-platform capabilities are among its greatest strengths. A single project can be deployed across a multitude of platforms, including Windows, macOS, Linux, Android, iOS, WebGL, major game consoles, and a wide variety of extended reality (XR) devices. This flexibility allows developers to address heterogeneous audiences without extensive platform-specific modifications.

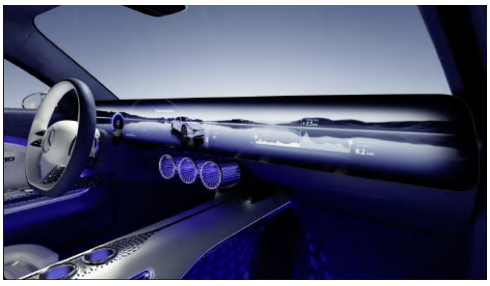
3.2.5 Unity in Augmented Reality (AR) Development

In recent years, Unity has established itself as a premier platform for the development of augmented reality (AR), virtual reality (VR), and mixed reality (MR) applications. Unity provides native support for leading AR devices—such as Microsoft HoloLens, Meta Quest, Magic Leap, and various mobile AR frameworks—through a set of extensible AR plug-ins and packages.

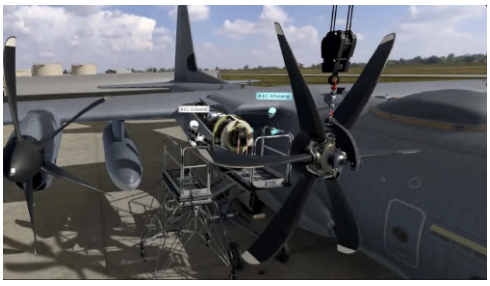
Unity's AR Interaction Toolkit, along with integrations like the Mixed Reality Toolkit (MRTK), offers out-of-the-box components for gesture recognition, spatial anchoring, input handling, and environment understanding. These abstractions dramatically reduce the complexity of developing sophisticated, interactive AR experiences, making Unity a common choice in both industrial and research settings.

3.2.6 Research, Prototyping, and Beyond

Beyond entertainment, Unity is increasingly employed as a research and prototyping tool. Its real-time simulation capabilities, rapid iteration cycle, and access to sensor data and external APIs make it well-suited for developing and validating new interaction techniques, visualization systems, and user studies. In fields such as robotics, navigation, healthcare, and maritime operations, Unity is used to build interactive simulations, training environments, and visualization dashboards.



(A) Mercedes-Benz MB.OS



(B) Virtual Hangar by Mass Virtual

FIGURE 3.2: Selected industrial applications developed with Unity

3.2.7 Summary

In summary, Unity is a robust, extensible, and widely adopted real-time development platform. Its combination of an intuitive visual Editor, powerful scripting capabilities, advanced rendering and physics, and unparalleled cross-platform reach make it a foundational technology for a wide array of interactive 3D applications. Whether for entertainment, education, simulation, or advanced research in fields like augmented reality, Unity offers the tools necessary to realize complex, interactive digital experiences.

Specific Role Within This Thesis

For the system developed in this thesis, Unity acts as the backbone: live GPS possitions and AIS target data are ingested through an intermediate server, parsed in C, and rendered via AR Foundation as world-anchored navigation cues. The XR Interaction Toolkit supplies gaze-based toggles for route layers and un- certainty

cones, while standard GameObject workflows ensure compatibility with shipping-grade devices. Built for Universal Windows Platform, the project deploys directly to on-board headsets or simulator rigs, confirming Unity's ability to bridge rapid prototyping with rugged maritime field use.

3.3 Mixed Reality Toolkit 3 (MRTK 3): Enabling Spatial Interaction in Unity

The Mixed Reality Toolkit 3 (MRTK 3) is a comprehensive, open-source framework developed by Microsoft and the broader XR community to accelerate the creation of immersive mixed reality (MR) applications. Building upon earlier iterations of the MRTK, version 3 represents a major redesign focused on modularity, cross-platform compatibility, and integration with modern Unity workflows. MRTK 3 delivers a rich set of building blocks, interaction patterns, and system services tailored for developing spatially aware and intuitive experiences on a wide range of XR devices.

FIGURE 3.3: Mixed Reality Toolkit

3.3.1 Evolution and Objectives

The origins of MRTK trace back to early efforts in supporting development for Microsoft HoloLens, with the first toolkit releases providing fundamental interaction components and utilities for Unity-based MR applications. As spatial computing platforms matured, demand grew for frameworks capable of supporting not only Microsoft hardware but also a diverse ecosystem of VR and AR devices.

MRTK 3 was architected in response to this evolving landscape. Its goals include promoting reusable interaction paradigms, minimizing device-specific dependencies, and aligning closely with Unity's UI and input systems. The toolkit embraces open standards and extensibility, enabling developers to target HoloLens, Windows Mixed Reality, Meta Quest, Magic Leap, and emerging platforms with minimal code changes.

3.3.2 Core Architecture and Features

MRTK 3 adopts a modular, package-based structure. Individual features—such as input, spatial awareness, interaction components, and UI controls—are organized into separate Unity packages. This modularity allows developers to include only the necessary functionality, resulting in lighter builds and easier project maintenance.

Key components and features of MRTK 3 include:

- **Input System:** Abstracts hand tracking, eye gaze, controllers, and traditional inputs into a unified model. Developers can process input events generically, enabling seamless cross-device interactions.
- **Spatial Awareness:** Provides access to environmental understanding data, such as surface meshes, planes, and spatial anchors, allowing applications to react to the real-world context.
- **Interaction Building Blocks:** Includes ready-made components for gaze, gesture, and voice interaction. Manipulation handlers, bounds controls, near and far interaction rays, and tooltips accelerate prototyping of spatial user interfaces.
- **UX Controls:** A library of customizable 3D and 2D UI elements—buttons, sliders, menus, dialogs—designed for use in world space and optimized for hands-free, spatial interaction.
- **Scene System and Services:** Utilities for managing scene transitions, spatial anchors, and session persistence, crucial for multi-room or collaborative mixed reality experiences.

3.3.3 Integration with Unity and Cross-Platform Support

MRTK 3 is designed to integrate seamlessly with Unity's latest workflows, leveraging features like Unity's XR Interaction Toolkit and new input subsystems. Developers interact with MRTK via Inspector-based configuration, visual scene tools, and a robust scripting API. The toolkit also supports Unity's Package Manager for straightforward installation and updates.

A key strength of MRTK 3 is its commitment to platform abstraction. By providing consistent interfaces for spatial input and interaction, applications built on MRTK 3 can be deployed to a variety of XR devices with minimal modification. This reduces development overhead, improves maintainability, and future-proofs applications as new hardware emerges.

3.3.4 Best Practices and Research Applications

MRTK 3 promotes best practices for spatial interaction design. Its components are informed by human-computer interaction (HCI) research and Microsoft's guidelines for effective MR UX, supporting discoverability, feedback, and ergonomic manipulation in 3D space.

In research and prototyping contexts, MRTK 3 is frequently used for:

- **Rapid Prototyping**: Quickly assembling and testing new interaction paradigms in AR and VR.

- **User Studies**: Implementing experimental setups with robust, repeatable spatial interfaces.
- **Data Visualization**: Anchoring 3D data, annotations, and user interfaces within physical or virtual environments.
- **Collaborative Applications**: Building shared MR experiences leveraging spatial anchors and synchronized scenes.

3.3.5 Community and Ecosystem

As an open-source project, MRTK 3 benefits from a vibrant community of contributors and users. Extensive documentation, sample scenes, and open forums facilitate learning, troubleshooting, and the dissemination of best practices. The toolkit's extensibility encourages custom solutions and the integration of third-party components.

3.3.6 Summary

MRTK 3 is a foundational toolkit for modern mixed reality development within Unity. By providing robust abstractions for input, interaction, spatial awareness, and user interface, MRTK 3 empowers developers and researchers to create advanced XR applications efficiently and with high usability across multiple hardware platforms.

Specific Role Within This Thesis

In the context of this thesis, MRTK3 facilitated the development of an intuitive user interface and dynamic in- teraction system on HoloLens 2. Pre-built volumetric elements—such as buttons, hand menus, and sliders—streamlined the design process, enabling captain to manage real-time data displays without manual controllers. Meanwhile, gaze- based interactions and gesture recognition ensured that users could seamlessly ma- nipulate on-screen holograms while maintaining situational awareness. This was particularly valuable for time-sensitive, safety-critical operations at sea, where quick access to navigational overlays and interactive data visualizations can significantly enhance operational efficiency.

3.4 Microsoft HoloLens 2: An Advanced Platform for Spatial Computing

The Microsoft HoloLens 2 is a self-contained, head-mounted augmented reality (AR) device designed to enable intuitive, hands-free interaction with digital content overlaid on the real world. Building on the foundations laid by the original HoloLens, the second-generation device introduces significant advancements in comfort, optics, tracking accuracy, and user experience, making it a leading platform for both industrial and research applications in mixed reality (MR).

3.4.1 Device Overview and Evolution

First announced in 2019, HoloLens 2 represents Microsoft's commitment to spatial computing and enterprise-grade AR solutions. While the initial HoloLens pioneered untethered holographic experiences, HoloLens 2 focuses on enhancing usability, immersion, and ergonomics. It features a lighter, more balanced form factor, improved

FIGURE 3.4: Microsoft HoloLens 2 headset

weight distribution, and a flip-up visor to accommodate extended wear and fast transitions between AR and the physical environment.

3.4.2 Key Hardware Features

The HoloLens 2 combines multiple sensor systems and advanced processing to support robust mixed reality experiences:

- **Optics and Display:** The device utilizes see-through waveguide displays with a field of view exceeding 50 degrees, offering vibrant, high-resolution holograms anchored in the real world. Eye comfort is further enhanced by improved color accuracy and automatic pupillary distance adjustment.
- **Sensors:** A sophisticated array of sensors—including depth cameras, inertial measurement units (IMUs), ambient light sensors, and an array of microphones—enable comprehensive spatial awareness, environment mapping, and voice interaction.
- **Hand and Eye Tracking:** HoloLens 2 features fully articulated hand tracking with direct manipulation of virtual objects, as well as real-time eye tracking, supporting natural and expressive input modalities.
- **Processing and Connectivity:** Powered by a custom Microsoft Holographic Processing Unit (HPU) alongside a Qualcomm Snapdragon 850 compute platform, the device operates independently, with built-in Wi-Fi, Bluetooth, and enterprise-grade security.

3.4.3 Interaction

One of the defining characteristics of HoloLens 2 is its commitment to natural user interaction. Users engage with holograms through a combination of gaze, gesture, voice, and spatial mapping:

- **Direct Manipulation:** Articulated hand tracking enables users to touch, grasp, and manipulate holographic content intuitively, closely mirroring real-world interactions.

- **Gaze and Dwell:** Eye tracking supports gaze-based targeting and context-aware UI responses.
- **Voice Commands:** Integration with Microsoft's speech recognition engine allows for hands-free control and conversational interfaces.
- **Spatial Awareness:** Real-time environmental mapping anchors digital content to physical surfaces, supporting persistent, context-aware experiences.

3.4.4 Development Ecosystem

The HoloLens 2 ecosystem is centered around open standards and widely adopted development platforms. Unity, together with toolkits like the Mixed Reality Toolkit (MRTK), serves as the primary framework for application development. Developers can access device features via C# scripting, the Universal Windows Platform (UWP), and a suite of APIs for spatial anchors, hand and eye tracking, networking, and sensor data.

Deployment and debugging are streamlined through integrated tools, device portals, and emulators. Continuous updates to the Windows Holographic operating system ensure ongoing support for new features, enhanced performance, and security updates.

3.4.5 Research and Industrial Applications

HoloLens 2 is deployed across diverse domains, including manufacturing, healthcare, education, architecture, and maritime navigation. Its ability to seamlessly blend digital and physical environments enables use cases such as:

- **Remote Collaboration:** Facilitating real-time guidance and support between field workers and remote experts through shared holographic content.
- **Training and Simulation:** Delivering interactive procedural training, equipment maintenance walkthroughs, and immersive simulations.
- **Data Visualization:** Overlaying complex data, schematics, or navigational aids directly onto the relevant physical context.
- **Prototyping and Research:** Serving as a testbed for new interaction techniques, spatial algorithms, and user studies in spatial computing.

3.4.6 Summary

The Microsoft HoloLens 2 exemplifies the forefront of augmented reality hardware, combining sophisticated sensing, natural interaction, and robust computational power within a wearable device. Its synergy with Unity and toolkits like MRTK 3 empowers developers and researchers to build advanced spatial computing applications, bridging digital and physical worlds for practical, industrial, and scientific innovation.

Specific Role Within This Thesis

In this thesis, the HoloLens 2 served as the core device for creating an AR-based maritime navigation and decision-support system. Its hand and eye tracking features were pivotal in designing an intuitive interface, enabling crew members to

visualize and interact with real-time navigational data while keeping their hands free for vessel controls. Additionally, the headset's spatial mapping facilitated accurate placement of hazard markers and navigational overlays, allowing mariners to receive context-specific information and respond quickly to evolving sea conditions.

3.5 Mapbox: A Platform for Custom Mapping and Geospatial Visualization

Mapbox is a powerful platform and suite of developer tools for the creation, customization, and deployment of interactive maps and spatial visualizations. Founded in 2010, Mapbox has become a leading provider of cloud-based geospatial services, enabling developers to build location-aware applications across web, mobile, and extended reality (XR) platforms. By offering highly customizable mapping APIs, robust data services, and cross-platform SDKs, Mapbox supports a wide range of use cases in navigation, logistics, urban planning, research, and beyond.

FIGURE 3.5: The current Mapbox word-mark (2025).

3.5.1 Core Services and Capabilities

At its core, Mapbox delivers dynamic map rendering and geospatial data access through a set of cloud-hosted APIs and SDKs. Key services and features include:

- **Mapbox Maps API:** Provides on-demand rendering of vector and raster maps with fully customizable styling, layer composition, and interactive features. Developers can design maps tailored to specific application needs using the Mapbox Studio web interface.
- **Geocoding and Directions:** Mapbox offers RESTful APIs for address lookup, reverse geocoding, route planning, and turn-by-turn navigation, supporting both real-time and batch processing scenarios.
- **Data Visualization:** The platform supports the overlay of custom data—such as points of interest (POIs), routes, heatmaps, and polygons—on top of base maps, enabling rich spatial analysis and interactive exploration.

- **Real-Time Data Integration:** Mapbox is designed to handle dynamic data sources, allowing for the real-time visualization of moving objects, traffic conditions, and environmental sensor feeds.

3.5.2 Integration with Unity and AR Applications

Recognizing the importance of immersive and interactive spatial experiences, Mapbox provides a dedicated Unity SDK that bridges advanced mapping capabilities with real-time 3D engines. This SDK enables developers to:

- Render interactive maps and terrain directly within Unity scenes, supporting both 2D and 3D perspectives.
- Integrate real-world geospatial data, including elevation, satellite imagery, vector features, and custom overlays, with the Unity physics and rendering systems.
- Attach and animate virtual objects—such as vehicles, avatars, or annotations—based on GPS coordinates or live data feeds.
- Combine AR frameworks (such as Unity's AR Foundation or MRTK) with Mapbox to anchor digital content to specific geographic locations in the physical world.

This integration is particularly valuable for research and industrial applications that require accurate spatial context, such as navigation aids, environmental monitoring, urban simulation, and maritime operations.

3.5.3 Customization and Data Sources

Mapbox empowers developers to create visually distinct and functionally rich maps. Through the Mapbox Studio editor, users can design map styles, select or exclude data layers, adjust color schemes, and import custom datasets. Maps can incorporate a variety of public and private data sources, including OpenStreetMap, satellite imagery, proprietary GIS layers, and real-time telemetry.

Custom map styles and data overlays can be exported and integrated into Unity applications via the Mapbox SDK, providing a consistent visual language across devices and platforms.

Specific Role Within This Thesis

In the prototype presented here, Mapbox supplies the **heads-up minimap** that complements the world-anchored holograms. Through the `Mapbox Maps SDK for Unity`, a frameless 2-D chart is anchored just above the bridge windshield in the HoloLens 2 view, allowing the captain to gain instant geographic context—own-ship position, surrounding traffic, and hazard around—without shifting focus.

Platform Integration Points

- **Basemap Streaming** A minimalist, low-ink basemap is streamed on demand, conveying only the reference geometry needed for safe conning.
- **Operational Overlays (Points, Areas, Routes)** Custom vector sources add live AIS contacts, static hazards such as reefs and wrecks, danger areas, and

server-generated recommended tracks or pilot waypoints (polylines). Styling matches the colour and icon set used in the forward AR scene for immediate visual correlation.

- **Sub-Second Refresh** GPS and AIS updates traverse a Kafka→WebSocket pipeline; freshly published geometry is written to a Mapbox tileset and re-rendered on the headset in well under one second, keeping the minimap in lock-step with the holographic overlay.

Overall, Mapbox's robust functionality and ease of integration with Unity made it a valuable asset for crafting a real-time map interface within the HoloLens 2 AR application.

3.5.4 Limitations and Practical Considerations

Despite their strengths, the selected platforms present several practical limitations. For instance, HoloLens 2's field of view and display brightness can constrain usability in direct sunlight. MRTK 3, while comprehensive, may require significant customization for maritime-specific interactions. Mapbox's base layers, though rich, may lack up-to-date maritime-specific overlays without custom data integration. These factors influenced both the design of visualizations and the calibration methods implemented in the thesis system.

3.6 Development Workflow and Toolchain

Development was managed using Git for version control, with iterative prototyping and field testing cycles. The Unity Editor was configured for Universal Windows Platform (UWP) builds, ensuring direct deployment to HoloLens 2 hardware. MRTK 3 packages were managed through Unity's Package Manager, while Mapbox SDK updates were tracked separately. Collaborative feedback was facilitated via SURREAL lab meetings, with code reviews and integration checks prior to major deployments or field trials.

Chapter 4

System Architecture and Implementation

4.1 Introduction

This chapter presents the implementation of the head-worn augmented reality bridge assistant for real-time ship navigation and collision avoidance. Developed in Unity and leveraging the Mixed Reality Toolkit 3 (MRTK3) on Microsoft HoloLens 2, the application integrates live GPS, AIS, and predictive trajectory data to overlay world-anchored navigation cues directly into the captain's forward view, complementing traditional ECDIS and radar systems.

The focus here is the client-side HoloLens software. Subsequent sections describe how the Unity/MRTK3 platform establishes WebSocket data connections, processes real-time maritime data streams, and renders dynamic AR guidance cues.

Special attention is given to interface design suited for the bridge environment. The system enables officers to intuitively visualize essential elements—such as nearby points of interest (POIs), vessel telemetry panels, and restricted-area alerts—while keeping hands free for vessel operation.

Finally, the chapter details the Unity implementation workflow, from interaction logic to uncertainty cone visualization, and discusses technical solutions adopted to address maritime-specific challenges, such as bridge-window glare, reduced depth cues, and GPS instability.

Topics addressed in this chapter:

- **System Architecture and Data Flow**: overview of the multi-tier system—Android data source, server backend, and HoloLens AR client—and how navigational data flows through the pipeline.
- **Real-Time Unity–Server Communication**: implementation of a persistent WebSocket link, message parsing, and main-thread data dispatch for continuous scenario updates on the headset.
- **User-Interface Layout and Interaction**: menu hierarchy, hand- and gaze-triggered controls, and interface ergonomics tailored for bridge operations using MRTK3.
- **Calibration Workflow**: a multi-stage process (true-north orientation, horizon fit, AR-window framing, QR-code assisted anchoring) enabling reliable spatial alignment—even on moving or non-parallel bridges.

FIGURE 4.1: **Prototype testing during the second harbour trial**

- **AR Window**: user-defined, window-aligned portals that restrict AR content to views "through the glass," minimizing distraction and respecting situational awareness on the bridge.
- **Points of Interest (POIs)**: symbol set based on official nautical charts—reefs, lights, buoys—updated in real time from AIS and server data.
- **Route & Hazard Visualization**: dynamic rendering of predicted vessel tracks, restricted areas, and cone-of-uncertainty overlays that widen with prediction horizon.
- **Interactive Mini-Map (Mapbox)**: a frameless, always-accessible 2D chart anchored above the bridge windshield, providing instant context and hazard awareness without head-down time.

4.2 System Architecture

The solution is organised around three runtime tiers (Fig. 4.2): an Android phone that streams own-ship telemetry, a Node/Kafka back-end that aggregates external feeds, and the HoloLens 2 client that renders navigation holograms.

- **Android Application**: A bridge-mounted smartphone captures the headset wearer's GPS stream and pushes latitude/longitude fixes over WebSocket.
- **Edge Server**: a Dockerised stack (Kafka + Node.js) ingests the phone feed, live AIS messages, and route-forecast ensembles, then broadcasts a compact JSON scene graph to all connected headsets with sub-second latency.
- **HoloLens 2 Client**: the Unity application converts WGS-84 lat/lon into local Unity space, draws POIs, route lines, restricted-area polygons, and widening cones of uncertainty in synchrony with the minimap layer.
- **Mapbox Services**: raster tiles for the minimap plus a custom vector tileset holding mission overlays streamed on demand to the headset.

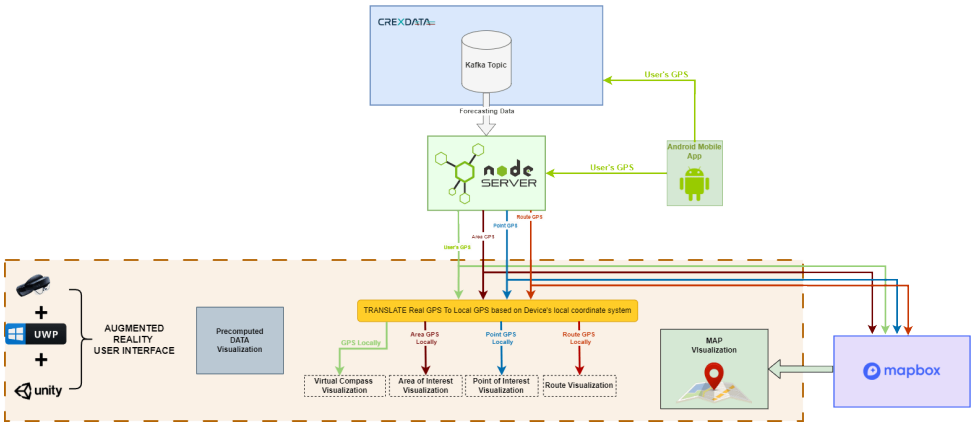


FIGURE 4.2: Data flow from bridge sensors to HoloLens 2

This architecture sustains sub-second update times and keeps all spatial layers—3-D holograms and 2-D minimap—co-registered, enabling the bridge team to make rapid, informed decisions.

4.3 Data Structure and Field Overview

All navigation, hazard, and situational data is delivered to the HoloLens client as JSON-formatted messages streamed from Kafka, via a Node.js server. Each message contains a standardized set of fields, which are parsed by the AR client and used to instantiate Points of Interest, route waypoints, and contextual overlays.

A summary of the primary fields included in each message is presented in Table 4.1.

TABLE 4.1: **Summary of incoming JSON fields used by the AR application.**

Field	Type or Units	Description
ship1_mmsi	string	MMSI of vessel 1 for the collision avoidance path
ship2_mmsi	string	MMSI of vessel 2 for the collision avoidance path
ship1.TARGET_SPEED	knots	Target speed for vessel 1 as defined by the collision-avoidance algorithm to resolve the path-planning problem for vessel 1
ship2.TARGET_SPEED	knots	Target speed for vessel 2 as defined by the collision-avoidance algorithm to resolve the path-planning problem for vessel 2
ship1_course	degrees	Course for vessel 1 with respect to the collision-avoidance path

Continued on next page

Table 4.1 – continued from previous page

Field	Type or Units	Description
ship2_course	degrees	Course for vessel 2 with respect to the collision-avoidance path
ship1_traj_lon	WGS-84 degrees (decimal)	Current longitude coordinate of vessel 1
ship1_traj_lat	WGS-84 degrees (decimal)	Current latitude coordinate of vessel 1
ship2_traj_lon	WGS-84 degrees (decimal)	Current longitude coordinate of vessel 2
ship2_traj_lat	WGS-84 degrees (decimal)	Current latitude coordinate of vessel 2
linestring1	Lon/Lat/t triple	Linestring of the collision-avoidance path solution for vessel 1 (longitude, latitude, timestamp [s])
linestring2	Lon/Lat/t triple	Linestring of the collision-avoidance path solution for vessel 2 (longitude, latitude, timestamp [s])
collision_lon	WGS-84 degrees (decimal)	Collision point: longitude
collision_lat	WGS-84 degrees (decimal)	Collision point: latitude
timestamp	milliseconds	Current time (Unix epoch ms)
collision_id	UUID4	Collision event identifier

This unified data structure is used across all real-time overlays in the system, ensuring consistency among POIs, route visualizations, and contextual alerts. Each field is mapped to its corresponding Unity object or UI component as described in the subsequent sections.

4.4 Real-Time Unity–Server Communication

4.4.1 Introduction to the Unity–Server Communication Model

In order to support real-time maritime situational awareness, the Unity client (running on the AR device) maintains a persistent bidirectional connection with a backend server. This always-on link is achieved via a WebSocket, which enables the server to push timely navigational updates (such as new or updated vessel routes) to the client as soon as they are available, rather than relying on periodic polling. The result is a low-latency data pipeline: the server forwards streaming data (originating from MarineTraffic AIS updates and collision-avoidance computations) directly to the Unity application, which in turn can instantly reflect those changes in the augmented reality scene. This design ensures that the AR system's representation of vessel routes and potential collision information is continually synchronized with the latest predictions from the backend.

From an architectural standpoint, the Unity–server communication model consists of a central publish/subscribe mechanism and a lightweight messaging protocol.

The backend collects and processes route-related data (e.g., via a Kafka-based pipeline as described in Section 5.3) and broadcasts standardized JSON messages over WebSocket to all connected AR clients. Each message encapsulates all the information needed for the client to update its internal state and visuals for a particular scenario. On the Unity side, a dedicated networking component listens for these incoming messages and handles them in an event-driven manner. The use of WebSockets means the connection remains open across the session, allowing a continuous flow of data (multiple messages per second if necessary) without the overhead of repeated HTTP requests. This persistent connection is particularly crucial for the marine AR application: it must respond promptly to dynamic events (such as a vessel's course change or a newly detected collision risk) and update the holographic content accordingly, thereby improving the timeliness and relevance of decision-support information presented to the user.

In summary, the Unity–server communication model lays the groundwork for real-time data integration by providing a steady stream of structured messages. The following subsections detail how this connection is established and maintained in the Unity client, how incoming messages are parsed and dispatched, and how a specific message carrying route information (the MarineTraffic_route update) is processed to update Unity's internal models and visuals.

4.4.2 WebSocket Initialization and Connection Lifecycle

The Unity client establishes the connection by calling the `ConnectToServer()` method when the application starts up. This method creates a new WebSocket instance pointing to the server's URI (e.g., `ws://:8080`) and initiates an asynchronous connection attempt. As part of the initialization, event handler callbacks are registered for key WebSocket events: an `OnOpen` event for when the connection is successfully opened, an `OnMessage` event for incoming data, and an `OnClose` event for when the connection closes or is disrupted. By setting up these handlers in advance, the Unity client can react immediately to changes in connection state or incoming messages.

```
void ConnectToServer()
{
    // Initialize the WebSocket connection to the specified URL.
    ws = new WebSocket("ws://" + ip + ":8080");
    Debug.Log("Connecting to server...");
    // Subscribe to the OnOpen event. This event is triggered when the connection is opened.
    ws.OnOpen += OnConnectionOpened;

    // Subscribe to the OnMessage event. This event is triggered when a message is received from the server.
    ws.OnMessage += OnMessageReceived;

    // Subscribe to the OnClose event. This event is triggered when the connection is closed.
    ws.OnClose += OnConnectionClosed;

    // Connect to the server asynchronously.
    ws.ConnectAsync();
}
```

FIGURE 4.3: **WebSocket connection initialization and event binding in Unity.** The connection is established to the server primes URL, setting callbacks for open, message, and close events to facilitate real-time communication.

Once the handshake with the server completes, the `OnConnectionOpened` callback is invoked. In this project's implementation, that event triggers a registration handshake from the client's side. The Unity application sends a small JSON-formatted

registration message identifying itself (including fields such as a device type, use-case, and a unique client ID). This notifies the server of a new client connection and provides any necessary context (for example, that this client is a HoloLens device for a marine navigation use case). The server may respond with a confirmation (e.g., a message of type "registrationResult") acknowledging the registration and providing any additional info like an assigned client identifier. After this exchange, the WebSocket connection is fully ready for data transfer. At this point, if this is not the first connection attempt (say the client reconnected after a drop), the client may also send a special message or command to inform the server it has rejoined (in the code, for example, sending a "Come" or "Recome" string based on whether it's a fresh connection or a reconnection). These signals can prompt the server to resend any initial data or simply log the reconnection; the exact behavior depends on the server's logic.

```
void OnConnectionOpened(object sender, EventArgs e)
{
    // Create a registration message
    var registrationMessage = new
    {
        type = "registration",
        deviceType = "unity",
        useCase = "marine",
        clientId = uniqueID
    };

    // Convert to JSON and send
    string jsonMessage = JsonConvert.SerializeObject(registrationMessage);
    SendMessageToServer(jsonMessage);

    if (attempt > 0)
    {
        Debug.Log("Reconnected to the server.");
        SendMessageToServer("Recome");
    }
    else
    {
        Debug.Log("Connected to the server");
        SendMessageToServer("Come");
    }

    attempt = 0;

    UnityMainThreadDispatcher.Instance.Enqueue(() =>

OnConnectionOpenedEvent?.Invoke(this, e);
```

FIGURE 4.4: **OnConnectionOpened handler logic.** Upon successful connection, the Unity client sends a registration message and signals readiness for receiving data updates.

During operation, the WebSocket link remains open, carrying a stream of messages from the server to the Unity client. If the connection is interrupted or closed for any reason, the `OnConnectionClosed` event is triggered on the client side. The Unity handler for this event logs the closure and initiates a reconnection strategy. In our

```
async void OnConnectionClosed(object sender, CloseEventArgs e)
{
    Debug.Log($"Connection closed. Attempting to reconnect. Total attempts: {++attempt}");

    UnityMainThreadDispatcher.Instance.Enqueue(() =>
    {
        OnConnectionClosedEvent?.Invoke(this, e);
    });

    // Wait for a delay before attempting to reconnect. The delay increases with each failed attempt.
    int delay = 3000;
    await Task.Delay(delay);

    // Check if the application is quitting before attempting to reconnect.
    if (!isApplicationQuitting)
        ConnectToServer();
}
```

FIGURE 4.5: **OnConnectionClosed handler logic.** This method manages automatic reconnection attempts following disconnections, maintaining continuous real-time communication.

implementation, the client automatically attempts to reconnect after a brief delay (on the order of a few seconds). Each reconnection attempt increments a counter (to keep track of how many times reconnection was tried) and uses a fixed delay before calling `ConnectToServer()` again. This simple retry loop continues until a connection is re-established or the application is terminated. This way, transient network issues are handled gracefully without requiring user intervention: the system will restore the real-time data feed as soon as network connectivity returns. If the application is shutting down (for example, the user exits the app), a flag is set to prevent any further reconnection attempts. In that case, the client will cleanly close the WebSocket (via an `OnApplicationQuit` routine that unsubscribes from events and closes the socket) to ensure no resources are leaked. This life-cycle management approach (connect $\rightarrow$ open $\rightarrow$ message exchange $\rightarrow$ close $\rightarrow$ reconnect if needed) provides robustness for continuous operation, even in less reliable network conditions.

4.4.3 Incoming Message Parsing and Dispatch

All incoming data from the server arrives as text frames on the WebSocket, which the Unity client's `OnMessage` handler receives. The client employs a central message-processing function (e.g., `HandleMessage()`) to interpret these incoming strings. The first step in handling a new message is to determine if it's in JSON format, as expected for our navigational updates. The handler attempts to parse the text into a JSON object (using the Newtonsoft `JObject` parser in this implementation). If parsing succeeds, the client then looks for a special field, typically called "`type`", within the JSON structure. This `type` field denotes the kind of message and dictates which code path should handle it.

Using the message type as a key, the client dispatches the message to the appropriate routine. In practice, this is implemented with a conditional or switch-case construct that checks the `messageType` string against known values. For example, if `messageType` is "`gps`", the code will invoke the GPS handler (the routine responsible for updating the user's location). If the type is "`MarineTraffic_route`" (the designated type for incoming route updates in the collision-avoidance use case), the route handling routine is called. This modular design allows each message category to be processed by a dedicated function that knows how to deal with that specific data structure. It also makes the system extensible; new message types (for instance,

future additions like ‘‘`weatherAlert`’’ or other POI updates) can be integrated by adding a new case and handler without altering the existing logic for GPS or route messages.

```
void HandleMessage(string message)
{
    try
    {
        // Try to parse as JSON first
        JObject jsonObject = JObject.Parse(message);
        string messageType = jsonObject["type"]?.ToString();

        if (messageType != null)
        {
            // Handle JSON messages by type
            switch (messageType)
            {
                case "gps":
                    HandleGpsMessage(jsonObject);
                    break;

                case "MarineTraffic_route":
                    HandleRouteMessage(jsonObject);
                    break;
```

FIGURE 4.6: **Message parsing and dispatching logic.** Incoming JSON messages are parsed to identify their type and routed to appropriate handlers for GPS updates and MarineTraffic route data processing.

Unrecognized message types are safely handled by logging a warning or error, ensuring that unexpected data doesn’t cause a crash — the client will simply report an unknown message and ignore it. Additionally, if the incoming text is not valid JSON at all (which might indicate either legacy message formats or corrupt data), the system can fall back to an alternative parsing routine or skip the message. In our application, a legacy handler exists for backward compatibility, but all primary navigation functions rely on JSON messages with well-defined types. In summary, this parsing and dispatch system acts as a router within the Unity client: it examines each incoming message’s label and delegates it to the correct processing function, thereby cleanly separating the logic for different data flows.

4.4.4 Route Message Processing (`HandleRouteMessage`)

One of the core message types the client handles is the navigational route update. These messages, labeled with a type such as ‘‘`MarineTraffic_route`’’, contain the data needed to visualize recommended routes and potential conflicts (like collision warnings) on the AR map. The `HandleRouteMessage` routine is responsible for processing this incoming route payload and preparing it for visualization. Its operation can be summarized in several key steps:

- **Clearing previous route data:** As a first step, the client clears out any existing route visualization from a prior update. This involves removing or resetting previously spawned route markers, lines, or icons on the map that were associated with the old route. By clearing stale data, the system ensures that outdated routes do not clutter the display when a new route update arrives.
- **Validating and accessing the payload:** The route message is expected to contain a nested JSON object (often under a field named `"payload"`) that holds all the route details. The handler checks that this payload is present and well-formed. If the payload section is missing or improperly structured, an error is logged and the routine exits early, since there is no valid route data to process. This validation guards against malformed messages and prevents null-reference errors in subsequent steps.
- **Extracting route details:** Once the payload is confirmed, the handler parses out the key fields needed for route visualization. In the maritime scenario, for example, the payload includes unique identifiers for the vessels involved (such as MMSI numbers for "ship 1" and "ship 2"), their current or predicted speeds and courses, the coordinates of a potential collision point (latitude and longitude), and sequences of coordinates that form each vessel's trajectory. Specifically, the message provides lists of latitude and longitude points for the recommended route of the user's vessel (`ship1_traj_lat` and `ship1_traj_lon`) and for the route of a second vessel (`ship2_traj_lat` and `ship2_traj_lon`) that may intersect with it. It may also include pre-formatted route geometry strings (e.g., `linestring1`, `linestring2` which describe the paths with timestamps and a collision identifier or timestamp for reference. The handler systematically retrieves these fields from the JSON structure, converting them into Unity-friendly data types. For instance, numeric values like speed or coordinates are read into floats/doubles, and coordinate arrays are parsed into `List<double>` collections or similar.
- **Preparing data for visualization:** After extraction, the new route information is passed downstream to the map visualization subsystem. Rather than directly drawing anything in the `HandleRouteMessage` function, the handler typically creates or updates data objects that the rendering code will use. In our case, it instantiates or populates data structures (e.g., lists of GPS waypoints) that represent the two vessels' routes and the collision point. These are then fed to routines that handle drawing the route lines and placing markers on the map. For example, the application's route visualizer will use the lists of coordinates to spawn route point markers for each waypoint and draw polyline renderers connecting them in the Unity scene (one line for the own ship's route and another for the other ship's route). Similarly, if a collision point was provided, a special marker can be placed at that location. Rendering details are not discussed in this section; it is sufficient to note that `HandleRouteMessage` transfers the parsed route data to the components responsible for updating the 3D map, thereby triggering an update of the AR display to show the new recommended route and any warning indicators.

By following these steps, the Unity client ensures that each incoming route message results in a complete refresh of the route depiction in the user's view. Importantly, the approach is idempotent: the full set of route parameters is contained in each message (as noted in Section 5.3), so the client does not need to diff against previous data — it simply replaces the old route with the new one each time. This design

```
public void HandleRouteMessage(JObject jsonObject)
{
    try
    {
        spawner.clearSpawnedObjectsByType(POIType.MyRoutePoint);
        spawner.clearSpawnedObjectsByType(POIType.OtherRoutePoint);
        spawner.clearSpawnedObjectsByType(POIType.CollisionPoint);

        JObject payload = (JObject)jsonObject["payload"];
        if (payload == null)
        {
            Debug.LogError("Spawn POI message has no payload");
            return;
        }

        string ship1Mmsi = payload["ship1_mmsi"]?.ToString();
        string ship2Mmsi = payload["ship2_mmsi"]?.ToString();

        // Extracting other scalar values
        double? ship1Speed = payload["ship1_speed"]?.Value<double>();
        double? ship2Speed = payload["ship2_speed"]?.Value<double>();
        double? ship1Course = payload["ship1_course"]?.Value<double>();
        double? ship2Course = payload["ship2_course"]?.Value<double>();
        double? collisionLon = payload["collision_lon"]?.Value<double>();
        double? collisionLat = payload["collision_lat"]?.Value<double>();
        long? timestamp = payload["timestamp"]?.Value<long>();
        string collisionId = payload["collision_id"]?.ToString();
        string linestring1 = payload["LINESTRING1"]?.ToString();
        string linestring2 = payload["LINESTRING2"]?.ToString();

        // Extracting arrays
        List<double> ship1TrajLon = payload["ship1_traj_lon"]?.ToObject<List<double>>() ?? new List<double>();
        List<double> ship1TrajLat = payload["ship1_traj_lat"]?.ToObject<List<double>>() ?? new List<double>();
        List<double> ship2TrajLon = payload["ship2_traj_lon"]?.ToObject<List<double>>() ?? new List<double>();
        List<double> ship2TrajLat = payload["ship2_traj_lat"]?.ToObject<List<double>>() ?? new List<double>();

        // You can now use these variables. For example, to log them:
        Debug.Log($"Ship 1 MMSI: {ship1Mmsi}, Speed: {ship1Speed}, Course: {ship1Course}");
        Debug.Log($"Ship 2 MMSI: {ship2Mmsi}, Speed: {ship2Speed}, Course: {ship2Course}");
        Debug.Log($"Collision Lon: {collisionLon}, Lat: {collisionLat}, Timestamp: {timestamp}, ID: {collisionId}");
        Debug.Log($"Linestring 1: {linestring1}");
        Debug.Log($"Linestring 2: {linestring2}");

        Debug.Log($"Ship 1 Trajectory Longitudes Count: {ship1TrajLon.Count}");
        if (ship1TrajLon.Count > 0)
        {
            Debug.Log($"First Ship 1 Traj Lon: {ship1TrajLon[0]}");
        }
        Debug.Log($"Ship 2 Trajectory Longitudes Count: {ship2TrajLon.Count}");
        if (ship2TrajLon.Count > 0)
        {
            Debug.Log($"First Ship 2 Traj Lon: {ship2TrajLon[0]}");
        }
```

FIGURE 4.7: **MarineTraffic_route message parsing logic.** The handler validates incoming JSON data, extracting essential information to prepare for route visualization updates.

is robust to packet loss or reordering; even if a message were missed, the next one carries all necessary information to visualize the current situation. Figure 4.7 shows an example of the outcome after processing a route update: the HoloLens interface displays the updated paths for both vessels and a highlighted collision point, all of which correspond to the data extracted by `HandleRouteMessage`.

4.4.5 GPS Message Processing (`HandleGpsMessage`)

Another fundamental message type is the GPS update, which provides the user's current geographic location. Whenever the server pushes a new location message (typed as ''gps'' in our system), the Unity client invokes the `HandleGpsMessage` routine to update the user's position and the map alignment. The processing of a GPS message is straightforward and focuses on integrating the new coordinates into the AR map's state:

- **Parsing the coordinates:** The handler extracts the latitude and longitude values from the incoming JSON message (typically found in fields like ''`lat`'' and ''`lon`''). These values represent the user's latest position as measured by the mobile GPS sensor. In some cases, additional metadata such as heading or speed might accompany the location, but the primary data needed for positioning on the map are the coordinates themselves.
- **Map alignment (reference update):** If this is the first GPS fix received (i.e., the map's geographic frame of reference has not yet been established), the system uses it to set up the map alignment. In our implementation, the first latitude/longitude received is used to define reference coordinates for the scene. This step is critical for AR applications: it anchors the virtual map to the real-world location context. By updating a reference point (often done via a function like `UpdateReferenceCoordinates(latitude, longitude)` in the supporting map or location manager), the Unity application calibrates the conversion between geographic coordinates and the local Unity coordinate system. After this one-time setup, subsequent GPS points can be translated into the Unity world with respect to this reference.
- **Re-centering the map view:** The next step is to ensure the map is centered on the user's new location. The client calls `mapManager.SetCoordinates()` with the latest latitude and longitude. Internally, this leverages Mapbox map functions (specifically, setting the map's center latitude/longitude) to pan or move the map such that the user's position is at the center (or another defined position in the view). Re-centering the map on each update ensures that the user's avatar will remain visible on the display, rather than wandering off-screen as the user moves
- **Updating the user's avatar position:** Finally, the handler updates the Unity representation of the user's location. The `mapManager.UpdatePlayerLocation()` function is invoked with the new latitude/longitude pair. This routine converts the geographic coordinates into Unity world coordinates (using Mapbox's utilities such as `GeoToWorldPosition()`, which translates a latitude/longitude into a position in the scene's coordinate space)

Through these steps, each incoming GPS message causes the AR map to seamlessly reflect the user's latest location. The combination of re-centering the map and moving the avatar ensures that the user's representation and the map content stay in sync with the real world at all times. For example, if the user (with the HoloLens) walks a few meters in the physical environment, the mobile app's GPS will report new coordinates to the server, which then relays them to the HoloLens; the Unity client, upon processing the update, will shift the map and the user icon accordingly. This real-time feedback loop creates the illusion that the AR map is fixed to the Earth, with the user moving through it just as they move through the real world.

Overall, the Unity–server communication workflow described in this section enables live navigational data to drive the AR display. The WebSocket-based approach provides a low-latency, event-driven channel for updates, while the structured message handling (dispatching to `HandleRouteMessage`, `HandleGpsMessage`, etc.) organizes the client's response to different data types. By promptly clearing old data and integrating new information into the map — whether updating a vessel's course or the user's own location — the system maintains an accurate and up-to-date representation of the maritime environment. This real-time synchronization between server

```
void HandleGpsMessage(JObject jsonObject)
{
    try
    {
        double latitude = jsonObject["lat"].Value<double>();
        double longitude = jsonObject["lon"].Value<double>();

        Debug.Log($"Received GPS: Lat {latitude}, Long {longitude}");

        try
        {
            if (spawner != null && !spawner.referenceCoordinatesSet)
            {
                spawner.UpdateReferenceCoordinates(latitude, longitude);
            }
        }
        catch (Exception e)
        {
            Debug.LogError("Error updating reference coordinates: " + e.Message);
        }

        mapManager.SetCoordinates(new Vector2d(latitude, longitude));
        mapManager.UpdatePlayerLocation(new Vector2d(latitude, longitude));
    }
    catch (Exception ex)
    {
        Debug.LogError("Error handling GPS message: " + ex.Message);
    }
}
```

FIGURE 4.8: **GPS message handling and map update logic.** Coordinates received via GPS messages trigger updates to the user primes location and map alignment in real-time.

data and the Unity application is crucial for delivering an effective AR navigation experience, allowing the user to make informed decisions based on the most current information available.

With a reliable data pipeline in place, the next focus is on how the HoloLens client presents this information to the user through an intuitive AR interface.

4.5 User Interface (UI)

The user-interface (UI) concept for the head-worn AR prototype was guided by one overarching theme: Captains must be able to find what they need in seconds, not minutes. The layout therefore favours clarity over decoration and minimises on-screen density so that essential data in the augmented and physical reality remain visible without distraction. Two major design drivers shaped the solution:

- **Optimized Spatial Layout:** The interface was tailored to the constrained visual space of the HoloLens 2, ensuring that digital overlays remained within comfortable viewing zones without obstructing critical sightlines or overwhelming the user's perception.
- **Accessibility and Responsiveness:** Designing an interface that remains intuitive and swiftly usable, even under high-pressure conditions. Emphasis was

placed on rapid navigation and seamless interaction to make the application effective for real-life conditions.

Initial rounds of prototype testing revealed that, while the general layout was functional, users encountered friction with the original calibration UIs —challenges that persisted even among those experienced with mixed-reality systems. Feedback from these early trials led to targeted refinements aimed at improving usability and responsiveness. Several entirely new UI components were introduced in subsequent iterations, streamlining interaction flows and enabling features that were absent in the initial build, ultimately resulting in a more capable and user-friendly system.

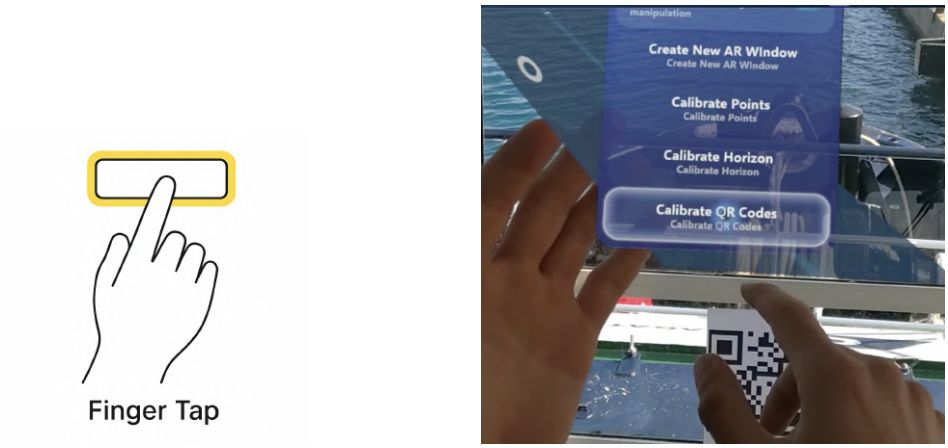
4.5.1 Interaction Methods

Interaction methods constitute the set of human–computer communication channels—hand poses, direct touch, eye focus, speech, and their combinations—through which intentions are conveyed to an extended-reality (XR) system. Careful selection of these channels is critical: it determines input throughput, cognitive demand, resilience to environmental constraints, and ultimately the operator's effectiveness when primary tasks compete for attention.

The system harnesses the full sensor suite of the Microsoft HoloLens 2, combining *gesture*, *direct-hand*, *gaze*, and *hybrid* interactions. All behaviours were implemented with the Mixed Reality Toolkit 3 (MRTK3) for Unity, which affords millisecond-level tracking accuracy and a coherent visual language across modalities.

Direct–Hand Interaction

Direct-hand interaction emulates a conventional touchscreen: the user intersects the hologram's surface with a fingertip to trigger an event (Fig. 4.9).



(A) (B)

FIGURE 4.9: **Direct-touch interaction. (A)** Schematic of the finger-tap gesture approaching an augmented button. **(B)** HoloLens capture showing successful activation of a list item immediately after contact.

Once the menu is visible, individual items are activated by a solitary finger tap, providing quick one-step access to frequently used commands.

Gesture Interaction

Gestural input interprets mid-air hand poses and trajectories, enabling contact-free command execution.

Presenting an open left palm toward the headset is recognised by MRTK3 as a *menu-summon* posture. After detection, a radial menu is instantiated at a position indexed to the user's hand, remaining within the visual periphery without occluding the scene (Fig. 4.10).

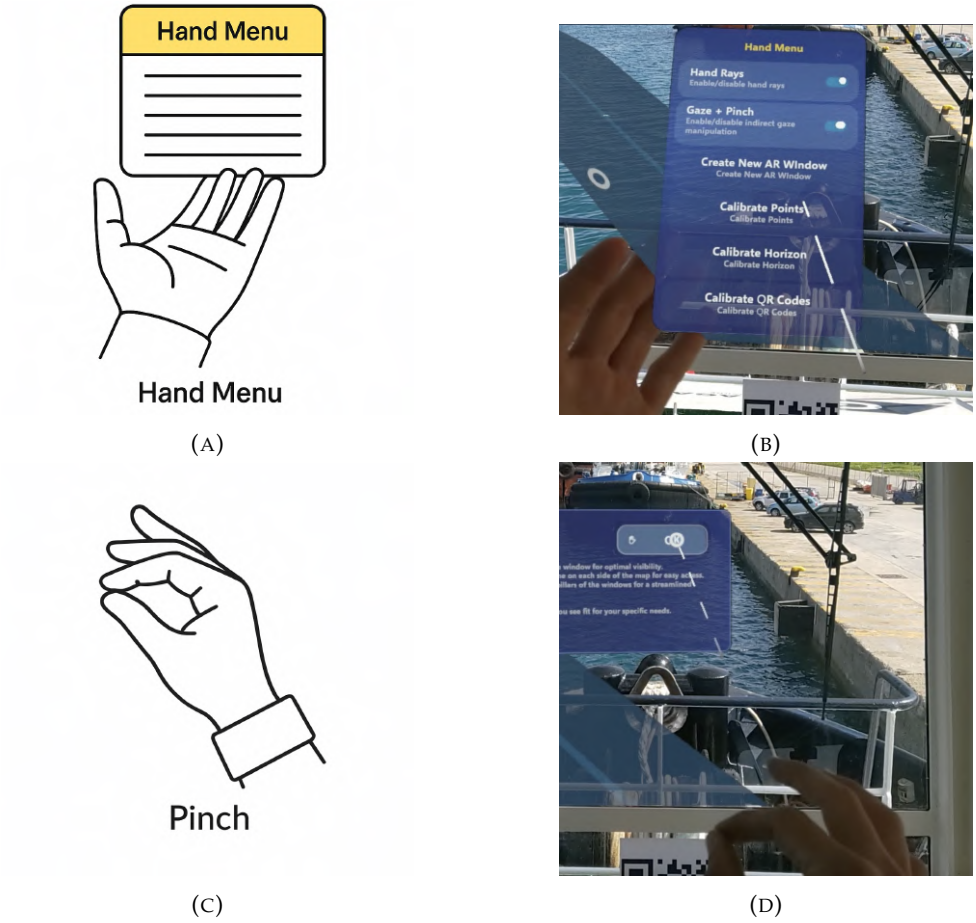


FIGURE 4.10: **Gesture–based UI controls. (A)** Open-palm "hand-menu" pose that spawns the on-hand menu. **(B)** In-device view of the menu tethered to the user's palm. **(C)** Air-pinch gesture used for selection and spatial anchoring. **(D)** Pinch confirmation executed on an in-situ overlay.

Interface elements situated beyond arm's reach are actuated by a *far-interaction Air Tap*—a single pinch performed at a distance—thus eliminating the need for physical proximity. Holographic panels annotated as `Moveable`—for example, the strategic map—respond to MRTK3's *one-hand grab* gesture, permitting users to translate or reposition them at will.

Gaze Interaction

Gaze input deploys the eye- or head-centric pointing vector to furnish hands-free, low-latency targeting (Fig. 4.11).

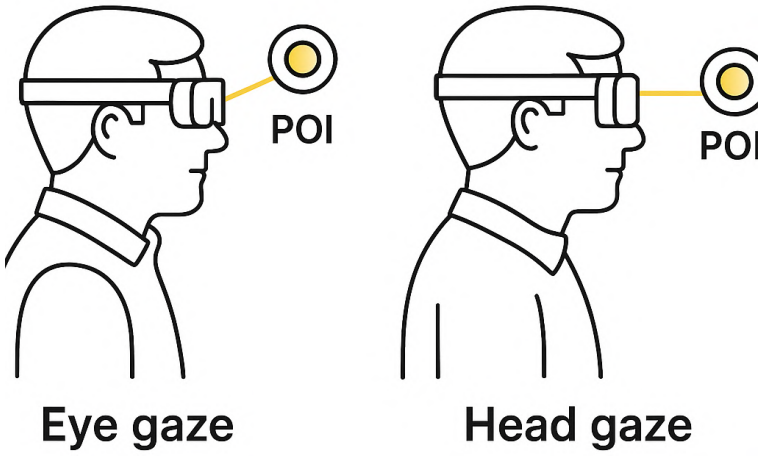


FIGURE 4.11: **Gaze-based pointing.** **(A)** *Eye-centric gaze:* the user keeps the head stationary and moves only the eyes to intersect the Point of Interest (POI). **(B)** *Head-centric gaze:* the user rotates the entire headset until the forward vector aligns with the POI.

MRTK3 continually samples eye orientation and projects a gaze ray into the scene. When this ray intersects an interactive object—such as a point of interest (POI) or a menu item—the framework automatically provides visual affordances (highlight, outline) and begins timing a configurable *dwell* interval. If the user maintains fixation for the required duration, the element is invoked; alternatively, the dwell timer can be bypassed by an explicit confirmation gesture (e.g. Air Tap) to reduce selection latency. This approach yields a robust, hands-free control channel that remains effective even when both hands are occupied with conventional bridge equipment.

Simultaneous Gaze + Gesture

Hybrid interactions intentionally couple two modalities to accomplish tasks that would be cognitively or ergonomically demanding when attempted with a single input modality(Fig. 4.12).

In the present implementation, gaze provides rapid, coarse aiming, while a prolonged Air Pinch supplies deliberate confirmation of a spatial anchor. This compound technique underpins the definition of a "magic window," discussed in detail in Section 4.7, and does so without introducing extra peripherals or menu complexity.

Exclusion of Voice and Handheld Controllers Despite MRTK3's support for voice commands and Bluetooth-enabled controllers, these modalities were intentionally excluded. The acoustic environment on a ship's bridge typically includes constant background noise from ventilation systems, radio communications, and audible alarms, significantly hindering reliable voice recognition. Similarly, handheld controllers were not suitable, as a ship captain must retain freedom of hand movement to manage vessel controls and communication equipment.

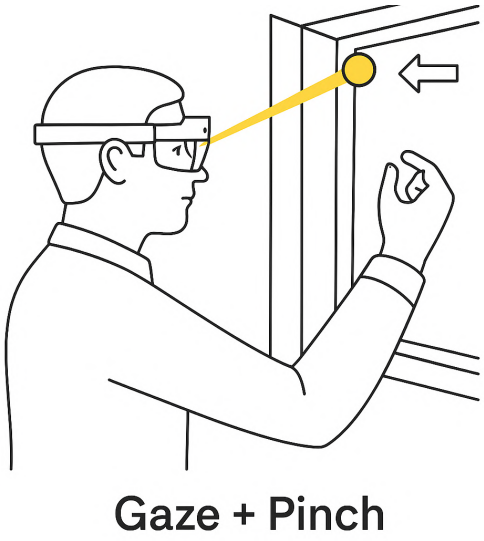


FIGURE 4.12: **Hybrid gaze–gesture anchoring.** The user directs the gaze ray toward a bridge-window corner and confirms the selection with a prolonged pinch, thereby storing the point as a spatial anchor.

By orchestrating these complementary modalities through MRTK3, the interface adapts to the temporal and physical constraints of maritime decision-making, enabling seamless transitions between hands-free and situations where the user's hands are occupied.

4.5.2 Interaction and UI Implementation Using MRTK3

The Mixed Reality Toolkit 3 (MRTK3) was central to the realization of intuitive interaction methods and the implementation of user interface (UI) elements within the augmented reality (AR) application for the Microsoft HoloLens 2. MRTK3 offers an extensive collection of interactive components, enabling effective and precise interactions with both virtual and augmented elements in the user's environment.

Gesture-Based Interactions

The primary gesture-based menu interaction utilised the MRTK3 *Hand Constraint Palm Up* component, which dynamically recognizes the user's palm orientation. Specifically, the component detects an upward-facing palm directed toward the HoloLens camera, triggering the display of a palm-attached menu (Fig. 4.14). Furthermore, the *Solver Handler* component complemented this by continuously anchoring the UI menu to the user's hand, ensuring smooth positional tracking and preventing visual obstruction during interactions (Fig. 4.13).

Additionally, the system incorporated the *Air Pinch* gesture for interactions involving virtual elements placed beyond immediate hand reach. This gesture facilitated accurate selection and confirmation of objects, further enriching the gesture-based interaction possibilities within the AR environment.

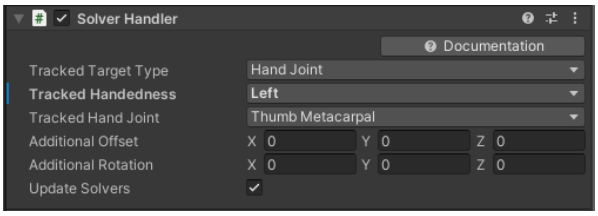


FIGURE 4.13: **Gesture-based menu activation components** The *Hand Constraint Palm Up* component triggers the menu when the user raises their palm toward the HoloLens camera.

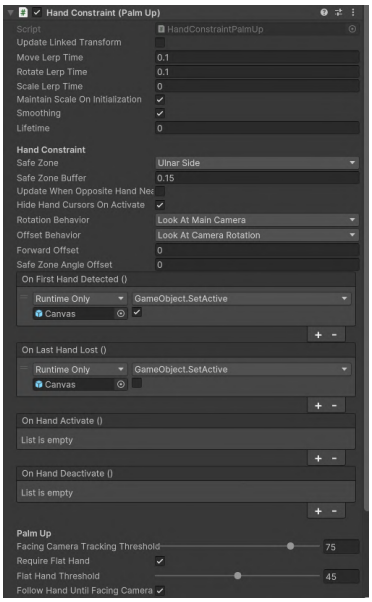


FIGURE 4.14: **Gesture-based menu activation components (B).** The *Solver Handler* maintains the menu's position relative to the hand, ensuring consistent tracking without obstructing the user's view.

Gaze-Based Interactions

Gaze interactions were robustly supported by MRTK3's integrated gaze-tracking capabilities. Users could interact effortlessly with virtual elements, such as Points of Interest (POIs), through intuitive visual targeting. MRTK3 automatically detected when a user's gaze was fixed on interactive elements and provided immediate visual feedback, such as highlighting or outlining the targeted object, enhancing usability even in scenarios where manual input was impractical or undesirable (Fig. 4.15).

Customization of UI Components

MRTK3 provided a diverse library of prefabricated UI elements, including interactive buttons, sliders, toggles, and dialog boxes. These prefabs were customized specifically to fit the maritime operational context of the AR System. For instance, the Hand Rays Toggle was developed using MRTK3's interactive toggle prefab, optimized for intuitive, gesture-based toggling.

Direct-hand interaction through finger taps was enabled via MRTK3's responsive button components, allowing users to activate menu items and other interactive elements through simple touch gestures. Additional buttons and sliders were also designed using MRTK3's responsive UI components, ensuring consistent behavior

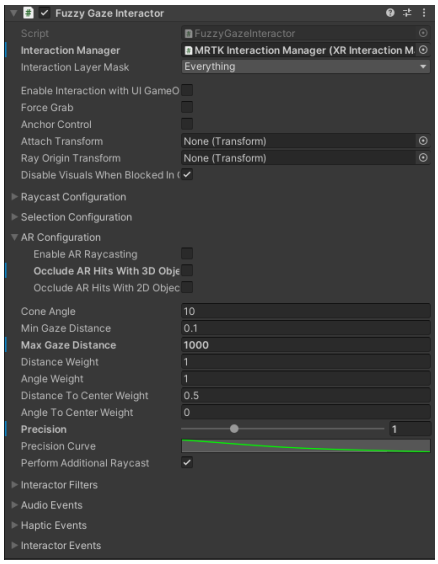


FIGURE 4.15: **Gaze Interactor Component.**

across the interface. These customizations contributed to a cohesive and seamless interaction experience (Fig. 4.16).

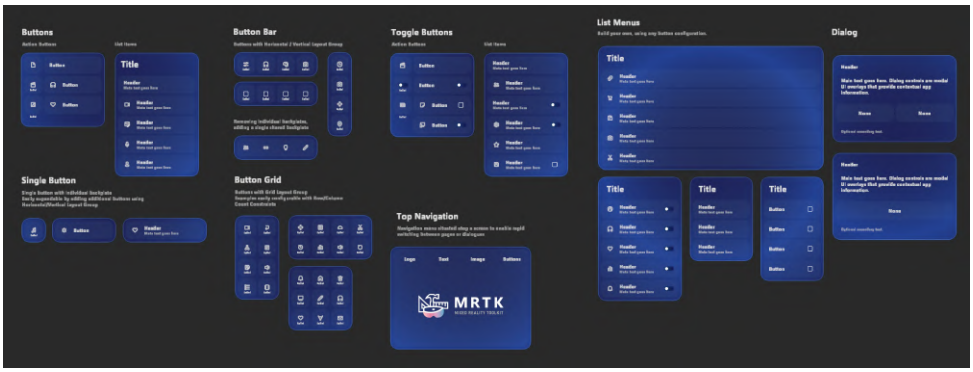


FIGURE 4.16: **Customized MRTK3 UI Components.**

4.6 Calibration Process

The calibration sequence is the indispensable first action when the AR application launches: it guarantees that every hologram conforms precisely to the physical scene that the user perceives. Its purpose is to register digital content—Points of Interest (POIs),Navigational map, AR-windows and route visualisations—so that each appears in the exact bearing, range, and elevation dictated by the user's true position and head orientation.

If this spatial handshake is skipped or performed incorrectly, Unity's virtual axes diverge from geographic reality. True-north indicators wander, POI labels drift off their targets, and route lines may float across bulkheads or disappear beneath the deck edge. Such discrepancies erode operator trust and, in time-critical scenarios, can mislead crew members during collision-avoidance. Accurate, repeatable calibration is therefore not a cosmetic refinement but a safety prerequisite, aligning the

synthetic layer with nautical instruments so that officers can act on the augmented cues with confidence.

4.6.1 Stage I: True-North Alignment

Context

A fundamental obstacle surfaced early in development: Unity's forward axis has no inherent relationship to geographic north because HoloLens 2 provides no absolute-heading sensor. When a new session starts, the device's visual-inertial SLAM establishes its own local reference frame, and Unity places the camera at (0,0,0) looking straight along the positive `Z` direction. The yaw of that frame is determined solely by whatever orientation the headset happened to hold while the tracking system locked on to its first set of environmental features. Consequently, there is an unknown—and typically different each launch—angular offset between "Unity-north" and true north on the chart.

Every bearing-driven overlay inherits that yaw error. A point of interest transmitted over AIS may be rendered several degrees to port of its actual bearing; a collision-risk cone intended to mark a vessel on the starboard bow could float into open water. In confined coastal waters or during time-critical manoeuvres, these misregistrations erode officer confidence and can prompt crews to discard the AR aid entirely. Resolving the unknown yaw offset—by explicitly aligning Unity's world frame to geographic north—therefore became the first, mandatory step in the overall calibration workflow.

Approach

To address this issue, a calibration routine was implemented to align the AR system's internal North with geographic true North. This procedure is initiated automatically when the application launches and can be re-run at any time via the Hand Menu if the spatial alignment appears incorrect. By performing this calibration, all augmented elements are guaranteed to be correctly positioned and oriented relative to the physical world.

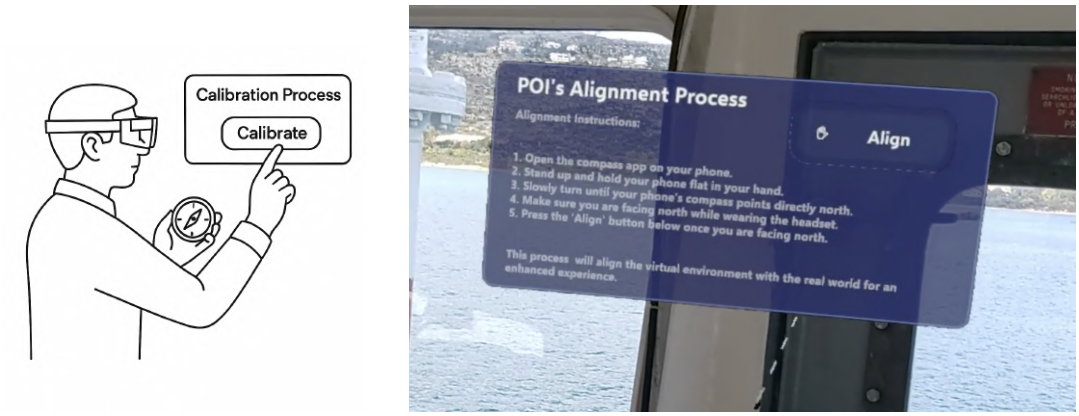
During calibration, the user follows guidance displayed in the Calibration Instructions UI, which directs them to orient themselves toward true North before tapping the `Calibrate` button. When the user confirms, the system executes the `Align()` routine to rotate the virtual scene so that its north axis corresponds with the user's actual heading.

Implementation Details

When prompted, the user opens a compass app on their smartphone, turns until it indicates true North, and then physically faces that direction while wearing the HoloLens 2. The headset displays on-screen prompts to assist with this orientation.

Once the user is correctly aligned, they select the `Calibrate` button, invoking the application's calibration routine. This action synchronizes the AR world's north reference with real-world North.

Internally, pressing `Calibrate` calls the `Align` function. This method computes the angular offset between the headset's current heading and true North, then applies



(A) (B)

FIGURE 4.17: **Calibration interface and user interaction. (A)** Simplified sketch of a HoloLens wearer consulting a compass and tapping the floating `Calibrate` control to begin true-North alignment. **(B)** On-device view showing the "POI Alignment Process" panel overlaid on the real environment, with stepwise instructions and the `Align` button.

the necessary rotation to the virtual environment so that all AR content is properly aligned.

```
0 references
public class AlignManager : MonoBehaviour
{
    2 references
    public GameObject alignObject; // The object whose Y-rotation we want to match to the camera
    1 reference
    public Camera mainCamera; // The camera we want to align with
    0 references
    public int degrees; // (Optional) degrees offset if you need to tweak alignment
    0 references
    public void Align()
    {
        // Compute yaw difference between target object and camera
        float angle = alignObject.transform.rotation.eulerAngles.y - mainCamera.transform.rotation.eulerAngles.y;
        Debug.Log("Angle: " + angle);

        // Set target's Y-rotation to that difference (zero out X and Z)
        alignObject.transform.rotation = Quaternion.Euler(0, angle, 0);

        // Turn off this manager once alignment is done
        gameObject.SetActive(false);
```

FIGURE 4.18: **Alignment Method**

4.6.2 Stage II: Alining Horizon

Context

In augmented reality maritime navigation, one of the most effective ways to improve depth perception is to ensure that the virtual route overlay visually extends all the way to the real horizon. This creates a convincing 3D illusion that the route is truly anchored in the world, helping users intuitively understand the scale and direction of their intended path.

However, achieving this effect is not as simple as drawing a line at a fixed height or distance. The position where the virtual route should "meet" the horizon depends entirely on how far away the last route point (the endpoint) is from the user. For

example, if the endpoint is only 100 meters away, the amount it must be elevated (shifted in Y) to reach the perceived horizon is quite different from a route that ends 1000 meters away. In practical terms, the greater the distance to the endpoint, the less vertical shift is needed for it to appear at the horizon in the user's view.

This happens because of basic perspective geometry: as objects get further from the user, their angular elevation relative to the observer's eyes decreases. To have a distant route point appear at the horizon line (as seen through the bridge window), the system must compute and apply just the right amount of vertical offset for that specific distance.

The horizon calibration process is therefore essential: it enables the system to "learn" the correct amount of Y-axis adjustment needed to make the route's final point merge with the real-world horizon from the user's current viewpoint. During calibration, the user is asked to align a movable virtual line—controlled hands-free by tilting their head—with the actual horizon as they see it. The system then records the exact Y-offset required for that particular route endpoint distance. This approach ensures that, no matter how long the planned route, the final waypoint will always be projected at the correct elevation to visually meet the perceived horizon, maximizing depth perception and spatial realism in the AR overlay.

Approach

To calibrate the route endpoint so that it meets the user's perceived horizon, the system implements an intuitive, hands-free workflow. When this stage of calibration begins, the AR application displays a clear, virtual horizon line across the user's view, accompanied by on-screen instructions (see Figure 4.19).

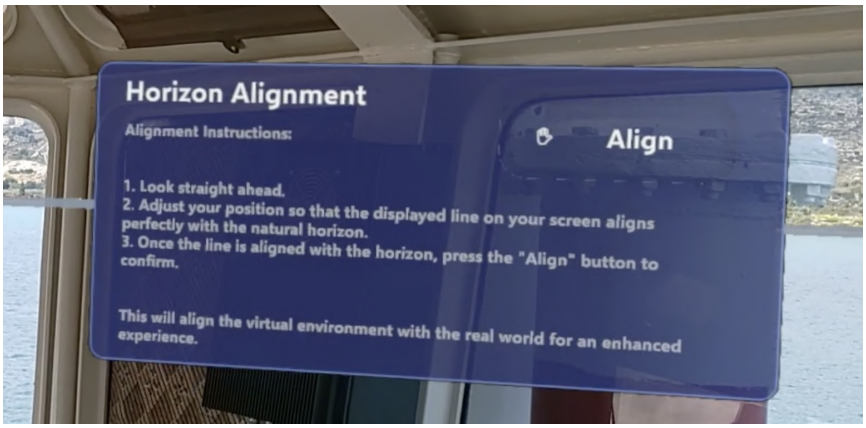


FIGURE 4.19: **Horizon alignment menu** with stepwise guidance

The unique aspect of this approach is that the horizon line is not fixed at a predetermined distance; instead, it is positioned at the same distance as the last route point. This ensures that the calibration directly corresponds to the actual route geometry and user perspective. The user does not manipulate the line with their hands—instead, they simply move their head vertically (looking up or down) to bring the virtual line into alignment with the real horizon as they perceive it through the bridge windows. This "gaze-aligned" interaction makes the process rapid, precise, and fully hands-free, minimizing cognitive load and physical distraction.

Once the user is satisfied that the virtual line coincides with the true horizon, they confirm the calibration with a simple gesture command. At this moment, the system records the vertical offset required to position the route endpoint at the perceived horizon for that specific route distance. This calibration value is then used to dynamically set the Y-position of the final waypoint during actual route rendering.

By linking the virtual route's endpoint directly to the user-defined horizon, this method ensures that route overlays maintain convincing perspective and spatial coherence, regardless of the route's length or the user's viewing position. As a result, the system delivers a much more legible, immersive, and operationally useful AR navigation experience on the bridge.

Implementation Details

The horizon alignment process leverages both real-time scene analysis and hands-free user input to ensure that the virtual route endpoint reaches the perceived horizon. The following Implementation Details summarize the core steps of this stage:

1. **Identify the Furthest Route Point.** At the beginning of the calibration, the system iterates through all available route points (POIs) and calculates their planar (XZ-plane) distance from the user. The route point that is furthest away is identified as the target for the horizon alignment, as shown in the code below (Figure 4.20).

```
public void StartHorizonCalibration()
{
    userPos = Camera.main.transform.position;
    userY = userPos.y;
    furthestDistance = -1f;
    furthestPOI = null;

    foreach (var poiList in spawnedObjects.Values)
    {
        foreach (var poi in poiList)
        {
            Vector2 userXZ = new Vector2(userPos.x, userPos.z);
            Vector2 poiXZ = new Vector2(poi.transform.position.x, poi.transform.position.z);
            float dist = Vector2.Distance(userXZ, poiXZ);
            if (dist > furthestDistance)
            {
                furthestDistance = dist;
                furthestPOI = poi;
            }
        }
    }

    if (furthestDistance < 0f || furthestPOI == null)
    {
        Debug.LogWarning("No POIs found to align.");
        return;
    }

    Vector3 forward = Camera.main.transform.forward;
    forward.y = 0; forward.Normalize();
    Vector3 targetPos = userPos + forward * furthestDistance;

    HorizonLine.transform.position = targetPos;
    HorizonLine.transform.rotation = Quaternion.LookRotation(-forward, Vector3.up);
    UpdateHorizonLineScale(HorizonLine.transform.position);
    Debug.Log("Horizon line moved for calibration.");
}
```

FIGURE 4.20: **Function for identifying the furthest POI and moving the horizon line.** The code finds the endpoint with the greatest distance and positions the virtual horizon accordingly.

```
public void UpdateHorizonLineScale(Vector3 horizonLinePosition)
{
    float distance = Vector3.Distance(Camera.main.transform.position, horizonLinePosition);

    // Use the proportional scaling with LerpUnclamped, just like with the POIs
    float t = (distance - userReferenceDistance) / (furthestReferenceDistance - userReferenceDistance);
    t = Mathf.Max(0, t);
    HorizonLine.transform.localScale = Vector3.LerpUnclamped(horizonLineScaleAtUser, horizonLineScaleAtFurthest, t);
}
```

FIGURE 4.21: **Script for scaling the virtual horizon line.** The line's size is adjusted based on its distance from the user.

2. **Move the Virtual Horizon Line.** Once the furthest POI is identified, the system projects a virtual line into the user's view at exactly the same distance as this endpoint, directly along the camera's forward vector. This guarantees that the calibration process is always tailored to the current navigational context, regardless of route length or orientation.
3. **Scale the Horizon Line for Perspective Consistency.** To preserve a realistic visual appearance across varying distances, the horizon line is dynamically scaled using proportional interpolation (Figure 4.21). This scaling ensures the line remains legible and consistent with other AR overlays, regardless of how far away the route endpoint is.
4. **Hands-Free Alignment.** The user aligns the horizon line with the real, visible horizon by simply moving their head up or down. As shown in the user interface (Figure 4.19), the line follows the user's head pitch, enabling rapid, precise adjustment without requiring any manual input.
5. **Capture Calibration Offset.** When the user confirms the alignment (e.g., by tapping the `Align` button), the system records the exact Y-offset required for the route endpoint to coincide with the perceived horizon at that specific distance. This value is then used in subsequent route rendering, guaranteeing that future overlays will consistently "reach" the horizon, regardless of the planned route's geometry or the user's position.

Through this mechanism, the AR navigation system delivers highly accurate, visually convincing route overlays that maximize depth perception and spatial realism for shipboard users.

4.6.3 Stage III: Aligning AR Windows

At the beginning of Stage III, the user is presented with an on-screen instruction panel explaining the four-corner placement workflow and available hand-gesture commands (Figure 4.22a).

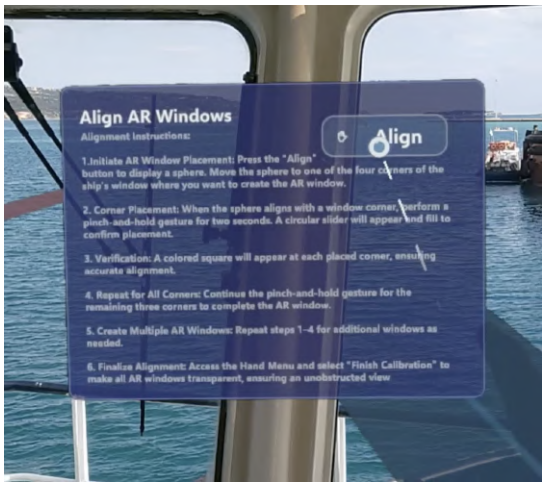
Context

To ensure virtual overlays never obscure critical bridge instruments and sight-lines, all virtual content is confined within configurable "AR windows" that map directly onto the ship's existing windows. This guarantees that no virtual element will occlude essential real-world controls or displays. Because bridge windows are often trapezoidal, raked, or otherwise non-rectangular, the placement workflow must support rapid, flexible definition of arbitrary quadrilateral regions without cumbersome manual resizing.

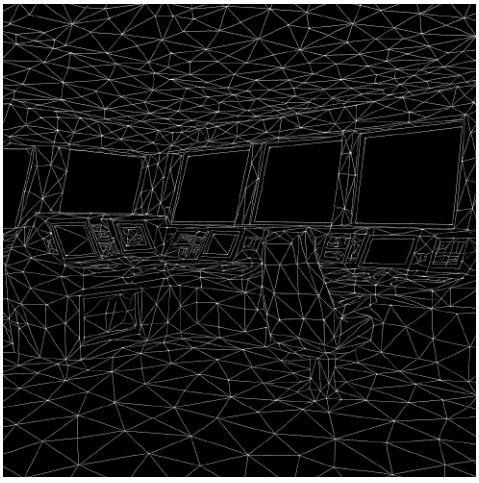
Approach

The system leverages the HoloLens 2's built-in spatial mapping to generate a transient mesh of the user's forward field of view(Figure 4.22b). A gaze-driven sphere then uses Unity's physics engine to perform collision detection against this mesh, snapping precisely to the nearest surface intersection—typically the corner of a window frame. Because only the small region directly in front of the user is mapped, there is no need to scan the entire bridge. As a result, mesh generation completes in under a second, keeping the calibration flow smooth and uninterrupted.

The ARMeshManager (from MRTK/AR Foundation) orchestrates mesh updates by interfacing with the XR MeshSubsystem. It filters out low-confidence triangles, dynamically adjusts mesh density within the gaze region, and discards irrelevant surfaces. By limiting mesh generation to the moment of corner placement, the ARMeshManager minimizes CPU and memory overhead while maintaining high-fidelity collision detection.



(A) **User guidance panel.** Instructions for four-corner placement and hand gestures.



(B) **Transient spatial mesh.** Real-time wireframe render.

FIGURE 4.22: **Stage III:** AR-window menu and Live scan.

Implementation Details

1. **Spatial Mapping & Collision.**
   - On pinch-and-hold start, the system invokes the HoloLens Spatial Mapping API to update a local mesh of the area in front of the user.
   - The gaze-controlled sphere (a Unity Rigidbody with a SphereCollider) uses OnCollisionEnter callbacks to detect contact with the mesh and highlight potential corner points.
2. **Corner Placement.** The user directs the sphere to a window corner, then performs a two-second pinch-and-hold. A circular fill UI around the sphere indicates progress (Figure 4.24b); upon completion, the corner's world position is locked.
3. **Visual Feedback.** As each corner is confirmed, a colored marker appears. Once all four corners are set, the newly formed AR window is shaded (e.g.,

```
Ray camRay = new Ray(Camera.main.transform.position, Camera.main.transform.forward);
int layerMask = ~(1 << LayerMask.NameToLayer("Ignore Raycast"));

if (Physics.Raycast(camRay, out RaycastHit hit, Mathf.Infinity, layerMask))
{
    currentHitPoint = hit.point;
}
```

FIGURE 4.23: **Sampling the spatial mesh.** A gaze-aligned ray cast retrieves **currentHitPoint**, anchoring the preview marker to the exact surface the officer is looking at.

```
if (isPinching)
{
    pinchTimer += Time.deltaTime;
    UpdateSlider(pinchTimer / pinchHoldTime);

    if (pinchTimer >= pinchHoldTime)
    {
        ConfirmCorner(currentHitPoint);
        CancelPinch();
    }
}
```

(A) **Pinch-hold logic.** The two-second dwell fills the slider and, once complete, calls **ConfirmCorner()**.

(B) **Real-world execution.** The officer holds a pinch over the window frame until the UI fills, locking the corner.

FIGURE 4.24: **Corner placement logic and execution. (A)** Pinch-hold loop that confirms the corner once the time threshold is reached. **(B)** User holding a pinch gesture to place a corner at the edge of the window frame.

semi-opaque blue) so the user can verify alignment (Figure 4.25a). A delete icon at the top of the window allows immediate removal if it's misplaced.

4. **Undo & Reset.** Via the hand-gesture menu, the user can "Undo Last Corner" or "Reset All Corners" at any time before finalizing the window(Figure 4.25a).

5. **Finalization & Re-Calibration.** After the user places all desired AR windows, their shading transitions to full transparency, revealing only the virtual content within. Any window can be re-calibrated later through the hand menu's "Re-align Window" command (Figure 4.25a) , which re-opens the corner-placement workflow for that window.

4.6.4 Stage IV: QR Code Assisted Calibration

Context

The HoloLens 2 relies on a Simultaneous Localization and Mapping (SLAM) engine to determine its position and orientation in space by continuously tracking visual and inertial features in the environment. Under normal conditions—walking around a room—this approach yields stable, world-locked content. However, on a ship's bridge the entire environment can shift: the vessel heels, pitches, or changes

(A) **In-situ feedback.** Two semi-transparent AR windows are visible while the contextual hand-menu offers **Set Windows**, **Undo Corner**, and **Cancel Current Panel**.

```
if (confirmedCornerMarkerPrefab != null)
{
    GameObject tempMarker = Instantiate(confirmedCornerMarkerPrefab, cornerPos, Quaternion.identity);
    tempMarker.name = $"TempCorner_{currentCornerIndex + 1}";
    temporaryCornerMarkers.Add(tempMarker);
}
```

(B) **Marker instantiation.** Each confirmed corner spawns a coloured marker via **Instantiate()**, then stores it in **temporaryCornerMarkers**.

FIGURE 4.25: **Visual feedback during AR window alignment. (A)** Hand menu with options to finalize, undo, or reset the current AR window. **(B)** Code snippet responsible for spawning corner markers used for visual reference and undo functionality.

course, and the headset's SLAM system cannot distinguish between these platform motions and user motion. As a result, virtual overlays such as AR-windows, navigational minimaps, and instrument annotations gradually "lag" behind their physical counterparts, accumulating several centimeters of drift over long transits or during abrupt maneuvers.

One might consider using the HoloLens's connection through a server to a mobile device GPS feed to correct drift. In theory, the mobile unit's GNSS fixes could be streamed to the headset and used to re-anchor the virtual content. In practice, however, consumer-grade GPS suffers from metre-level accuracy and update rates on the order of 1 Hz—orders of magnitude too coarse and too slow to maintain the sub-decimetre precision and real-time responsiveness required for a seamless AR experience on a moving vessel.

To overcome these limitations, a network of fixed, ship-mounted visual landmarks in the form of Version-3 QR codes is introduced, each mounted at a precisely surveyed pose on the bridge coaming or bulkhead. The HoloLens's high-resolution camera and onboard computer-vision pipeline detect and decode these codes at up to 30 fps and estimate their 6 DoF pose in under 100 ms at typical operating distances (0.5–2 m). This near-instantaneous re-anchoring procedure is unaffected by multipath or signal blockage, and—when combined with SLAM's fine-grained relative tracking—provides both smooth continuous motion and periodic absolute resets that keep all virtual content registered to the true ship frame with sub-decimetre stability.

By periodically detecting these QR fiducials and creating corresponding AR anchors at their known installation poses, the system can re-align the SLAM-derived world frame to the actual bridge geometry—arresting drift and ensuring that AR-windows, instrument overlays, and minimaps remain precisely locked to their real-world counterparts throughout extended voyages.

Approach

A distributed network of Version-3 QR fiducials (40 mm modules) is installed at precisely surveyed locations on the bridge coaming and bulkheads. Each code encodes a unique ID and its installation pose in ship-local coordinates. At runtime, the calibration routine proceeds as follows:

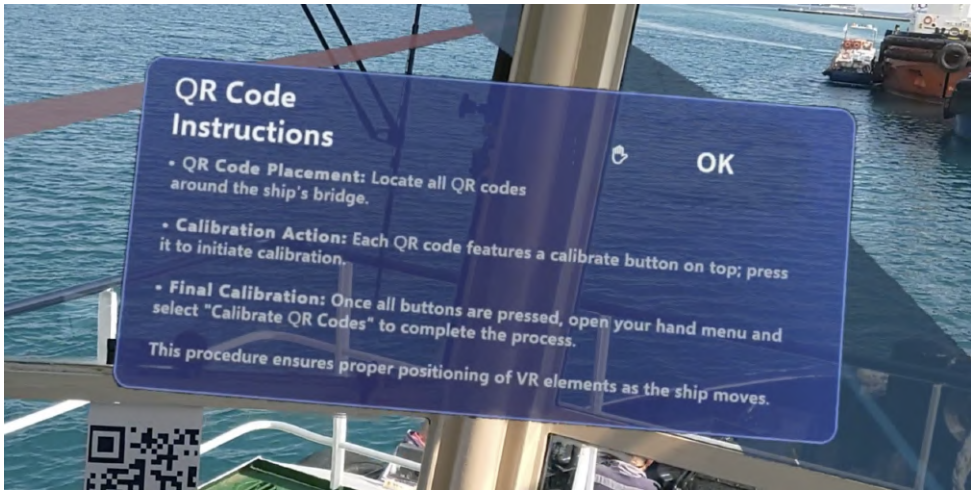


FIGURE 4.26: **QR-code calibration instruction panel.** Step-by-step guidance for locating ship-mounted fiducials, using the floating "Calibrate" buttons and finalizing the calibration.

1. **Marker Detection.** The HoloLens camera continuously scans its viewport for QR patterns using the OpenXR computer-vision pipeline. Detected codes are decoded in real time (up to 30 fps) and their 6 DoF pose estimated relative to the headset.
2. **Anchor Creation.** For each newly observed marker, ARAnchorManager.AddAnchor(pose) is invoked to instantiate a persistent ARAnchor at the marker's world position. Subsequent detections of the same ID update the anchor's transform via a running average filter to smooth jitter and compensate for detection noise.
3. **Global Fusion.** Once all fiducials have been anchored at least once, the hand-menu command "Calibrate QR Codes" triggers FinishCalibration(). This routine collects the full set of anchors and hands them to the World-Locking Tools sliding-window optimiser, which computes a corrective transform for the headset's world frame that best aligns the SLAM-derived map to the known marker poses.
4. **On-Demand Re-Anchoring.** At any time—after extended standby or abrupt vessel maneuvers—the user may invoke "Calibrate QR Codes" again. The same sequence of detection, anchoring, and fusion runs, instantly snapping

all AR content (windows, minimaps, overlays) back to its correct ship-fixed positions.

This QR-assisted calibration blends high-rate SLAM tracking with periodic absolute resets, combining smooth continuous motion with decimetre accuracy—even under extended transits.

Implementation Details

1. **Calibration-Offset Storage**
   - A dictionary is declared at the class level to hold each code's offset from the scene root (Figure 4.27).
   - Entries are added or updated at runtime by the `CalibrateMarker(ARMarker)` method.(Figure 4.28b).

```
// A dictionary storing each marker's offset from environmentRoot. Key = TrackableId
private Dictionary<TrackableId, CalibrationOffset> calibrationOffsets
    = new Dictionary<TrackableId, CalibrationOffset>();

// Represents the environmentRoot's offset (position + rotation) relative to a particular code.
private struct CalibrationOffset
{
    public Vector3 positionOffset;
    public Quaternion rotationOffset;
}
```

FIGURE 4.27: Declaration of the **calibrationOffsets** dictionary and **CalibrationOffset** struct, which store each QR marker's positional and rotational offset from **environmentRoot**.

2. **Anchor Lifecycle**
   - In calibration mode, `Update()` polls the `ARMarkerManager.trackables` collection each frame. For any newly detected marker with no child button, `SpawnCalibrationButton(marker)` instantiates a floating prefab directly in front of that code (Figure 4.28a).
   - Pressing the spawned button invokes `CalibrateMarker(marker)` (Figure 4.28b), which:
     (a) Reads the marker's world-space transform from `marker.transform`.
     (b) Computes and stores the scene-root offset in the marker's local frame
     (c) Saves these values in `calibrationOffsets[marker.trackableId]`.
3. **Mode Switch**
   - Invoking `FinishCalibration()` via a hand-menu
     – Destroys all calibration-button children of each marker.
     – Sets `isCalibrationMode = false`, entering operation mode.
     – Resets `currentUsedCode` to `TrackableId.invalidId`.
4. **Drift Detection & Alignment**
   - In operation mode, `Update()` iterates over all keys in `calibrationOffsets`. For each calibrated code:

```
// For each tracked marker, if it has no child button, spawn a calibration button
foreach (var marker in markerManager.trackables)
{
    if (marker == null) { continue; }

    // Only spawn a button if none exists yet
    if (marker.transform.childCount == 0 && calibrationButtonPrefab != null)
    {
        SpawnCalibrationButton(marker);
    }
}
```

(A) Loop in **Update()** that iterates over **markerManager.trackables** and spawns a floating calibration button for any newly detected QR marker.

```
public void CalibrateMarker(ARMarker marker)
{
    if (!isCalibrationMode)
    {
        Debug.LogWarning("[CalibOp] CalibrateMarker called but we're not in calibration mode!");
        return;
    }
    if (marker == null || environmentRoot == null)
    {
        Debug.LogError("[CalibOp] CalibrateMarker: marker or environmentRoot is null!");
        return;
    }

    TrackableId id = marker.trackableId;
    Debug.Log($"[CalibOp] CalibrateMarker called for code {id}.");

    // Compute offset: environmentRoot's position/rotation in marker's local coordinate space
    Vector3 posOffset = marker.transform.InverseTransformPoint(environmentRoot.position);
    Quaternion rotOffset = Quaternion.Inverse(marker.transform.rotation) * environmentRoot.rotation;

    CalibrationOffset offset = new CalibrationOffset
    {
        positionOffset = posOffset,
        rotationOffset = rotOffset
    };

    calibrationOffsets[id] = offset;
    Debug.Log($"[CalibOp] --> Code {id} CALIBRATED. positionOffset={posOffset}, rotationOffset={rotOffset}");
```

(B) The **CalibrateMarker(ARMarker)** method: computes the inverse transform of **environmentRoot** into the marker's local frame and stores the resulting **positionOffset** and **rotationOffset**.

FIGURE 4.28: **(A)** Spawning calibration buttons for new QR markers. **(B)** Computing and storing the scene-root offset in the marker's local frame via `CalibrateMarker()`.

(a) Finds the corresponding `ARMarker` via `FindMarkerById(codeId)`.

(b) Compares its current `marker.transform.position` to the last recorded position; if it moves beyond `moveThreshold`, calls `AlignWithCode(codeId)`.

5. **Re-Alignment**

- `AlignWithCode(codeId)` looks up the stored `CalibrationOffset` for that ID, locates the live `ARMarker`, then computes new possition and new rotation(Figure 4.29).
- Applies these to `environmentRoot.SetPositionAndRotation(newPosition, newRotation);`
- All child holograms—AR-windows, minimaps, panels—are parented under `environmentRoot` and thus instantly snap back into alignment.

```
private void AlignWithCode(TrackableId codeId)
{
    if (!calibrationOffsets.TryGetValue(codeId, out CalibrationOffset offset))
    {
        Debug.LogWarning($"[CalibOp] AlignWithCode({codeId}) => code not in dictionary!");
        return;
    }

    // Find the actual ARMarker in the trackables
    ARMarker foundMarker = null;
    foreach (var marker in markerManager.trackables)
    {
        if (marker != null && marker.trackableId == codeId)
        {
            foundMarker = marker;
            break;
        }
    }
    if (foundMarker == null)
    {
        Debug.LogWarning($"[CalibOp] AlignWithCode({codeId}) => marker not found in trackables!");
        return;
    }

    // Now compute the environmentRoot's new position/rotation
    Vector3 newPos = foundMarker.transform.TransformPoint(offset.positionOffset);
    Quaternion newRot = foundMarker.transform.rotation * offset.rotationOffset;

    environmentRoot.SetPositionAndRotation(newPos, newRot);
```

FIGURE 4.29: The **AlignWithCode(TrackableId)** method: retrieves the stored offset for a given marker, then repositions and reorients **environmentRoot** to snap all child holograms back into alignment.

4.6.5 Stage V: Map and Assistant Panel Placement

Context

With AR-windows aligned and world-drift arrested via QR-code calibration, the final step is to position the navigational map and auxiliary panels in ergonomic, non-obstructive locations on the bridge. These panels provide route selection toggles, ship status, and POI controls without blocking critical sight-lines through the forward windows.

Approach

When the operator presses the "OK" button on the Map and Panel Instructions panel (see Figure 4.30), the system opens all five UI elements in a recommended layout:

- A world-locked navigational map is placed centrally above the main forward window for maximum visibility.
- Two route-toggle panels appear just below the map, one on the lower-left and one on the lower-right, so that switching between planned and active routes is always within reach.
- Two ShipInfo panels spawn beneath each toggle panel, anchored to the left and right window pillars to display speed, heading, and wind data without obscuring the view ahead.

These default positions serve as a starting arrangement only. The user then grabs, pinches, moves, and even scales each panel to fine-tune its placement and size. Once satisfied, invoking the "Finalize Panel Placement" command from the hand menu

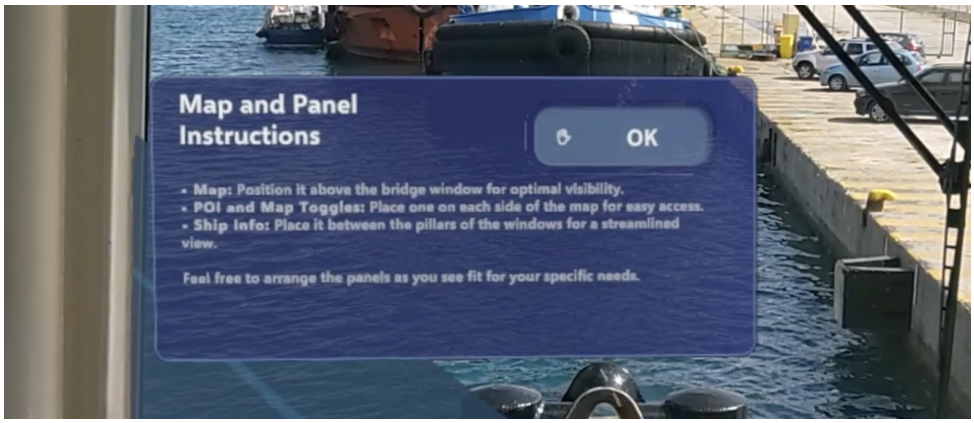


FIGURE 4.30: **QR-code calibration instruction panel.** step-by-step guidance for locating ship-mounted fiducials, using the floating "Calibrate" buttons and finalizing the calibration.

locks all five panels in place. These suggested positions are chosen according to maritime human-factors guidelines: the map sits in the operator's primary line-of-sight, toggles are within comfortable arm's-reach, and status information is peripheral but easily glanced at.

Implementation Details

- **Inspector-Driven Toggles and Clicks.**
  - The route lines (planned and uncertain) are enabled/disabled via the "On Toggled (Single)" and "On Untoggled (Single)" UnityEvents in the Inspector.
  - Each panel prefab (Map, POI toggles, Route toggles, ShipInfo left/right) is opened or closed by wiring its GameObject.SetActive calls into the "On Clicked ()" event of the OK button (Figure 4.31).

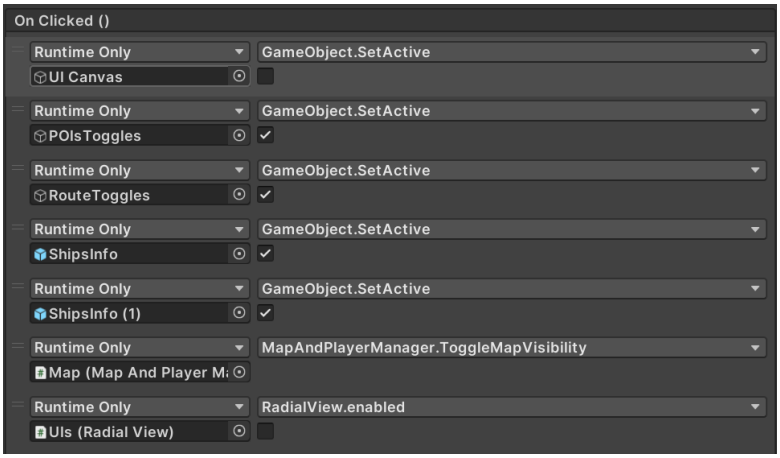


FIGURE 4.31: OK button's **"On Clicked ()"** event bindings for toggling map and panel GameObjects.

- **Direct Manipulation.** After instantiation, each panel is user-draggable and resizable via standard HoloLens grab-and-pinch gestures.
- **Finalization.** The hand-menu action `FinalizePanelPlacement()` iterates over the five panels, reads each panel's world-space `Transform`, computes its offset relative to the nearest QR code anchor (using the same offset-storage pattern from Stage IV), and then disables placement mode so the panels remain fixed.
- **Persistence.** On subsequent sessions or re-anchoring events, the stored offsets are reapplied so that each panel reappears exactly where the user left it, maintaining ergonomics and avoiding obstruction.

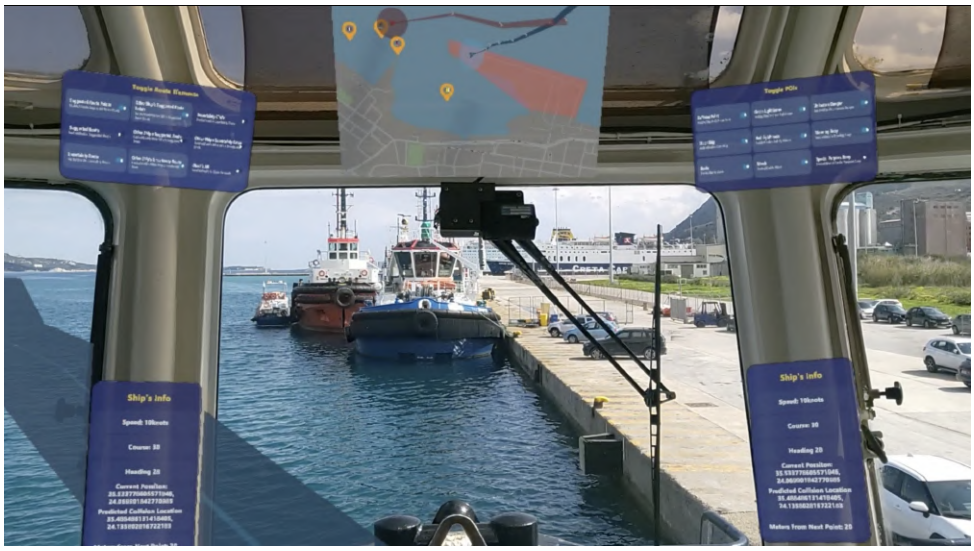


FIGURE 4.32: **Map, toggle, and info panels** in their suggested positions.

This completes the calibration process. One outcome of this process is the definition of AR display windows, a concept which the next section explores in detail.

4.7 AR Windows: Concept and Implementation

With the AR Window framing established during calibration, the AR Window is now examined in depth, including its implementation and its central role in AR visualization.

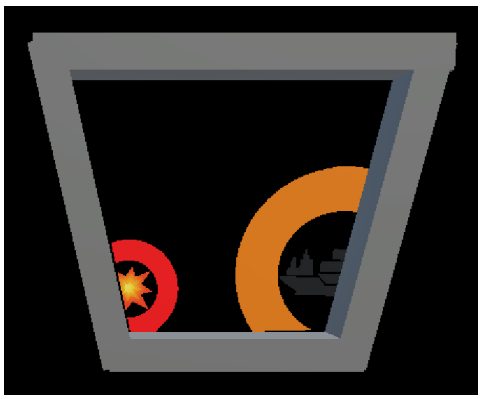
4.7.1 Definition and Rationale

In the context of head-worn augmented reality (AR) for maritime navigation, the term *AR Window* refers to a user-defined, spatially anchored portal through which digital content becomes visible only when viewed from specific angles or positions. Rather than displaying all AR elements globally across the user's field of view, the AR Window acts as a selective filter, ensuring that essential navigation aids, hazard visualizations, and points of interest (POIs) are visible only through intentional alignment and perspective.

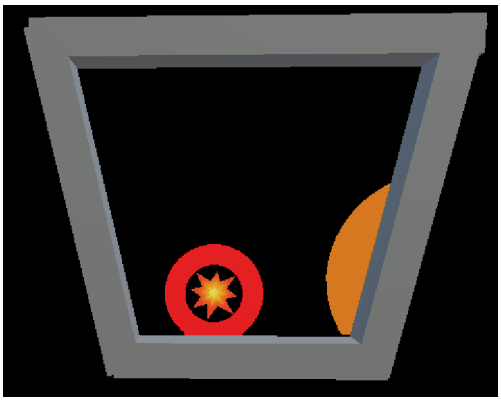
A fundamental motivation for this approach is that many virtual elements are only relevant when the captain is looking outside through the ship's actual windows.

For example, the predicted navigational route is generated starting from the ship's bow and follows a path ahead of the vessel; without AR Windows, this route could appear unrealistically on the bridge's interior walls or consoles when the bow is not visible, resulting in confusion and unnecessary visual clutter. By anchoring the route to display only within AR Windows, the system ensures the visualization remains contextually appropriate and always aligned with the real external scene.

This approach directly addresses longstanding challenges in maritime AR systems, which often suffer from information overload or distracting overlays in inappropriate contexts—especially in complex and safety-critical environments like ship bridges.



(A) From this angle, specific virtual items are visible within the portal's frame.



(B) Adjusting the perspective exposes different parts of the anchored content.

FIGURE 4.33: **Demonstration of AR Window functionality.**

4.7.2 Functional Overview

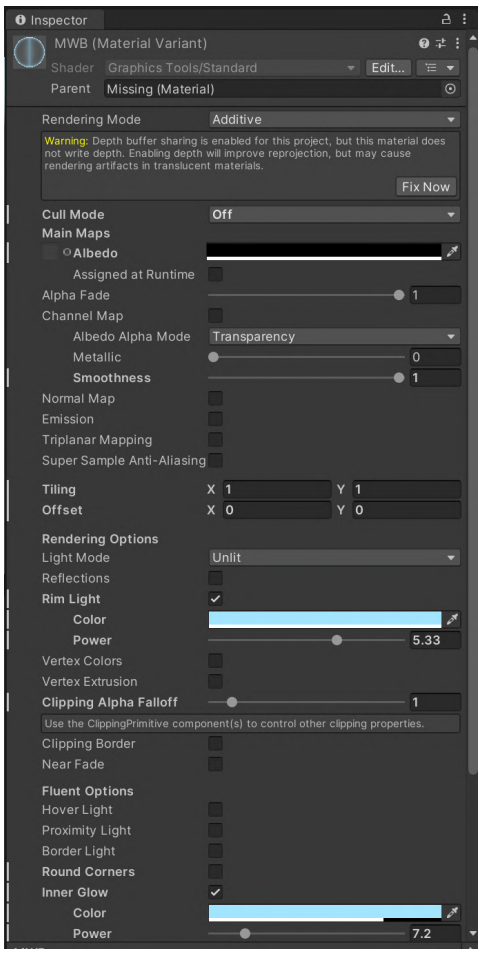
As described in Subsection 4.6.3, AR Windows are defined by the user marking four corners through precise gaze targeting and pinch gestures, creating a virtual rectangular panel matching the ship's real windows or bulkheads. Once placed, only digital elements located within the portal's defined spatial boundaries are rendered visible (Figure 4.33). This selective rendering dynamically adapts as the user changes position and viewing angle, closely mimicking the behavior of physical windows.

4.7.3 Technical Implementation

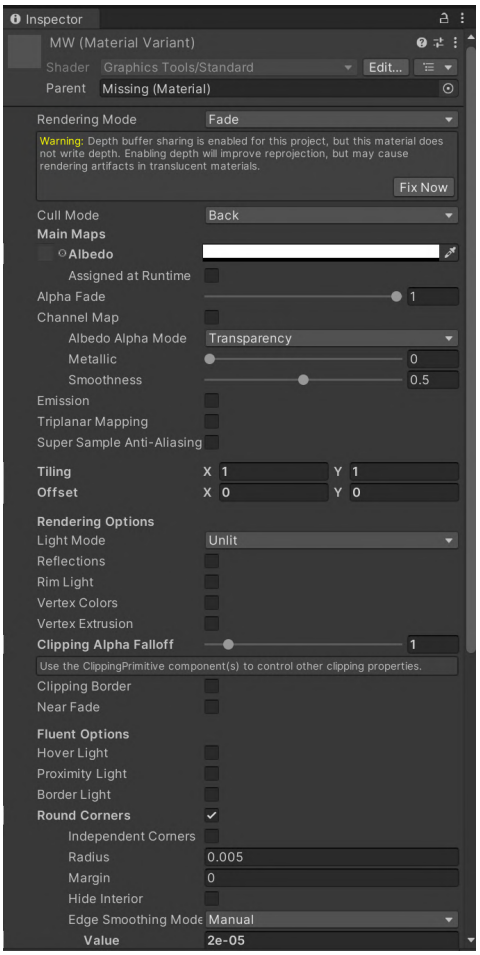
AR Window Panel Construction and Materials Upon corner placement, the AR Window panel is instantiated with two distinct visual states:

- **Placement State (semi-transparent blue):** This material provides rim lighting, inner glow, and gradual transparency, offering clear visual feedback during initial alignment and calibration.
- **Operational State (fully transparent):** Activated after final calibration, this material removes all visual distractions, providing a clear, unobstructed view for regular operations.

ARWindowClipped Material Properties Objects intended to appear only within AR Windows use a specialized ARWindowClipped material, characterized by:



(A) Semi-transparent placement state material.



(B) Fully transparent operational state material.

FIGURE 4.34: **Unity Inspector configurations for AR Window materials.**

- **Shader Type:** Custom clipping shader compatible with MRTK's ClippingPrimitive or custom implementations.
- **Rendering Mode:** Transparent rendering to integrate seamlessly with real-world backgrounds.
- **Alpha Clipping:** Pixels outside the AR Window's clipping volume are fully transparent.
- **Depth Handling:** Configured for correct spatial occlusion and depth consistency.

Clipping Shader and Rendering Logic The AR Window selective visibility mechanism relies on a clipping shader, implemented via MRTK's `ClippingPrimitive` or custom ClippingBox scripts. For each rendered pixel, the shader checks spatial boundaries; pixels outside the AR Window mask are rendered transparent. For example, the navigational route appears exclusively when aligned with the AR Window, avoiding unrealistic interior rendering.

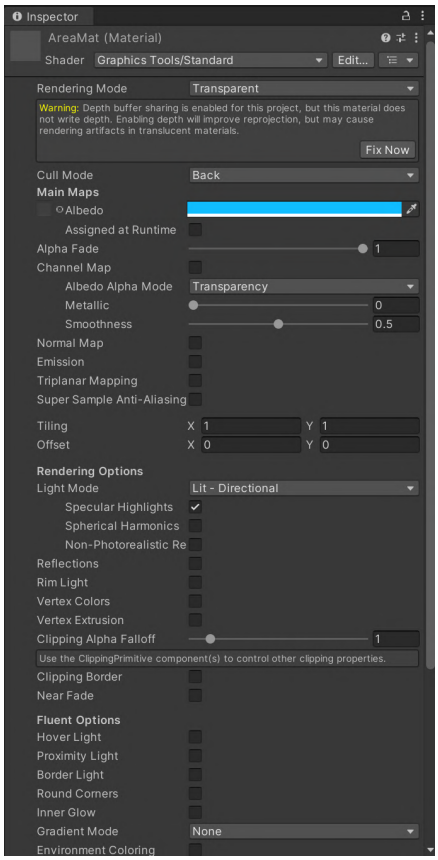


FIGURE 4.35: **ARWindowClipped material settings.** Ensures visibility strictly through AR Windows.

4.7.4 Justification for Use in Maritime Navigation

There are several reasons why the AR Window paradigm was chosen for this maritime AR system:

- **Context-Aware Visualization:** By displaying information only when the user is looking through the ship's actual windows, the system prevents the presentation of virtual content (e.g., POIs, hazard zones, routes) when it is irrelevant or distracting—such as when the captain is facing a wall, equipment cabinet, or interacting with interior elements.
- **Route Realism and Relevance:** The predicted route line begins at the bow and should logically continue into the real-world sea beyond the bridge. AR Windows ensure that this route never appears in interior spaces, thus preserving the realism and utility of the visualization.
- **Reduction of Visual Clutter:** Maritime environments are highly dynamic and visually complex. Global overlays can easily obscure important visual cues, such as other vessels, buoys, or weather phenomena. The AR Window confines digital content to intentional portals, minimizing distraction and reducing the risk of missing critical real-world information.
- **Safety and Regulatory Compliance:** International maritime regulations and bridge design standards often require clear, unobstructed views for navigation and collision avoidance. The AR Window ensures compliance by guaranteeing

that AR content does not intrude upon the user's peripheral vision or block essential sightlines.

- **User Flexibility and Ergonomics:** By enabling custom placement and resizing of AR Windows, users can tailor the interface to their specific bridge layout and personal preferences. This adaptability is essential for practical deployment across vessels of varying design and crew workflow.
- **Enhanced Depth Perception and Alignment:** Anchoring AR visualizations to real-world windows enhances depth cues and spatial awareness. Predicted routes, hazard zones, and POIs projected through AR Windows are intuitively interpreted in the context of the external scene, supporting rapid and accurate decision-making.

4.7.5 Comparison to Alternative Approaches

Alternative AR display strategies, such as global overlays, floating HUDs, or context-free holograms, were considered but found lacking in maritime applications due to their propensity for visual occlusion, loss of depth reference, and user distraction. The AR Window paradigm uniquely addresses these limitations by:

- Limiting the spatial extent of digital content,
- Enabling context-aware alignment with ship infrastructure,
- Preserving the integrity of the user's external view at all times.

In summary, the AR Window is a core innovation in the presented AR system, enabling safe, effective, and user-centric visualization of complex maritime information.

With the AR Window mechanism in place to control where graphics appear, the chapter next addresses what information to display. The following sections introduce the core AR content elements, starting with Points of Interest.

4.8 Points of Interest (POIs) in the AR Environment

Points of Interest (POIs) are a central component of the head-worn AR navigation system, designed to enhance operational decision-making for ship captains by marking and contextualizing critical locations and hazards within the maritime domain. POIs provide immediate visual cues about navigational risks, other vessels, obstacles, and maritime landmarks, seamlessly integrated into the user's field of view via stereoscopic overlays.

This section details the technical realisation and design rationale for POIs in the developed system, covering:

- The design and categorization of maritime POIs, including the adoption of standardized nautical symbols.
- The workflow for dynamically placing POIs within the AR environment based on real-time GPS and AIS data.
- Interactive and ergonomic features that optimize POI usability, including gaze-based interaction and information panels.

4.8.1 POI Design and Structure

The design of Points of Interest (POIs) is crucial in enhancing situational awareness for professional mariners. Each POI is represented by a unique 3D model, crafted in Blender, an open-source 3D modeling software that enables intricate designs and textures. o maximize usability and minimize cognitive load, the POI icons were designed to resemble familiar symbols from official nautical charts (e.g., Navionics). This familiarity makes each POI immediately recognizable, requiring minimal training for effective use. This approach allows critical navigational information to be conveyed at a glance, supporting intuitive and efficient decision-making on the bridge.

A summary of the designed POI icons, each corresponding to a key category within the maritime context, is presented in Figure 4.36 below. The main types and their specific roles are described in the following paragraphs.

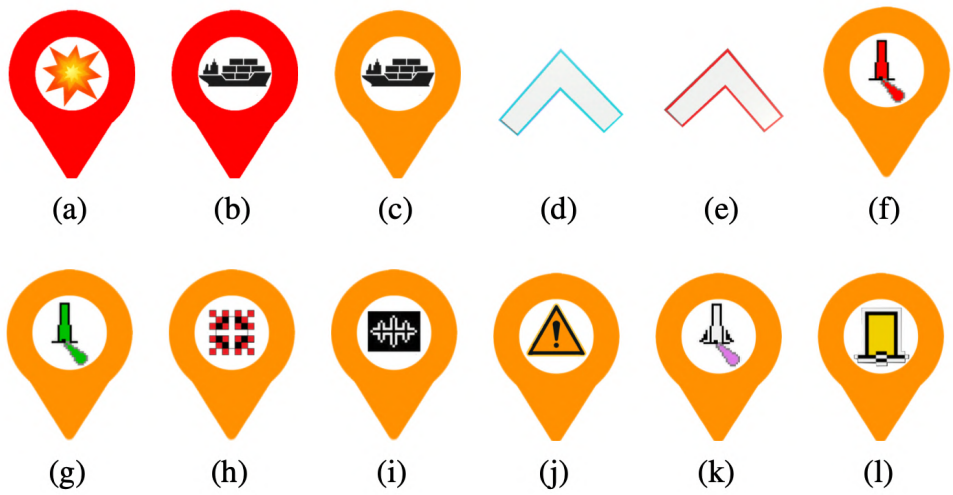


FIGURE 4.36: **POI Designs**

- **Collision Point:** Marks the predicted location where a potential ship collision could occur, based on real-time trajectory analysis.
- **Conflicting Vessel:** Highlights the vessel identified as being on a collision course with the user's ship.
- **Ship Passing By:** Indicates a nearby vessel passing within close proximity but not on a collision path.
- **Arrow at Ship's Route Point:** Displays an arrow marking a waypoint or direction change along the user's planned route.
- **Arrow at Conflicting Vessel's Route Point:** Shows the projected route or waypoints for the conflicting vessel, aiding in traffic assessment and collision avoidance.
- **Red Lighthouse:** Represents a port-side (left) navigational beacon, used to mark the edge of safe water.

- **Green Lighthouse:** Represents a starboard-side (right) navigational beacon, used to mark the edge of safe water.
- **Reef:** Identifies the position of a reef or shallow area that poses a hazard to navigation.
- **Wreck:** Marks the location of a sunken vessel, warning of underwater obstruction.
- **Unknown Danger:** Used for reported but unclassified hazards or obstacles requiring caution.
- **Mooring Buoy:** Indicates a designated point for mooring or anchoring a vessel safely.
- **Special Purpose or General Buoy:** Marks a buoy used for special navigational purposes (e.g., restricted zones, data collection) or general warnings not covered by other types.

These visual models not only facilitate intuitive recognition but also underpin the logic for managing and displaying POIs throughout the application. The available POI categories are standardized in the `POIType` enumeration (Figure 4.37), which ensures consistent handling and display of all POI types throughout the application.

```
public enum POIType
{
    CollisionPoint, Ship,
    OtherShip, MyRoutePoint,
    OtherRoutePoint, General,
    ShipPassingBy, RedLighthouse,
    GreenLighthouse, Reef,
    UnknownDanger, MooringBuoy,
    SpecialPurposeBuoy, Wreck
}
```

FIGURE 4.37: **Definition of the `POIType` enumeration in Unity.** This enumeration lists all available categories for Points of Interest (POIs) supported by the AR navigation system. Each value corresponds to a unique POI class, ensuring consistent type handling throughout the application.

These icon models not only act as visual cues but also categorize navigational information, making the AR display more organized and user-friendly.

4.8.2 POI Spawning

This subsection examines the implementation of POI spawning within the AR application, covering both static Points of Interest—preloaded and initialized at startup—and dynamic POIs, which are generated in real time based on data received from the server. A discussion is presented on the methods and workflows employed to accurately place and manage each type of POI within the AR environment, ensuring seamless integration and accurate placement.

Receiving POI Data from the Server

The server operates as a Kafka consumer, continuously monitoring a designated Kafka topic for new messages. Upon receiving a relevant message, it forwards the content to the HoloLens application via an established WebSocket connection. Each message is formatted as a JSON object containing detailed POI information.

Upon receipt, the AR application parses the JSON payload to extract the POI type, latitude, and longitude, constructing a corresponding `GPSCoordinate` instance. This class standardizes the handling of geographic data within Unity by storing the coordinate string and enumerated POI type, while also providing properties for direct access to the parsed latitude and longitude values. This ensures type safety and simplifies downstream spatial computations.

Figure 4.38 presents the implementation of the `GPSCoordinate` class in Unity, which serves as the foundational data structure for all Points of Interest. This design enables seamless conversion from server-sent JSON data to interactive, spatialized AR content.

```
public class GPSCoordinate
{
    [Tooltip("Input format: 'Latitude,Longitude' e.g., '34.0522,-118.2437'")]
    public string coordinates;
    public POIType POIType;

    public double Latitude
    {
        get
        {
            return double.Parse(coordinates.Split(',')[0].Trim());
        }
    }

    public double Longitude
    {
        get
        {
            return double.Parse(coordinates.Split(',')[1].Trim());
        }
    }
}
```

FIGURE 4.38: **Implementation of the `GPSCoordinate` class in Unity.** The class standardizes the parsing and storage of POI information, facilitating accurate placement and efficient management of both static and dynamic POIs in the AR environment.

GPSCoordinate Class Fields The `GPSCoordinate` class encapsulates all essential information required to represent a Point of Interest within the Unity environment. Its fields are designed for both flexibility and type safety:

- `coordinates` **(string):** Stores the latitude and longitude in a comma-separated string format (e.g., "34.0522,-118.2437"). This design simplifies storage and initial parsing from JSON messages.
- `POIType` **(enum):** An enumerated value specifying the category of the POI, as defined by the `POIType` enumeration (see Figure 4.37). This field enables type-safe handling and ensures each POI is rendered with the appropriate icon and behavior.

- `Latitude` **(double, property):** Extracts and returns the latitude component from the `coordinates` string, parsed as a double for spatial calculations.
- `Longitude` **(double, property):** Extracts and returns the longitude component from the `coordinates` string, also parsed as a double.

This structure allows the system to efficiently convert both static and server-sent POI data into spatially accurate objects within the AR environment, facilitating real-time navigation and interaction.

The JSON parsing logic responsible for this transformation is illustrated in Figure 4.39. Here, the application reads the POI type, latitude, and longitude fields from the incoming message and populates the relevant properties of the `GPSCoordinate` class.

```
JObject payload = (JObject)jsonObject["payload"];
if (payload == null)
{
    Debug.LogError("Spawn POI message has no payload");
    return;
}

double? collisionLon = payload["collision_lon"]?.Value<double>();
double? collisionLat = payload["collision_lat"]?.Value<double>();
if (collisionLon != null && collisionLat != null)
{
    GPSCoordinate coordinates = new GPSCoordinate
    {

        coordinates = collisionLat + "," + collisionLon,
        POIType = POIType.CollisionPoint
    };
```

FIGURE 4.39: **Parsing incoming POI JSON data and converting it into a `GPSCoordinate` object.** This process ensures seamless integration of server-sent dynamic POIs into the Unity-based AR environment.

Before a POI can be instantiated in the AR scene, its real-world geographic coordinates must be accurately mapped to Unity's local coordinate system.

Latitude and Longitude to Unity Coordinates

Accurate placement of Points of Interest within the AR environment requires converting real-world GPS coordinates (latitude and longitude) into Unity's local coordinate system. This is achieved through a conversion algorithm that computes spatial offsets from a defined reference point—typically the user's current location.

The conversion process operates as follows:

- **Latitude Offset Calculation:** Computes the difference in latitude between the POI and the reference point, multiplying this value by an approximate metric conversion factor (111,000 meters per degree of latitude).
- **Longitude Offset Calculation:** Computes the longitude difference, adjusts for the Earth's curvature by multiplying with the cosine of the latitude, and then applies the same conversion factor to obtain the offset in meters.
- **Vector3 Position Construction:** Combines the latitude and longitude offsets to produce a new `Vector3` position, which places the POI relative to the user in Unity's 3D scene.

```
Vector3 CalculateObjectLocalPosition(double latitude, double longitude)
{
    double latOffset = (latitude - referenceLatitude) * 111000.0;  // meters per latitude degree
    double lonOffset = (longitude - referenceLongitude) * (111000.0 * Mathf.Cos((float)(referenceLatitude * Mathf.PI / 180.0)));
    return new Vector3((float)lonOffset, 0, (float)latOffset);
}
```

FIGURE 4.40: **Conversion of GPS coordinates to Unity's local coordinate system.** The function calculates relative spatial offsets to ensure correct placement of POIs within the AR scene.

The `CalculateObjectLocalPosition` function (Figure 4.40) implements this approach, determining each POI's position by referencing the user's current GPS location. While this method provides a practical and efficient solution for typical maritime navigation scenarios, it does introduce minor inaccuracies due to the Earth's curvature. The standard approximation of 111,000 meters per degree of latitude is generally sufficient for short-range spatial placement; however, for greater precision over larger distances, future iterations of the system may adopt more advanced projections such as the Equirectangular or Mercator projection.

Spawn POI in the Environment

The instantiation of Points of Interest within the AR environment is managed by a dedicated workflow that takes incoming POI data, parses it, and renders the corresponding POI at the correct real-world position. This process is primarily orchestrated by the `SpawnPOIAtLocation` function (Figure 4.41), which accepts a fully constructed `GPSCoordinate` object as input.

When a spawning message is received from the server, the system first calls the `POIJsonParsing()` method (see Figure 4.39) to extract and construct the `GPSCoordinate` instance for the specified POI. This ensures that the type and coordinates are correctly formatted and ready for the instantiation process.

The `SpawnPOIAtLocation` method performs several key steps to accurately place and manage POIs in the Unity-based AR environment:

- **Coordinate Conversion:** Converts the GPS coordinates (latitude and longitude) into Unity's local coordinate system using the `CalculateObjectLocalPosition` function (Figure 4.40).
- **Prefab Instantiation:** Selects and instantiates the correct POI prefab from a mapping dictionary based on the POI type, placing it at the computed local position.
- **Scene Hierarchy Organization:** Assigns the instantiated POI GameObject to a parent object (named "POIs") for streamlined scene management and group operations.
- **Attribute initialisation:** Configures the `POIInteraction` component with specific attributes such as name, description, and risk level, enabling interactive features within the AR interface (see Figure **??**).
- **Dictionary Registration:** Stores the newly spawned POI in the `spawnedObjects` dictionary, grouped by type, to enable efficient filtering, enabling/disabling, and management according to user preferences or application state.

```
public GameObject SpawnObjectAtLocation(GPSCoordinate coord)
{
    Vector3 position = CalculateObjectLocalPosition(coord.Latitude, coord.Longitude);

    // Apply horizon projection only for specific POI types
    if (isHorizonCalibrated && ShouldProjectToHorizon(coord.POIType))
    {
        position = ProjectOntoHorizon(position);
    }

    GameObject prefab = poiPrefabs[coord.POIType];
    GameObject createdObj = Instantiate(prefab, position, Quaternion.identity);

    if (!spawnedObjects.ContainsKey(coord.POIType))
    {
        spawnedObjects[coord.POIType] = new List<GameObject>();
    }
    spawnedObjects[coord.POIType].Add(createdObj);
    SpawnPOIForMap(coord);
    return createdObj;
}
```

FIGURE 4.41: **Method responsible for spawning POI inside the AR environment.** This function manages the placement, configuration, and registration of Points of Interest, ensuring correct integration with both the AR scene and map interface.

- **Map Synchronization:** Calls `SpawnPOIForMap` to ensure the POI appears both in the AR environment and on the Mapbox-based mini-map, maintaining visual consistency across interfaces.

Static POIs: Preloaded Navigation Aids and Hazards

In addition to dynamic Points of Interest received from the server in real time, the system also supports a set of static POIs that are preloaded during application initialisation. These static POIs represent fixed navigation aids and permanent hazards—such as lighthouses, buoys, reefs, and wrecks—which are critical for maritime situational awareness but do not change frequently.

Each static POI is defined within the application as a `GPSCoordinate` object (see Figure 4.38), with its category specified by the `POIType` enumeration (Figure 4.37). This collection is either embedded directly in the application or loaded from a local configuration file at startup, ensuring that essential maritime features are always available regardless of connectivity.

Upon launching the AR application, the preloaded dataset is processed and each static POI's latitude and longitude are converted into Unity's local coordinate system using the `GPSToLocalAlgorithm` (Figure 4.40). The instantiated POIs are then placed in the AR scene using the same `SpawnPOIAtLocation` workflow as dynamic POIs (Figure 4.41), ensuring visual and interactive consistency across all navigation markers and hazards.

This approach guarantees that fundamental maritime landmarks are always visible and interactable in the AR environment, providing reliable baseline information for the user even in offline or degraded communication scenarios. Additionally, static POIs can be updated by modifying the preloaded dataset, allowing the system to

adapt to changes in navigational infrastructure or operational requirements without dependency on real-time data feeds.

4.8.3 POI Distance Helper Panel

To enhance users' spatial awareness and navigation efficiency, each Point of Interest (POI) is accompanied by a distance helper panel that displays the real-time, Euclidean distance in Unity's local coordinate space from the user to the POI. This floating panel is positioned above the corresponding POI, offering immediate feedback on proximity and supporting better decision-making in the maritime environment.

FIGURE 4.42: **POI Distance Panel.** Above each POI, a dynamic panel provides real-time feedback on the user's distance to the point of interest.

This feature is implemented as part of the `POIInteraction` script, which is attached to every POI object. The panel's value is continuously updated using Unity's `Vector3.Distance` method to calculate the distance between the user's position (camera) and the POI's position in the AR environment. This value is recalculated each frame thus panel always reflects the current distance as the user moves, keeping navigational information accurate and actionable.

While distance feedback is crucial for spatial awareness, in scenarios involving collision risks, mariners require more detailed, context-specific information—provided by the Conflicting Vessel Information Panel described next.

4.8.4 Conflicting Vessel Information Panel

To deliver actionable and context-specific data during collision avoidance scenarios, the AR system provides a dedicated information panel for the Conflicting Vessel POI—that is, the ship identified as being on a potential collision course with the user's vessel. This panel appears when the user maintains gaze focus on the Conflicting Vessel POI for a designated threshold (typically 2 seconds), utilizing MRTK3's eye-gaze functionality and collider detection.

Once activated, the information panel is presented in the user's field of view, displaying the most relevant real-time navigational data for the conflicting vessel. As illustrated in Figure 4.43, the following fields are shown:

- **Speed:** The current speed of the conflicting vessel
- **Position:** The real-time latitude and longitude coordinates

- **Heading:** The vessel's current heading (degrees)
- **Course:** The vessel's course over ground (degrees)
- **Distance:** The straight-line distance from the user's vessel to the conflicting vessel

FIGURE 4.43: **Conflicting Vessel Information Panel.** This dynamic panel presents live navigational data for the vessel posing a collision risk, including speed, position, heading, course, and distance.

The panel is automatically updated in real time as new data is received, ensuring that the user always has access to the latest situational information. The panel can be closed by pressing the "Close" button, keeping the interface unobtrusive.

The underlying functionality is managed by the `ShowInfo` method (see Figure 4.44), which calculates and updates all relevant fields in response to gaze interaction events. This approach streamlines the user experience by focusing attention on critical collision-avoidance data, delivered exactly when and where it is needed.

```
public void ShowInfo()
{
    if (SimplePOIInformer.Instance == null) return;
    SimplePOIInformer.Instance.ShowInfoPanel();
    SimplePOIInformer.Instance.SetInfoText(header, body);
}
```

FIGURE 4.44: **ShowInfo method.** Handles display logic and real-time updates for the Conflicting Vessel information panel.

To ensure these panels and other POI elements remain visible regardless of user movement or orientation, a billboarding technique is employed, as detailed in the following section

4.8.5 Billboarding for POI Visibility

To ensure Points of Interest (POIs) remain clearly visible and legible regardless of user orientation, a billboarding technique is employed within the AR environment.

Without billboarding, POIs may appear at arbitrary angles or become obscured, hindering situational awareness.

The billboarding functionality is implemented via a custom script that continuously rotates each POI to face the user's camera. This is achieved by updating the POI's rotation about the Y-axis in every frame using the `LateUpdate()` method (see Figure 4.45). By constraining rotation to the horizontal plane, the system avoids unwanted tilting and keeps icons upright for optimal readability.

```
void LateUpdate()
{
    if (cameraTransform == null)
    {
        cameraTransform = Camera.main.transform;
    }

    if (!isSmoothing)
    {
        // Update the object to face the camera, ignoring pitch
        Vector3 targetPosition = new Vector3(cameraTransform.position.x, transform.position.y, cameraTransform.position.z);
        targetRotation = Quaternion.LookRotation(-targetPosition + transform.position);
        transform.rotation = targetRotation;
        //Rotate parent too
        if (enableForParent && parentTranform != null)
            parentTranform.rotation = targetRotation;
    }
}
```

FIGURE 4.45: **LateUpdate-based billboarding script.** The POI's orientation is dynamically adjusted to face the camera each frame.

For scenarios requiring more gradual and visually appealing transitions, a `SmoothLookAt()` coroutine is used. This method leverages `Quaternion.Slerp` to interpolate smoothly between the POI's current rotation and the target orientation over a set duration, resulting in fluid, non-disruptive movement (see Figure 4.46).

```
private IEnumerator SmoothLookAt()
{
    isSmoothing = true;
    Quaternion initialRotation = transform.rotation;
    Vector3 targetPosition = new Vector3(cameraTransform.position.x, transform.position.y, cameraTransform.position.z);
    targetRotation = Quaternion.LookRotation(-targetPosition + transform.position);

    float elapsedTime = 0f;
    while (elapsedTime < 0.5f)
    {
        transform.rotation = Quaternion.Slerp(initialRotation, targetRotation, elapsedTime * rotationSpeed);
        if (enableForParent && parentTranform != null)
            parentTranform.rotation = transform.rotation; // Rotate parent too
        elapsedTime += Time.deltaTime;
        yield return null;
    }

    transform.rotation = targetRotation;
    if (enableForParent && parentTranform != null)
        parentTranform.rotation = transform.rotation; // Rotate parent too
    isSmoothing = false;
}
```

FIGURE 4.46: **SmoothLookAt() coroutine.** The POI rotates smoothly to face the user, enhancing visual comfort.

Both billboarding methods support rotating not only the POI itself but also its parent transform. This ensures that any attached elements, such as info panels (see Figure 4.42), remain correctly oriented towards the user, maintaining overall visual coherence in the AR scene. With the key maritime features defined as Points of Interest, the system next visualizes navigational routes that guide safe vessel passage and provide early warnings of emerging hazards.

4.9 Route Visualization in the AR Environment

Route visualization is a core feature of the head-worn AR navigation system, designed to support ship captains in collision avoidance and hazardous situation management. The system receives dynamically generated recommended routes from the backend, updating them in real time as new predictions become available. This approach ensures that navigational guidance always reflects the latest situational context, without requiring user-defined paths.

FIGURE 4.47: **In-situ AR route visualization during system development.** The AR system overlays a blue route path and its uncertainty cone as seen from the operator's perspective.

4.9.1 Overview and Design Rationale

The primary goal of the route visualization module is to deliver immediate, context-aware information for safe navigation. To address this, the system simultaneously projects both the recommended route for the user's vessel and the predicted path of any conflicting vessel directly within the captain's field of view. This dual-route display allows intuitive assessment of intersection points and vessel movements, enhancing situational awareness during potential collision scenarios.

Routes are rendered as sequences of directional arrows (waypoints), with each arrow clearly indicating the heading at every segment. To further improve spatial perception and reduce visual clutter, waypoints are rendered with a gradual vertical ascent along the Y-axis, creating a 3D effect that makes route geometry more discernible within the AR environment.

In addition to route overlays, the system visualizes uncertainty by enveloping each route with a conical or shield-shaped zone. These zones indicate the predicted margin of error, accounting for potential deviations due to GPS inaccuracies, latency, or uncertainties in the forecasting algorithms.

All route-related visual elements—including route lines, arrow-shaped waypoints, and uncertainty regions—are integrated with an interactive support panel. This panel enables captains to selectively activate or deactivate individual route elements

in real time, tailoring the display to operational needs and ensuring that information remains clear and manageable.

This section details the technical realisation and design rationale for route visualization in the developed system, including:

- The dynamic rendering and real-time updating of recommended and conflicting vessel routes, with attention to waypoint orientation and depth effects.
- The visualization of route uncertainty, encompassing methods for communicating spatial deviation and prediction confidence.
- Interactive and ergonomic features, such as the route control panel for customizing visible route elements to match situational demands.

4.9.2 Route Data Acquisition and Update Pipeline

The AR system receives navigational routes and collision-avoidance data as JSON-formatted messages, streamed from the backend server to the HoloLens client over a persistent WebSocket connection. All required parameters for route visualization are included in each message, as described in Section 4.3.

For this subsystem, the most relevant parameters are:

- `ship1_traj_lon`, `ship1_traj_lat`: Arrays representing the longitude and latitude of waypoints along the recommended route for the user's vessel.
- `ship2_traj_lon`, `ship2_traj_lat`: Arrays encoding the longitude and latitude of waypoints for the predicted route of the conflicting vessel.
- `linestring1`: A series of tuples (longitude, latitude, timestamp) describing the time-sequenced trajectory of the user vessel, as computed by the collision-avoidance algorithm.
- `linestring2`: The corresponding sequence for the conflicting vessel, capturing its predicted route and associated timing.

The update process is event-driven and originates with the Kafka producer: it waits for new AIS data updates from tracked vessels. Once an AIS position update is received, the prediction module runs to generate the latest collision-avoidance scenario and route recommendations. The Kafka producer then packages these results into a standardized JSON message and publishes it to the designated Kafka topic.

From there, the backend server acts as a consumer, relaying the message to all connected HoloLens clients via WebSocket. Upon receipt, the AR application parses the payload and updates all displayed route overlays accordingly, ensuring that navigational guidance always reflects the latest real-world situation.

Time-stamped fields and unique event identifiers within the JSON payload guarantee proper synchronization between the AR interface and real-world vessel movements, even in rapidly changing maritime environments.

4.9.3 Rendering Vessel Routes

User Vessel Route Visualization

The visualization of the recommended route for the user's vessel is accomplished through a structured process that transforms received JSON data into meaningful,

interactive holographic guidance in the AR environment. This subsection details the workflow, from parsing backend data to real-time rendering of waypoints and route lines.

Waypoint Extraction and Storage. Upon receiving a new JSON message containing route information, the application parses the relevant segments (see Section 4.3), focusing on the arrays `ship1_traj_lat`, `ship1_traj_lon`, and `linestring1`. Each set of latitude and longitude coordinates is converted into a `GPSCoordinate` object, which encapsulates the precise location and POI type associated with each route point.

These `GPSCoordinate` objects are collected in a dedicated list named `RouteCoordinates`. This list serves as the primary data structure for managing all waypoints that define the vessel's intended path, as shown in Figure 4.48.

```
[Header("Route Coordinates")]
[Tooltip("Add all object coordinates here")]
public List<GPSCoordinate> RoutecCoordinates = new List<GPSCoordinate>();
```

FIGURE 4.48: **Definition of the `RouteCoordinates` list.** This public field, visible in the Unity Inspector, collects the route's waypoints as `GPSCoordinate` objects.

Instantiation of Route Waypoints. After the route coordinates are parsed and stored, the system iterates through the `RouteCoordinates` list. For each entry, it spawns a corresponding waypoint marker in the AR environment using the `SpawnObjectAtLocation()` method. This process mirrors the workflow used for Points of Interest (Section: 4.8), ensuring that route markers benefit from the same precise spatial mapping and organizational logic (see Figure 4.41).

```
foreach (var coord in RoutecCoordinates)
{
    GameObject pointObject = SpawnObjectAtLocation(coord);
    spawnedObjects[coord.POIType].Add(pointObject);
}
```

FIGURE 4.49: **Code for spawning route waypoints.** Each waypoint in `RouteCoordinates` is processed, resulting in a marker object placed in the AR environment and cataloged by POI type.

Each waypoint marker representing the vessel's route is assigned the `POIType.MyRoutePoint` type and rendered using a distinct blue-white arrow icon (see Figure 4.50). This visual distinction ensures that navigational waypoints are immediately recognizable amidst other AR content, supporting rapid situational awareness for the operator.

Line Renderer initialisation and Assignment. To visually connect the sequence of route waypoints, the application employs a `LineRenderer` component. If a LineRenderer has not already been created for the vessel's route, it is initialized by instantiating a new GameObject, attaching a LineRenderer, and assigning a semi-transparent, light blue material for optimal contrast and clarity in the AR environment. The initialisation process is depicted in Figure 4.51.

FIGURE 4.50: **Arrow icon for route waypoints.** Each point along the recommended route is visualized using this distinctive arrow model, facilitating rapid identification.

```
if (myRouteLineRenderer == null)
{
    // Create a LineRenderer for MyRoute if not assigned
    GameObject myRouteLineObj = new GameObject("MyRouteLineObject");
    myRouteLineRenderer = myRouteLineObj.AddComponent<LineRenderer>();
    myRouteLineRenderer.material = new Material(Shader.Find("Sprites/Default"));
}
```

FIGURE 4.51: **Creation and configuration of the `LineRenderer`.** This code ensures that a route line is always available to connect waypoints.

Real-Time Route Updating. As the positions of route waypoints may change due to updates from the backend, the system must ensure that the visual route line remains accurate at all times. This is achieved by continuously updating the `LineRenderer`'s positions in the `Update()` loop. For each frame, the system counts the number of `MyRoutePoint` markers and sets the LineRenderer's position array accordingly. If any waypoint's position is updated—such as during dynamic rerouting or data refresh—the route line instantly reflects the new configuration. Figure 4.52 shows the relevant code logic.

```
private void Update()
{
    int myRoutePointCount = spawnedObjects[POIType.MyRoutePoint].Count;
    int otherRoutePointCount = spawnedObjects[POIType.OtherRoutePoint].Count;

    if (myRoutePointCount > 0)
    {
        // Update certain line renderer positions (no width change)
        myRouteLineRenderer.positionCount = myRoutePointCount;
        for (int i = 0; i < myRoutePointCount; i++)
        {
            if (spawnedObjects[POIType.MyRoutePoint][i] != null)
            {
                myRouteLineRenderer.SetPosition(i, spawnedObjects[POIType.MyRoutePoint][i].transform.position);
            }
        }
```

FIGURE 4.52: **Real-time updating of the route line.** The `Update()` method ensures that the line remains connected to each waypoint, adapting instantly to any changes in position or route composition.

Visual Output. The combination of discrete arrow markers and the connecting route line provides a clear, intuitive representation of the vessel's intended trajectory. To optimize situational awareness, the route line is rendered using a semi-transparent light blue material. This deliberate design choice ensures that the digital overlay

does not obscure the real-world view through the bridge windows, allowing the captain to maintain visibility of physical hazards—such as small boats or floating obstacles—that may not be represented in AIS or other digital data streams. The transparency level balances guidance clarity with operational safety, so that virtual elements never conceal critical real-world cues.

The system's architecture allows for seamless integration of updated navigation guidance, supporting both static and highly dynamic maritime operations. In summary, the route visualization workflow transforms backend trajectory data into interactive, visually distinct waypoints and connecting lines, ensuring that ship captains receive clear, up-to-date guidance at all times. The next subsection will discuss the parallel visualization of the predicted route for conflicting vessels and additional enhancements for route clarity and situational awareness.

Conflicting Vessel Route Projection

The visualization of the predicted route for a conflicting vessel closely mirrors the pipeline and structure described in the previous subsection for the user vessel. This deliberate parallelism ensures consistency in user experience and minimizes cognitive load, allowing officers to rapidly interpret both their own trajectory and that of any vessel posing a potential collision risk.

Data Parsing and Object Instantiation. Upon receipt of a new JSON update, the AR application extracts the `ship2_traj_lat`, `ship2_traj_lon`, and `linestring2` fields, each corresponding to the predicted waypoints and trajectory of the opposing vessel as calculated by the backend prediction module. Each tuple of latitude and longitude is converted into a `GPSCoordinate` instance and appended to a dedicated list (see Figure 4.48). This process is performed identically to the user's own route, leveraging the same data handling structures and instantiation logic for consistency and code maintainability.

For each point in this list, the system calls the same `SpawnObjectAtLocation()` function as before, generating a 3D arrow marker at each waypoint (see Figure 4.49). These markers are assigned the POIType `OtherRoutePoint`, clearly distinguishing them from those used in the user's own recommended route.

Visual Output and Color Coding. To ensure immediate visual discrimination between the user's route and the predicted path of the conflicting vessel, all elements associated with the opposing vessel are rendered in a highly visible red theme:

- **Route Line:**: The trajectory of the opposing vessel is drawn using a `LineRenderer` with a semi-transparent red material (see Figure 4.54). This choice of color, in strong contrast to the blue used for the user route, serves as an intuitive risk indicator and instantly draws the operator's attention to potential threats.
- **Arrow Markers:** The 3D arrow icons denoting each route segment are colored bright red with a distinctive outline (see Figure 4.53), matching the overall hazard convention and further reducing the risk of misinterpretation.
- **Uncertainty Region:** The predicted uncertainty or "cone of risk" surrounding the opposing route is likewise shaded with a translucent red, maintaining consistency in the system's visual language.

FIGURE 4.53: **Arrow icon for the conflicting vessel's route.** The red coloring with a clear outline provides rapid visual distinction.

```
if (otherRouteLineRenderer == null)
{
    // Create a LineRenderer for OtherRoute if not assigned
    GameObject otherRouteLineObj = new GameObject("OtherRouteLineObject");
    otherRouteLineRenderer = otherRouteLineObj.AddComponent<LineRenderer>();
    otherRouteLineRenderer.material = new Material(Shader.Find("Sprites/Default"));
}
```

FIGURE 4.54: **Line renderer initialisation for the conflicting vessel.** The renderer is configured with a semi-transparent red material for clear visibility without obscuring the real-world view.

Continuous Update and Real-Time Feedback. The process of rendering and updating the conflicting vessel's route is event-driven and fully automated: every time a new scenario is predicted and broadcast via the Kafka/WebSocket pipeline, the AR system refreshes the visual elements—route line, arrows, and uncertainty cone—so that the captain always sees the most up-to-date situation.

Field Demonstration. Figure 4.55 demonstrates the AR overlay as seen during a live field test onboard a vessel. The red route and accompanying uncertainty region are clearly visible through the bridge window, providing immediate feedback to the bridge team while maintaining situational awareness of the real-world environment outside. The choice of semi-transparent overlays ensures that even with the addition of digital guidance, critical external cues—such as small vessels without AIS transponders—remain unobstructed and visible to the user.

4.9.4 Uncertainty Visualization Along Routes

A critical aspect of safe maritime navigation—especially in congested or dynamically changing environments—is the ability to visualize not only the planned route, but also the uncertainty associated with predicted vessel positions. The AR navigation system addresses this challenge by enveloping each recommended route with a visually distinctive "uncertainty region"—a semi-transparent, cone-shaped area that widens with distance into the prediction horizon. This gives officers immediate, intuitive awareness of where the vessel is *likely* to be, accounting for both GPS error and the inherent limitations of real-time forecasting.

Parameters Influencing Uncertainty. The width and progression of the uncertainty region are influenced by several operational factors:

- **GPS Position Error:** Consumer-grade GPS can introduce meter-level inaccuracies due to signal noise, multipath effects, and environmental obstructions.

FIGURE 4.55: **Conflicting vessel route visualized from the bridge at sea.** The system maintains clarity and immediate recognizability under operational conditions.

- **Prediction Model Limitations:** The collision-avoidance algorithm makes predictions based on current AIS positions and vessel speeds, but real-world dynamics (e.g., current, wind, helmsman response) introduce additional uncertainty as forecast duration increases.
- **Communication Latency:** Delays in AIS updates or network communication can further desynchronize the predicted and actual vessel positions, particularly during abrupt maneuvers.

As a result, the system deliberately increases the width of the uncertainty cone or route region as it moves further from the current location—visually encoding diminishing confidence with greater spatial "spread."

Rendering the Uncertainty Region. The uncertainty zone for each route is rendered as a semi-transparent overlay, using Unity's `LineRenderer` component. This overlay is dynamically generated and updated on every frame to ensure that it always tracks the current recommended route, adapting in real time to new predictions and environmental inputs.

Implementation Workflow. The rendering logic is handled by a dedicated LineRenderer, as shown in Figure 4.56. When the AR application processes new route data, it checks for an existing uncertainty LineRenderer instance and initializes one if necessary. The overlay material is chosen for high transparency, ensuring that the uncertainty zone does not obscure critical real-world visual cues, such as small vessels or floating obstacles that may lack AIS transponders.

```
if (myRouteUncertainLineRenderer == null)
{
    // Create a LineRenderer for MyRouteUncertain if not assigned
    GameObject myRouteUncertainLineObj = new GameObject("MyRouteUncertainLineObject");
    myRouteUncertainLineRenderer = myRouteUncertainLineObj.AddComponent<LineRenderer>();
    myRouteUncertainLineRenderer.material = new Material(Shader.Find("Sprites/Default"));
}
```

FIGURE 4.56: **initialisation of the uncertainty route LineRenderer.** The code ensures a dedicated renderer exists for the uncertainty region, applying a transparent material for unobtrusive overlay.

Progressive Width Control. A key feature of the uncertainty visualization is its progressive widening, which communicates to the user that positional confidence decreases with prediction horizon. This effect is achieved using the `SetLineRendererWidthProgressive` function (Figure 4.57), which applies an animation curve to the LineRenderer, interpolating the width from a narrow base at the vessel's current position to a broader cone at the projected endpoint.

```
private void SetLineRendererWidthProgressive(LineRenderer lineRenderer, float startWidth, float endWidth)
{
    // Set the start and end width of the LineRenderer to different values for a progressive effect
    lineRenderer.startWidth = startWidth;
    lineRenderer.endWidth = endWidth;

    // Create a width curve to make the line progressively wider
    AnimationCurve widthCurve = new AnimationCurve();
    widthCurve.AddKey(0.0f, startWidth);
    widthCurve.AddKey(1.0f, endWidth);

    // Apply the width curve to the LineRenderer
    lineRenderer.widthCurve = widthCurve;
}
```

FIGURE 4.57: **Function to progressively widen the uncertainty overlay.** This approach visually encodes increased uncertainty with distance, directly reflecting the output of the prediction model.

This dynamic adaptation is critical: it enables mariners to quickly interpret the operational "envelope" for safe passage, assess potential collision risk under uncertainty, and maintain a heightened level of situational awareness during navigation.

Visual Output and Operational Impact. In the field, this uncertainty region appears as a gently expanding, translucent "cone" overlaid directly onto the real sea view (see Figure 4.47 and Figure 4.58). Its semi-transparent coloration ensures that the underlying environment remains visible at all times—an essential safety feature, as smaller objects or vessels without AIS cannot be digitally tracked and must always remain in the captain's unobstructed field of view.

By integrating real-time uncertainty visualization into the AR route display, the system provides mariners with actionable insight into the reliability of each recommended path. The design choices—progressive width, high transparency, and real-world alignment—ensure that safety is never compromised, and that the digital overlay enhances, rather than distracts from, the captain's situational awareness.

FIGURE 4.58: **Uncertainty region projected through the bridge window during an at-sea field test.** The overlay is clearly visible in varying conditions, yet remains non-intrusive, ensuring that all physical hazards stay in view.

4.9.5 Depth Perception and Waypoint Arrangement

One of the unique challenges of head-worn AR navigation, particularly with Microsoft HoloLens 2, is the device's limited depth perception beyond a few meters. In maritime scenarios, route overlays may extend many kilometers ahead, making it difficult for the user to intuitively gauge the spatial layout and direction of the recommended path. To overcome this limitation, the AR system implements a vertical arrangement for route waypoints, enhancing the perceived 3D structure of the route and improving spatial clarity.

Vertical Ascent for Waypoints. Instead of placing all route markers at sea level, the system gradually increases the Y-axis (vertical) position of each waypoint as the distance from the vessel grows. The closest waypoint appears just above the ship, while the farthest waypoint—typically placed at the horizon line—has the highest elevation within the AR scene. All intermediate points are automatically assigned heights proportional to their distance between the start and end points—ensuring that the ascent forms a clear trajectory from the ship to the horizon, that mimics the perspective of looking "out to sea." This arrangement gives the user an immediate sense of both the direction and extent of the planned route, making the spatial layout far more legible in the HoloLens display.

Implementation: Coupling POI Movement with LineRenderer. Because the route lines themselves are rendered dynamically (updated each frame in the `Update()` function), the system simply moves the POIs (route points) to their calculated Y-positions. As the POIs move vertically, the LineRenderer automatically updates the connecting line, creating a continuous, ascending route overlay that visually pops out from the sea surface.

Directional Arrow Adjustment. Each waypoint along the route is rendered not just as a simple marker, but as an arrow icon. To ensure these arrows always point in the

correct direction of travel, the system uses the `AdjustDirectionsForPoints` function, shown in Figure 4.59. This function iterates through the list of route points (POIs), orienting each arrow so that it "looks at" the next point in the sequence, while the final arrow is aligned according to the direction from the second-to-last point—preserving continuity at the route's end. Importantly, `AdjustDirectionsForPoints` is invoked inside the `Update()` method, so that whenever a waypoint's position is altered (for example, if new prediction data is received), the arrow's orientation is immediately recalculated and updated. This dynamic adjustment ensures that the direction of each route segment is always visually accurate, providing immediate and intuitive guidance for the user.

```
void AdjustDirectionsForPoints(POIType pointType)
{
    var pointsList = spawnedObjects[pointType];

    for (int i = 0; i < pointsList.Count - 1; i++)
    {
        Transform currentPoint = pointsList[i].transform;
        Transform nextPoint = pointsList[i + 1].transform;

        Vector3 direction = nextPoint.position - currentPoint.position;
        if (direction != Vector3.zero)
        {
            currentPoint.LookAt(nextPoint);
        }
    }

    // Handle the last object separately
    if (pointsList.Count > 1)
    {
        Transform lastPoint = pointsList[pointsList.Count - 1].transform;
        Transform secondLastPoint = pointsList[pointsList.Count - 2].transform;

        Vector3 lastDirection = secondLastPoint.position - lastPoint.position;
        if (lastDirection != Vector3.zero)
        {
            lastPoint.rotation = secondLastPoint.rotation;
        }
    }
}
```

FIGURE 4.59: **Implementation of the `AdjustDirectionsForPoints` function.** Each arrow is dynamically rotated to point toward the next waypoint, ensuring correct route heading visualization.

Disambiguation of Overlapping Route and Uncertainty Route. To avoid visual overlap between the main route and its associated uncertainty region, the update process—executed each frame—computes the world-space position for each waypoint, then applies a vertical offset along the Y-axis to the uncertainty region (see Figure 4.60). This separation ensures that the uncertainty overlay remains clearly visible above the baseline route, even when segments closely follow or cross one another. By maintaining this vertical distinction, the system prevents visual confusion and preserves the clarity of both route and uncertainty cues in the user's field of view.

By combining vertical arrangement of waypoints and dynamic direction adjustment, the AR navigation system transforms what would otherwise be a flat, difficult-to-interpret overlay into a richly informative 3D guidance structure. This approach maximizes the utility of the HoloLens display for real-world ship navigation, supporting faster decision-making and more intuitive route following, even across extended distances.

```
myRouteUncertainLineRenderer.positionCount = myRoutePointCount;
SetLineRendererWidthProgressive(myRouteUncertainLineRenderer, myRouteUncertainStartWidth, myRouteUncertainEndWidth);
for (int i = 0; i < myRoutePointCount; i++)
{
    if (spawnedObjects[POIType.MyRoutePoint][i] != null)
    {
        Vector3 offsetPosition = spawnedObjects[POIType.MyRoutePoint][i].transform.position;
        offsetPosition.y += myRouteUncertainYOffset;
        myRouteUncertainLineRenderer.SetPosition(i, offsetPosition);
    }
}
```

FIGURE 4.60: **Frame-by-frame update of the uncertainty overlay.** Each point in the uncertainty region is offset above its corresponding route point, maintaining clear visibility regardless of route complexity.

4.9.6 Interactive Features and Route Control Panel

To ensure ergonomic operation and adaptability to diverse navigational scenarios, the AR navigation system integrates an interactive control panel that allows captains to selectively manage the visual layers associated with route guidance. This interactive layer is essential for maintaining clarity in complex environments and empowering the operator to focus only on information relevant to the situation at hand.

Panel Functionality and User Experience. The control panel (Figure 4.62) is accessible directly within the AR interface and designed for hands-free interaction, utilizing gaze or gesture inputs compatible with HoloLens 2. It features a grid of toggle switches, each corresponding to a specific visual component of the navigation overlay:

- Suggested Route Points (user)
- Suggested Route (user)
- Uncertainty Route (user)
- Other Ship's Suggested Route Points
- Other Ship's Suggested Route
- Other Ship's Uncertainty Route
- Uncertainty Circle (user)
- Other Ship's Uncertainty Circle
- Disable All Route Elements

Each toggle directly controls the visibility of its associated GameObjects. For example, deactivating the "Suggested Route" switch hides the route line from the user's field of view while preserving other overlays. This fine-grained control helps the captain de-clutter the AR display, especially when focusing on critical navigational cues.

Implementation and Responsiveness. The toggling logic is implemented in code, as illustrated in Figure 4.61. When a toggle is changed, the corresponding function iterates through the relevant child objects (route lines, waypoints, or uncertainty

overlays) and sets their visibility accordingly. This operation is lightweight and performed in real-time, ensuring the interface responds instantly to the user's preferences.

```
public void ToggleSuggestedRouteLineOnMap(bool isActive)
{
    if (mapGameObject != null)
    {
        foreach (Transform child in mapGameObject.transform)
        {
            if (child.name.Contains("SuggestedRouteLine"))
            {
                child.gameObject.SetActive(isActive);
            }
        }
    }
}
```

FIGURE 4.61: **Implementation of route line toggling logic.** The function activates or deactivates elements within the map based on user input.

FIGURE 4.62: **Interactive route element toggle panel in AR.** Each switch allows for precise control over displayed overlays, supporting rapid de-cluttering and information tailoring.

Example Use Scenario. Consider a scenario where the bridge is crowded with multiple route overlays and uncertainty regions during a close-quarters encounter. The captain can quickly disable non-essential elements—such as the opposing vessel's uncertainty cone or suggested points—using the panel, allowing immediate focus on their own recommended route and any real-world visual cues outside the window. Conversely, during a complex collision-avoidance maneuver, the captain may opt to display all overlays to comprehensively understand risk zones and vessel trajectories. This flexibility enables the system to adapt to both routine and emergency operations.

The interactive route control panel is a critical usability feature, providing mariners with intuitive, real-time customization of the AR environment. Its design balances the need for information richness with clarity, ensuring that decision-critical cues are never lost amid digital clutter.

4.10 Augmented Reality Map (Mapbox Integration)

The interactive map is a central element of the AR navigation system, providing a comprehensive, real-time overview of the user's environment and enhancing situational awareness during maritime operations. Unlike conventional digital maps, this map is anchored within the AR workspace—placed above the bridge windows at the user's preferred position—and is always visible for quick reference, minimizing distraction and supporting seamless task flow.

The map leverages the Mapbox SDK for Unity, which enables dynamic rendering of nautical charts, real-time user location tracking, visualization of Points of Interest (POIs), recommended and opposing vessel routes, and identified danger zones. This integration allows for a unified spatial context, combining live sensor data with static geospatial layers, and providing critical support for decision-making under uncertain or rapidly changing conditions.

FIGURE 4.63: **AR map interface anchored above the bridge windows.** The digital map provides real-time positional and situational data without obstructing the operator's primary field of view.

This section covers:

- The technical approach for integrating Mapbox with Unity, including API configuration and map initialisation.
- Key map features: real-time user location, dynamic POI display, route and danger area overlays.
- Interaction design, map placement ergonomics, and how the fixed map supports continuous, distraction-free access.
- The architecture for updating map content, focusing on script management and data flow for seamless real-time integration.

4.10.1 Integration of Mapbox SDK in Unity

To enable advanced mapping capabilities within the AR navigation system, the Mapbox SDK was integrated directly into the Unity project. Mapbox's robust toolkit provides real-time rendering, dynamic data overlays, and customization options that make it highly suitable for maritime AR scenarios.

Integration Workflow

The Mapbox SDK was incorporated into Unity through the following process:

- **SDK Installation:** The Mapbox Unity SDK was downloaded from the official Mapbox website and imported into the Unity project. The SDK includes a suite of prefabs and components designed for rapid map deployment.
- **Map initialisation:** After installation, a Mapbox map prefab was added to the Unity scene. This prefab forms the interactive base map, allowing for extensive configuration of visual style, zoom, map type, and data layers.
- **API Key Management:** To enable communication with Mapbox services, a valid API access token was generated and inserted into the Unity project via the Mapbox configuration window. This key authenticates map requests and enables the retrieval of up-to-date map tiles and data overlays.

Mapbox Functionality in the AR Application

With the SDK in place, the AR map panel offers the following real-time features:

- **User Positioning:** The current location of the vessel or user is rendered in real time, continuously updated via live GPS input.
- **Dynamic POI Markers:** Points of Interest (POIs)—including ships, hazards, and navigational aids—are visualized as interactive icons on the map.
- **Hazardous Area Visualization:** Designated danger zones—such as shallow waters, restricted areas, or regions of increased risk—are projected directly onto the map as shaded polygons or icons, supporting rapid threat identification.
- **Route and Uncertainty Overlays:** The recommended route, the predicted path of opposing vessels, and their associated uncertainty regions are rendered atop the map in real time, providing a comprehensive view of navigation plans and risk envelopes.

4.10.2 Map Features

The interactive map within the AR navigation system provides ship captains and crew with essential real-time situational awareness. By integrating live vessel data, Points of Interest (POIs), and hazard overlays, the map serves as a crucial decision-support tool during navigation and collision avoidance scenarios.

User Vessel Location

The map continuously displays the real-time GPS location of the user's vessel, as reported by the onboard navigation systems and relayed through the server infrastructure. This location marker is updated with each incoming data packet, ensuring

the displayed position remains synchronized with the vessel's actual movements and course changes.

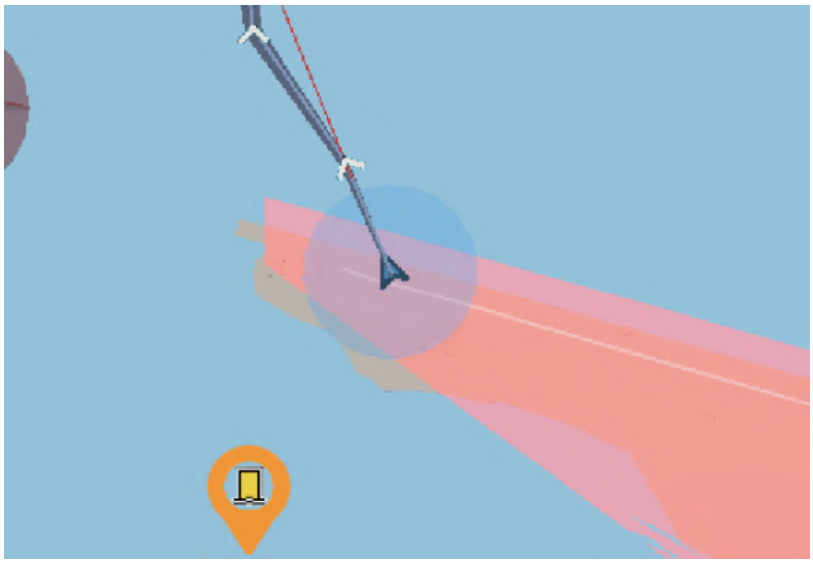


FIGURE 4.64: **Live vessel location on the AR map.** The user's current position is visualized with a distinct blue arrow icon for immediate reference.

Points of Interest (POIs)

The AR map overlays a range of Points of Interest, including other vessels, navigational aids, ports, and hazard zones—each represented by unique icons. POIs are dynamically updated according to their current GPS coordinates, giving the bridge team a comprehensive, up-to-date view of their operational environment and allowing for rapid assessment of surrounding traffic, key locations, and potential risks.

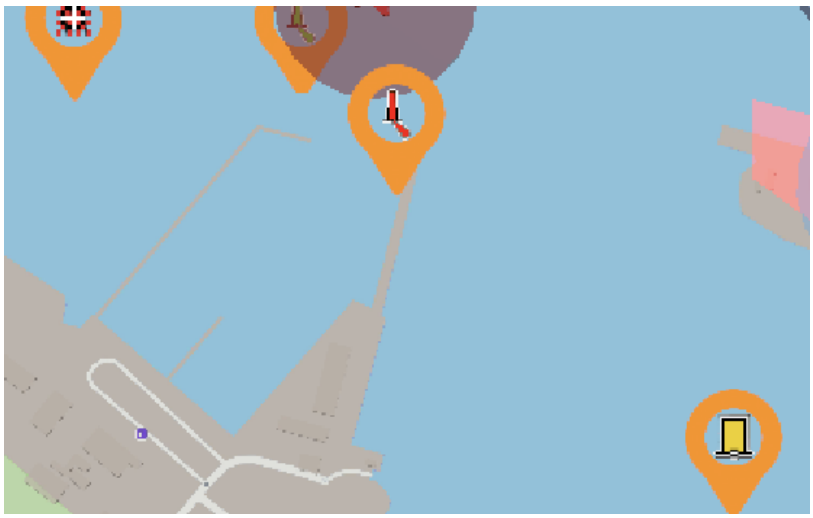


FIGURE 4.65: **Comprehensive Points of Interest (POI) visualization on the AR map.** Distinctive icons denote different types of POIs, including vessels, ports, navigational aids, and hazard zones, all dynamically positioned according to real-time GPS data.

Toggle POIs

A notable feature of the AR navigation system is the ability to dynamically filter and manage Points of Interest (POIs) using a dedicated control panel. During the initial calibration, users can position this panel anywhere within their workspace—placement next to the AR map is recommended for quick access.

The POI panel provides individual toggle switches for each POI category, as shown in Figure 4.66. This allows the operator to selectively show or hide categories such as collision points, rocks, wrecks, lighthouses, buoys, or other vessels, according to situational needs. For example, a captain navigating in open water may choose to hide non-critical POIs like special purpose buoys and focus only on hazards and vessel traffic.

Crucially, the filtering applies both to the AR map overlay and to the full 3D augmented reality scene, ensuring a consistent and decluttered visual experience. By tailoring visible information in real time, the system supports enhanced situational awareness and user-defined focus, reducing cognitive load in complex operational environments.

FIGURE 4.66: **Interactive POI filtering panel.** Users can enable or disable specific types of Points of Interest, such as lighthouses, buoys, wrecks, rocks, and hazard points, ensuring the display remains relevant and manageable during navigation.

Danger Areas and Hazard Overlays

A key feature of the AR map is the dynamic visualization of danger areas—zones identified as potential navigational hazards, such as restricted regions, shallow waters, collision risk zones, or other critical maritime dangers. These areas are rendered directly on the map as prominent colored polygons or circles, typically in red, as illustrated in Figure 4.67. The clear, high-contrast overlays ensure that such hazards are immediately visible to the bridge team, supporting timely course adjustments when approaching high-risk zones. The precise location and shape of each hazard is determined from both static nautical data and real-time predictive algorithms, ensuring that the map always reflects up-to-date risks in the operational environment.

Route and Uncertainty Route Overlays

In addition to POIs and hazard areas, the AR map visualizes both the recommended route for the user's vessel and the predicted routes for other vessels, with each trajectory shown as a colored line—blue for the user and red for conflicting vessels, as seen in Figure 4.67. These lines enable immediate recognition of navigation paths and potential intersection points at a glance.

To enhance clarity, small arrow icons are placed along each route at the waypoint positions, indicating the recommended direction of travel for every segment. These arrows are clearly visible in Figure 4.67, providing operators with quick and intuitive understanding of heading and maneuvering instructions.

Additionally, each route is accompanied by a semi-transparent "uncertainty region"—depicted as a widening, shaded cone around the predicted path. This overlay visually communicates forecast uncertainty, which increases with prediction distance due to GPS inaccuracies, data latency, or modeling limitations. Both the blue and red routes in Figure 4.67 are surrounded by these uncertainty regions, enabling users to directly assess areas of reduced confidence and proactively manage collision risk.

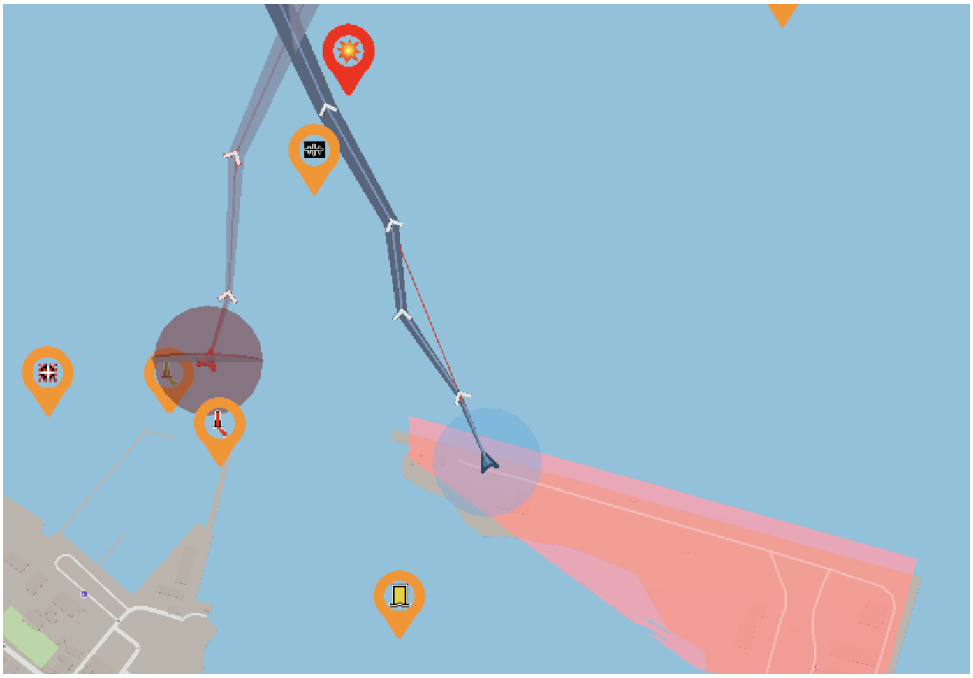


FIGURE 4.67: **Comprehensive AR map overlays for routes, uncertainty, and hazards.** Blue and red lines indicate predicted routes for the user and conflicting vessels, with small directional arrows marking each route segment. Semi-transparent shaded regions visualize route uncertainty, while prominent red overlays highlight danger zones.

Toggle Route Elements

Just as with the POI toggle panel, the AR navigation system provides a dedicated control panel for managing the visibility of all route-related elements. During the calibration stage, users can freely position this panel anywhere in their workspace,

and it is recommended to place it for optimal accessibility—ideally, one toggle panel on the left and the other on the right side of the AR map for balanced workflow.

As illustrated in Figure 4.62, this interactive panel allows users to selectively enable or disable visual elements such as suggested route points, route lines, uncertainty regions, and corresponding overlays for other vessels. Each switch operates in real time, providing immediate feedback and letting the bridge team quickly tailor the display to operational needs—whether that means minimizing clutter or revealing comprehensive situational data.

This mirrored arrangement of the route element and POI control panels ensures that all critical filtering options are always within easy reach, supporting both ergonomic operation and rapid information management.

4.10.3 Technical Implementation

The interactive map's advanced features—including real-time vessel tracking, dynamic POI visualization, and overlay updates—are made possible through a combination of Mapbox SDK functionality and custom Unity scripting. Below, the main technical components are described.

User Location Tracking

To accurately represent the user's real-time position on the AR map, GPS data is continuously transmitted from the Android mobile application to the HoloLens device. This communication occurs via a dedicated WebSocket channel managed by the server.

The server acts as a relay: when a new GPS message arrives from the mobile app, the relevant fields (latitude, longitude) are extracted and forwarded to the paired HoloLens client. The client then updates the user's marker on the Mapbox-powered map accordingly, ensuring that location visualization always reflects the most current data.

Receiving GPS Data

The HoloLens application receives the GPS data from the server in the form of messages. The HandleMessage() function is responsible for processing these messages. When a message containing GPS coordinates is detected, the message is parsed to extract the latitude and longitude values, which are then passed to the mapManager to update the user's position on the map.

Map Update Mechanism

Upon receiving updated latitude and longitude data, the `UpdatePlayerLocation()` function is invoked within the map management script. As shown in Figure 4.68, this method leverages Mapbox utilities to keep the map view and the user's avatar in sync with real-world movements.

Key steps in the location update routine:

- The `SetCenterLatitudeLongitude()` method re-centers the map on the latest user position, ensuring the avatar remains in view.
- `GeoToWorldPosition()` converts the GPS coordinates into Unity's world-space, accurately mapping real locations onto the digital environment.

```
public void UpdatePlayerLocation(Vector2d coordinates)
{
    if (!mapInitialized) return;
    try
    {
        if (MapGameobject.activeSelf)
        {
            map.SetCenterLatitudeLongitude(coordinates);
            map.UpdateMap();
            Player.transform.position = map.GeoToWorldPosition(coordinates);
            Debug.Log("Player location:" + Player.transform.position);
        }
    }
    catch (System.NullReferenceException)
    {
        Debug.LogWarning("Map is not initialized yet");
    }
}
```

FIGURE 4.68: **User location update routine.** The function centers the map and positions the user avatar based on the most recent GPS coordinates.

- The user's `transform.position` is then updated to this calculated world position, resulting in immediate visual feedback on the map as the user moves.

This workflow ensures that both the map display and the user's representation are always aligned with real-time sensor input, maintaining situational awareness throughout navigation.

Points of Interest Map Implementation

Points of Interest (POIs) play a central role in enhancing situational awareness, providing users with instant access to location-based information such as navigational aids, hazard areas, ports, and other vessels. In the AR map, POIs serve as both static and dynamic markers, supporting real-time decision-making on the bridge.

There are two primary categories of POIs in the system:

- **Static POIs:** These include fixed features such as lighthouses, buoys, or known hazards, which are preloaded into the application and rendered as soon as the map initializes.
- **Dynamic POIs:** These represent elements such moving vessels, emergent hazards, or temporary danger zones. Dynamic POIs are delivered through the real-time data pipeline: they are streamed from backend services to Kafka topics, relayed over WebSocket, and instantiated in Unity on the HoloLens client as described in Section 4.8.2.

Each POI is visualized using a distinctive icon or marker, allowing users to immediately distinguish between types of objects or risks. The icons are placed on the map at their corresponding GPS coordinates, and are dynamically updated as new data is received, ensuring that all displayed information accurately reflects the latest situational context.

This dual approach—combining static, preloaded information with live, dynamic updates—ensures that the AR map always provides comprehensive and current awareness, supporting effective navigation and risk management.

Spawning POIs on the Map

When POI data arrives via the real-time pipeline (see Section 4.8.2), it is processed and passed to the `SpawnPOIFromString()` method. This function delegates placement to `SpawnPOIForMap()`, which checks for valid reference coordinates before instantiating the correct POI prefab at the appropriate GPS location on the map (Figure 4.69).

```
public void SpawnPOIForMap(GPSCoordinate coord)
{
    if (referenceCoordinatesSet)
    {
        try
        {
            GameObject prefab = mapPrefabs[coord.POIType];
            GameObject createdObj = Instantiate(prefab, mapGameObject.transform);
            mapGameObject.GetComponentInParent<MapAndPlayerManager>().spawnedPOIs.Add(createdObj, new Vector2d(coord.Latitude, coord.Longitude));
            mapGameObject.GetComponentInParent<MapAndPlayerManager>().UpdateSpawnedPOIs();
        }
        catch (Exception ex)
        {
            Debug.LogError("Failed to spawn map point object: " + ex.ToString());
        }
    }
    else
    {
        Debug.LogError("Reference coordinates not set");
    }
}
```

FIGURE 4.69: **POI instantiation logic.** The script handles type selection and initial placement of POI objects on the AR map.

Prefabs are chosen from a pool based on POI type and are added to an active list for ongoing management. After instantiation, the `UpdateSpawnedPOIs()` method in the MapManager ensures each POI's map position is synchronized with its real-world GPS coordinates, keeping the map up to date as the user moves or new information is received (Figure 4.70).

```
public void UpdateSpawnedPOIs()
{
    if (!mapInitialized || !MapGameobject.activeSelf) return;

    foreach (KeyValuePair<GameObject, Vector2d> poi in spawnedPOIs)
    {
        poi.Key.transform.position = map.GeoToWorldPosition(poi.Value);
        poi.Key.transform.position = new Vector3(poi.Key.transform.position.x, poi.Key.transform.position.y + 0.01f, poi.Key.transform.position.z);
    }
}
```

FIGURE 4.70: **Map POI position updater.** This method keeps all POIs correctly aligned with real-world locations as the user or map view changes.

POI Data Structure

Each POI leverages a lightweight `GPSCoordinate` class, containing essential information such as latitude, longitude, POI type. This streamlined structure enables efficient placement and consistent display of POIs both on the map and in the 3D AR view. Further details on POI management are presented in the dedicated Points of Interest section 4.8.2.

Hazardous Area Rendering and Polygon Overlay

Beyond points and icons, the AR map supports the visualization of hazardous areas—critical for alerting the bridge crew to spatially extensive risks such as shallow waters, exclusion zones, and real-time danger regions. These areas are rendered as colored polygons (commonly in red for high risk) and dynamically projected onto the map using custom mesh generation in Unity.

Dynamic Area Rendering Workflow. The process of displaying a hazard area begins when the system receives a new set of coordinates describing the vertices of a danger zone. These vertices are streamed through the same data pipeline as POIs, processed on the HoloLens, and rendered on the AR map.

The main rendering logic is managed by the `UpdateSpawnedAreas()` function, shown in Figure 4.71. For each defined hazard area, the method:

```
public void UpdateSpawnedAreas()
{
    if (!mapInitialized || !MapGameobject.activeSelf)
        return;
    foreach (var kvp in spawnedAreas)
    {
        GameObject areaObj = kvp.Key;
        List<Vector2d> latLonVertices = kvp.Value;
        MeshFilter meshFilter = areaObj.GetComponent<MeshFilter>();
        if (meshFilter == null)
            continue;
        Transform parentT = areaObj.transform.parent;
        if (!parentT)
            continue;
        Vector3[] vertices = new Vector3[latLonVertices.Count];
        for (int i = 0; i < latLonVertices.Count; i++)
        {
            Vector3 worldPos = map.GeoToWorldPosition(latLonVertices[i]);
            worldPos.y += 0.01f;
            Vector3 localPos = parentT.InverseTransformPoint(worldPos);
            vertices[i] = localPos;
        }
        int[] triangles = Triangulate(vertices.Length);
        Mesh mesh = meshFilter.sharedMesh;
        if (mesh == null)
        {
            mesh = new Mesh();
            meshFilter.sharedMesh = mesh;
        }
        else
        {
            mesh.Clear();
        }
        mesh.vertices = vertices;
        mesh.triangles = triangles;
        mesh.RecalculateNormals();
    }
}
```

FIGURE 4.71: **Hazard area update and mesh assignment.** The `UpdateSpawnedAreas()` function converts coordinate lists to map overlays

- Extracts the list of latitude and longitude vertex coordinates for the area.

- Converts each coordinate to world space using Mapbox's `GeoToWorldPosition()` method, with a slight vertical offset to ensure the overlay sits above the map surface.
- Transforms the world-space vertices into local coordinates relative to the map's parent object.
- Collects these positions into a mesh vertex array.
- Triangulates the polygon to define the interior faces, and assigns these triangles and vertices to the area's mesh for visualization.

Polygon Triangulation. At the heart of rendering hazard zones as map overlays is the triangulation of their vertices into a mesh suitable for display. This is handled by the `Triangulate()` method (see Figure 4.72). For simple convex polygons, this method generates triangles by connecting the first vertex to each successive pair of vertices—effectively creating a fan of triangles covering the polygon area.

```
private int[] Triangulate(int vertexCount)
{
    List<int> triangles = new List<int>();
    for (int i = 1; i < vertexCount - 1; i++)
    {
        triangles.Add(0);
        triangles.Add(i);
        triangles.Add(i + 1);
    }
    return triangles.ToArray();
}
```

FIGURE 4.72: **Triangulation logic for area overlays.** The `Triangulate()` function creates a mesh for rendering polygons, connecting the area's vertices into triangles.

Route and Uncertainty Overlays

The dynamic rendering of vessel routes and their associated uncertainty regions on the AR map is achieved through a coordinated set of scripts that manage the positioning, orientation, and visual connection of route waypoints. The following figures illustrate the principal methods used to realize this functionality.

UpdateLineRenderers() Function. As shown in Figure 4.73, the `UpdateLineRenderers()` method is responsible for synchronizing all map route overlays with real-time data. The function executes the following key steps:

- **Collecting Waypoints:** It iterates through all spawned Points of Interest (POIs), classifying them into user or opposing vessel route points, and recording their current positions.
- **Updating Route Lines:** The positions are passed to Unity's `LineRenderer` components, which draw the main route lines on the map for both vessels.
- **Generating Uncertainty Regions:** The function calls `OffsetPositions()` to create vertically shifted versions of the route waypoints. These are then used to render the uncertainty cones above each route line, ensuring they remain visually separated for clarity.

```
private void UpdateLineRenderers()
{
    List<Vector3> myRoutePositions = new List<Vector3>();
    List<Vector3> otherRoutePositions = new List<Vector3>();

    List<GameObject> myRouteObjects = new List<GameObject>();
    List<GameObject> otherRouteObjects = new List<GameObject>();

    myRoutePositions.Add(Player.transform.position);
    myRouteObjects.Add(Player);

    foreach (KeyValuePair<GameObject, Vector2d> poi in spawnedPOIs)
    {
        GameObject currentPOI = poi.Key;

        if (currentPOI.name == "MyArrow(Clone)")
        {
            myRoutePositions.Add(currentPOI.transform.position);
            myRouteObjects.Add(currentPOI);
        }
        else if (currentPOI.name == "OtherArrow(Clone)")
        {
            otherRoutePositions.Add(currentPOI.transform.position);
            otherRouteObjects.Add(currentPOI);
        }
        else if (currentPOI.name == "OtherShipMap(Clone)")
        {
            otherRoutePositions.Insert(0, currentPOI.transform.position);
            otherRouteObjects.Insert(0, currentPOI);
        }

    }

    myRouteMapLineRenderer.positionCount = myRoutePositions.Count;
    otherRouteMapLineRenderer.positionCount = otherRoutePositions.Count;
    myRouteUncertainMapLineRenderer.positionCount = myRoutePositions.Count;
    otherRouteUncertainMapLineRenderer.positionCount = otherRoutePositions.Count;

    myRouteMapLineRenderer.SetPositions(myRoutePositions.ToArray());
    otherRouteMapLineRenderer.SetPositions(otherRoutePositions.ToArray());

    List<Vector3> myRouteUncertainPositions = OffsetPositions(myRoutePositions, -0.001f);
    List<Vector3> otherRouteUncertainPositions = OffsetPositions(otherRoutePositions, -0.001f);

    myRouteUncertainMapLineRenderer.SetPositions(myRouteUncertainPositions.ToArray());
    otherRouteUncertainMapLineRenderer.SetPositions(otherRouteUncertainPositions.ToArray());

    OrientPointsTowardsNext(myRouteObjects);
    OrientPointsTowardsNext(otherRouteObjects);
```

FIGURE 4.73: **Main update routine for route and uncertainty overlays.** The `UpdateLineRenderers()` function assembles position lists for both the user and opposing vessel routes, updates LineRenderer components to draw the route lines, generates vertically offset positions for uncertainty overlays, and ensures all waypoints are visually connected and oriented.

- **Orienting Arrows:** Finally, it invokes `OrientPointsTowardsNext()` for both user and opposing vessel waypoints. This ensures each arrow icon along the route is correctly rotated to indicate the direction of travel.

The entire process runs dynamically, ensuring that any route update, new prediction, or movement is instantly reflected on the AR map.

OffsetPositions() and OrientPointsTowardsNext() Functions. Figure 4.74 provides the implementation details of two essential utility functions:

- **OffsetPositions():** This method receives a list of 3D positions (route waypoints) and a vertical offset value. It generates a new list where each point is lifted along the Y-axis by the given offset. This separation is crucial for rendering uncertainty overlays above the main route, so they remain visually distinct and do not obscure each other.
- **OrientPointsTowardsNext():** For each arrow (waypoint) along the route, this

```
private List<Vector3> OffsetPositions(List<Vector3> originalPositions, float yOffset)
{
    List<Vector3> offsetPositions = new List<Vector3>();
    foreach (var position in originalPositions)
    {
        offsetPositions.Add(new Vector3(position.x, position.y + yOffset, position.z));
    }
    return offsetPositions;
}

private void OrientPointsTowardsNext(List<GameObject> pointsList)
{
    // Make each point look at the next point in the list
    for (int i = 0; i < pointsList.Count - 1; i++)
    {
        Transform currentPoint = pointsList[i].transform;
        Transform nextPoint = pointsList[i + 1].transform;

        Vector3 direction = nextPoint.localPosition - currentPoint.localPosition;
        if (direction != Vector3.zero)
        {
            currentPoint.localRotation = Quaternion.LookRotation(direction, Vector3.up);
        }
    }

    // Handle the last object separately
    if (pointsList.Count > 1)
    {
        Transform lastPoint = pointsList[pointsList.Count - 1].transform;
        Transform secondLastPoint = pointsList[pointsList.Count - 2].transform;

        // Copy the rotation of the second last point to the last point
        lastPoint.localRotation = secondLastPoint.localRotation;
    }
}
```

FIGURE 4.74: **Key methods for overlay positioning and orientation.** `OffsetPositions()` applies a vertical offset to route waypoints to separate the uncertainty region from the route line, while `OrientPointsTowardsNext()` iteratively orients each arrow icon along the route to point towards the next waypoint, ensuring the heading is always visually correct.

function calculates the direction to the next point and applies the appropriate rotation. The final arrow's orientation is copied from the previous arrow, preserving visual continuity at the end of the route. This logic ensures that all arrow icons immediately and intuitively communicate the route's intended heading.

Together, these functions provide the backbone for real-time, visually intuitive map overlays that allow mariners to interpret route direction, spatial uncertainty, and navigation risk at a glance.

4.11 Conclusion

The implementation of the head-worn augmented reality (AR) navigation system detailed in this chapter demonstrates a robust, ergonomic, and highly adaptive solution for real-time maritime decision-making. Leveraging Unity, MRTK3, and Mapbox SDK, the system fuses live GPS, AIS, and predictive data to deliver actionable navigational guidance directly within the user's field of view. Key innovations include context-aware AR windows, dynamic Points of Interest (POIs), advanced route and uncertainty visualizations, and intuitive user interfaces optimized for hands-free operation.

A multi-stage calibration workflow ensures precise spatial alignment of digital content with the bridge environment, while ergonomic placement of interactive panels

and filtering controls empowers users to tailor the AR experience to their operational needs. The modular architecture enables real-time data integration from multiple sources, supporting reliable updates and synchronized overlays even in complex, high-pressure scenarios.

Through rigorous field testing and iterative refinement, the system has proven its capability to enhance situational awareness, reduce cognitive load, and support safer, more informed decision-making at sea. The technical solutions and design principles presented here lay a strong foundation for future extensions, including broader vessel integration, multi-user collaboration, and advanced predictive analytics, paving the way for next-generation AR-assisted maritime operations.

Chapter 5

Evaluation

5.1 Introduction

The evaluation of the AR navigation system was conducted as part of the CREX-DATA project, with the primary objective of assessing the prototype's usability and effectiveness in real-world maritime operations. These trials played a key role in determining the system's impact on navigational decision-making, situational awareness, and operational safety for ship crews. Testing took place on two distinct vessels—a small training ship and a tugboat—allowing the system to be evaluated across different ship types and operational contexts. The insights gathered during these field trials were instrumental in refining the AR system's design and validating its utility for real-world maritime navigation.

FIGURE 5.1: **Field trial in progress.** A user wearing the HoloLens 2 operates the augmented reality navigation system from the bridge of the test vessel.

5.2 Harbor Trials: Sites and Procedures

The evaluation trials for the AR navigation system were conducted at two key maritime locations: Kissamos Port and Souda Bay in Crete. These settings provided a diverse operational context for testing the prototype aboard different vessel types under realistic conditions.

The trials involved participants from a range of backgrounds—including professional mariners, vessel crew, and research staff—who interacted with the system on board both a small passenger vessel and a tugboat. Before testing, all participants were introduced to the system's capabilities and given instruction on operating the HoloLens 2 device and AR interface.

During the trials, both vessels remained stationary at their berths within Kissamos Port and Souda Bay. In this controlled environment, participants interacted with the AR navigation system to explore its core functionalities, including route visualization, Points of Interest (POI) management, danger area identification, and real-time situational monitoring. This setup allowed for a thorough evaluation of the system's usability and effectiveness without the complexities introduced by vessel movement.

The trials were carefully documented through observation, video recordings, and user feedback. This process allowed the research team to assess usability, identify potential improvements, and collect suggestions for further development. Overall, conducting the trials in Kissamos and Souda Bay ensured the system was evaluated in real-world maritime conditions and provided valuable insight into its performance and user acceptance.

5.2.1 Cruise Ship: First Field Trial

re introduced to the AR navigation system and received hands-on instruction in operating the HoloLens 2 headset. The evaluation focused on core system functionalities such as real-time route visualization, Points of Interest (POI) management, and dynamic map interaction within the vessel bridge environment.

FIGURE 5.2: **The 44-meter course** ship at Kissamos Port, used during initial field testing of the AR navigation system.

Participants engaged with the system as they would during standard navigational tasks, exploring its capabilities in situ. Throughout the trial, particular attention was paid to real-world operational factors—such as glare, variable lighting conditions, and the spatial limitations of the vessel bridge—to assess how these influenced the usability and effectiveness of the AR system.

Observations and First Feedback

A major challenge highlighted during the initial field trial was the complexity and inefficiency of the AR window calibration process (see Figure 5.3). At that time, users were required to open a dedicated hand menu in order to create an AR window, which then needed to be manually dragged to its intended location within the bridge. Once positioned, users had to interact with a floating control panel—placed above each window—containing separate buttons for adjusting the window's width and height independently, as well as for rotating or tilting the panel. Each dimension (X and Y axis) had to be fine-tuned separately, making the overall process both confusing and time-consuming.

This approach proved especially problematic given the irregular and non-square geometry typical of ship bridge windows. The manual controls lacked precision, making it difficult for users to center the window or to match the virtual AR Window to the actual contours of the physical window frame. As a result, the calibration outcome was often less than ideal—virtual overlays might not align properly with the view outside, reducing the effectiveness of navigational cues.

Beyond the AR window setup, participants—especially the ship's captain—emphasized the need for a greater variety of Points of Interest (POIs) and for more detailed information associated with each POI. At that stage, the system provided only two general-purpose POIs, limiting its flexibility. Users also requested the ability to selectively disable segments of the route and individual POIs within the AR interface, to reduce clutter and focus on the most critical information during operations.

Further feedback pointed out significant challenges with glare from sunlight and the inherent depth perception limitations of HoloLens 2. These factors made it difficult to judge the true position of POIs and accurately align virtual overlays with real-world objects, signaling a need for future ergonomic and visual refinements.

FIGURE 5.3: **Early AR window placement and menu interface** during initial trials. User feedback prompted significant redesign for improved usability.

Additional images captured during the trial further document the vessel setup, system deployment, and hands-on user evaluation (Figure 5.4).

FIGURE 5.4: **Field trial team aboard the 44-meter course** ship at Kissamos Port. System evaluation and feedback collection took place in real bridge conditions.

5.2.2 Tug Boat: Second Field Trial

The second round of field testing was conducted aboard a tug boat at Souda Bay, deploying a significantly enhanced version of the AR navigation system (see Figure 5.5). Drawing on insights gained from the initial trial, this iteration featured a redesigned AR window placement workflow, an expanded catalogue of Points of Interest (POIs), and newly integrated hazard area overlays—visible both on the AR map and within the user's field of view.

FIGURE 5.5: **The tug boat** at Souda Bay used for the second round of AR navigation system trials.

Observations and Feedback

A key improvement in this trial was the new AR window placement mechanism, which was met with much greater approval from users—especially the captain. This

updated approach allowed for more intuitive and precise alignment of virtual windows, accommodating the irregular shapes found on ship bridges and greatly reducing the time and effort required for calibration. The inclusion of a wider variety of POIs—now covering nearly all relevant landmarks and navigational hazards—enabled richer situational awareness and operational flexibility. Additionally, hazard areas were clearly visualized both on the map and directly in the AR environment, giving the bridge team immediate warning about restricted or dangerous zones.

Figure 5.6 shows the in-situ view as seen by users during the trial, with all major AR interface elements overlaid in real time. This illustrates how map data, POIs, and navigational guidance were visually integrated into the operational bridge environment.

FIGURE 5.6: **In-situ user perspective during the second field trial.** The AR system overlays the navigational map, route information, and Points of Interest (POIs) directly onto the bridge windows, providing real-time decision support and situational awareness to the crew.

Despite these major improvements, some challenges persisted. The tug boat's bridge was surrounded by windows from floor to ceiling, creating an environment with extremely strong ambient lighting. As a result, the HoloLens 2 display was often washed out by sunlight, which made it difficult for users to see AR overlays and digital content with the necessary clarity. This limitation—common on bright, modern ship bridges—remains an ongoing area for further system refinement, particularly with respect to display technology and adaptive UI design.

Overall, the second trial demonstrated substantial progress in usability, flexibility, and operational value, while also revealing the remaining practical hurdles that must be overcome for AR navigation systems to achieve seamless adoption in challenging maritime settings.

5.2.3 Challenges and Limitations

While the AR navigation system received positive feedback for its innovative features and operational value, several key challenges were identified during both rounds of field trials. One of the most prominent issues was the additional cognitive load imposed on users. Captains and bridge crew had to continuously shift their attention between the AR overlays and the dynamic real-world environment outside the windows. This split focus increased the risk of missing important cues in either domain, particularly in high-traffic or rapidly changing maritime settings.

Hardware-related limitations were also significant. The HoloLens 2, while powerful, struggled in the demanding lighting conditions found on ship bridges, especially those surrounded by windows. Strong sunlight often reduced display visibility, making it difficult to clearly discern digital overlays—an issue that was particularly acute aboard the tug boat, where ambient light levels were extremely high. Furthermore, prolonged use of the headset led to discomfort for some users, especially during extended periods of navigation.

Connectivity and infrastructure requirements also emerged as a potential constraint. The AR system depends on a stable Wi-Fi connection to receive real-time AIS, GPS, and hazard data from backend servers. In port environments this was manageable, but in more remote or open-sea scenarios, connectivity may be unreliable or unavailable, potentially limiting the usefulness of live guidance and dynamic updates.

Addressing these issues—through improved hardware design, adaptive display technologies, ergonomic calibration tools, and robust offline functionality—will be crucial for the AR navigation system to reach its full potential in real-world maritime operations. Future iterations should continue to prioritize user comfort, reliability in harsh conditions, and seamless integration into the complex workflow of ship bridge teams.

5.2.4 Summary and Outlook

The field trials of the AR navigation system, conducted aboard both the 44-meter course ship and the tug boat, provided invaluable insights into the system's practical strengths and areas for improvement. Real-world use by maritime professionals revealed how the platform's interactive route visualization, dynamic POI management, and integrated hazard overlays can directly support navigational awareness and bridge team decision-making.

User feedback underscored the importance of intuitive calibration, ergonomic UI design, and flexible information filtering for successful AR adoption on ship bridges. The shift to more user-friendly AR window placement and the expansion of map features in the second trial represented significant progress, demonstrating the system's rapid evolution and responsiveness to operator needs.

At the same time, persistent challenges—ranging from hardware limitations and display visibility in bright environments to cognitive load and network reliability—highlighted the need for further technical and usability enhancements. These lessons will guide ongoing refinement, with a focus on optimizing the AR experience for the demanding conditions of maritime operations.

Despite these challenges, participants from both field trials expressed strong support for the system's continued development, noting its considerable potential to

enhance operational safety and efficiency—not only for routine navigation, but also for emergency response and risk management in complex maritime environments.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The development and evaluation of a head-worn augmented reality (AR) system for real-time decision-making in maritime navigation mark a significant step forward in the integration of advanced digital tools with traditional seafaring practices. Throughout this thesis, we presented the conception, design, implementation, and real-world testing of an AR prototype that dynamically overlays critical navigational, environmental, and predictive data directly into the captain's field of view. By harnessing state-of-the-art technologies—including GPS and AIS data integration, robust back-end communication via WebSocket and Kafka, and intuitive hands-free interaction through gaze and gesture—the system addresses longstanding challenges of situational awareness and cognitive load on the bridge.

A cornerstone of the solution is its modular and user-centered design, enabling flexible calibration and ergonomic adaptation to the wide range of vessel bridge geometries found in the maritime sector. The novel AR window placement and calibration workflows, refined through iterative field testing, allow digital overlays to be precisely aligned with real-world perspectives, while the inclusion of dynamic Points of Interest (POIs), route visualizations, and uncertainty cones ensures that mariners are equipped with timely, actionable information for collision avoidance and safe passage planning. The seamless integration with the Mapbox SDK enables rich spatial context and customizable mapping, further enhancing operational awareness.

Extensive trials aboard a 44-meter cruise ship and a tug boat provided invaluable feedback from expert maritime users, shaping iterative improvements to both interface and core functionality. The trials confirmed the system's ability to improve decision-making efficiency, increase situational awareness, and provide clear guidance under realistic maritime conditions. Critically, these real-world evaluations also surfaced persistent challenges: the difficulty of AR display visibility in strong ambient light, depth perception issues with current headset hardware, and the importance of streamlined calibration processes for rapid deployment.

Despite these challenges, user feedback consistently indicated strong potential for the AR system to transform bridge operations, particularly in scenarios requiring fast, well-informed decisions. The modular software and robust data integration framework established by this project lay the groundwork for future enhancements.

In summary, this thesis demonstrates the feasibility and operational value of deploying AR navigation systems in demanding maritime contexts. It advances the state-of-the-art in maritime AR by providing a practical, tested solution that directly

supports captains and bridge teams in their daily operations. The research outcomes and field trial findings not only validate the effectiveness of the current system but also illuminate a clear path for further innovation, with the goal of achieving safer, smarter, and more efficient maritime navigation in an increasingly complex and data-rich world.

6.2 Future Work

While the developed AR navigation system demonstrates significant potential and operational value, several opportunities for further improvement and research have been identified. The following directions are recommended to address current limitations, optimize user experience, and further align the system with the demands of modern maritime operations:

System Optimization and Integration

One of the key next steps involves optimizing the system's performance for deployment in real-world maritime settings. This includes streamlining data processing pipelines, reducing computational load on the AR device, and minimizing latency in the visualization of real-time data streams. Enhanced code efficiency and robust error handling will ensure smoother operation and greater reliability during extended voyages and complex scenarios.

Direct Integration with Vessel Navigation Instruments

Currently, the system relies on a mobile device to provide real-time GPS updates. To improve accuracy and reduce architectural complexity, future iterations should pursue direct integration with the vessel's onboard navigation instruments (e.g., ECDIS, AIS transponders, shipborne GPS units). Such integration would yield more precise, higher-frequency positional data and eliminate the dependency on external devices, resulting in faster and more robust performance. Developing standardized interfaces and communication protocols for a variety of maritime hardware is a critical area for future technical research.

Persistent Calibration and Session Management

To improve usability and reduce setup time, the system should implement persistent storage for calibration settings, including AR window placements and user interface preferences. This would allow the bridge crew to save their customized environment layouts for reuse in subsequent sessions, enabling near-instantaneous deployment and consistent user experience across shifts and operations.

Depth-Aware Map and Bathymetric Overlays

A valuable extension to the current map functionality is the integration of depth (bathymetric) information. Future development should enable the AR map to display real-time or chart-based depth data, including contour lines, shallow zones, and underwater hazards. Overlaying this information directly within the AR scene and on the interactive map would provide bridge teams with immediate, intuitive awareness of water depth beneath the vessel and along planned routes, reducing the risk of grounding and supporting safer passage through complex or unfamiliar areas.

Hardware Adaptation and Environmental Resilience

Field trials have highlighted the limitations of current AR headsets in environments with extreme ambient lighting, glare, and variable weather conditions. Future research should focus on adapting the software for compatibility with next-generation

hardware—potentially including devices with improved display brightness, anti-glare technology, and environmental sealing (e.g., waterproofing). In addition, integrating adaptive UI elements that adjust contrast, color, or transparency in response to changing lighting conditions could significantly enhance usability and safety.

Expanded Data Fusion and Predictive Analytics

Integrating additional data sources such as radar, LIDAR, weather forecasting, and engine telemetry can further enrich the AR overlays and provide comprehensive situational awareness. Future development may explore advanced data fusion techniques and predictive analytics, such as machine learning algorithms for risk assessment, traffic forecasting, or anomaly detection, delivering even more proactive support to the bridge team.

Enhanced Interaction and Collaboration

Exploring new methods for user interaction—such as voice control, haptic feedback, or remote multi-user collaboration—may further improve the system's ergonomics and adaptability. Multi-crew coordination features, allowing several users to share annotations or jointly manage route planning in real time, represent a promising area for development in large-vessel or fleet operations.

User Training and Documentation

To facilitate broader adoption, future work should include the development of comprehensive user training modules, context-aware help systems, and in-situ tutorials integrated directly within the AR interface. These features will reduce onboarding time for new users and ensure consistent, safe use of the technology in varied operational contexts.

Summary

Continued development in these areas will strengthen the system's ability to meet the evolving needs of the maritime industry. By advancing integration with shipboard instruments, optimizing performance, expanding data sources, and refining the user experience, the AR navigation platform can become an indispensable tool for safe, efficient, and intelligent vessel operation.

Bibliography

(2016), Wartsila Corporation (2016). "Wartsilas service ensures a reliable and pleasant cruise for passengers of M/S Artania". In: *Wartsila Corporation*. URL: https://www.wartsila.com/media/news/02-11-2016-wartsilas-service-ensures-a-reliable-and-pleasant-cruise-for-passengers-of-ms-artania.

(2018), Wärtsilä Corporation (2018). "Augmented reality creates a new dimension in marine maintenance services". In: *Wärtsilä Corporation*. URL: https://www.wartsila.com/media/news/31-07-2018-augmented-reality-creates-a-new-dimension-in-marine-maintenance-services.

Byeong-Wook Nam Kyung-Ho Lee, Jung-Min Lee (2017). "A Study on Developing Image Processing for Smart Traffic Supporting System Based on AR". In: *2nd World Congress on Civil, Structural, and Environmental Engineering (CSEE'17)*.

Christine Plumejeaud-Perreau, Silvia Marzagalli (2022). "Mapping the uncertainty of past maritime routes". In: *European Cartographic Conference – EuroCarto 2022, 19–21 September 2022, TU Wien, Vienna, Austria*. DOI: 10.1109/SMC.2017.8123156.

Frydenberg S., Nordby K. and J. O. Eikenes (2018). "Exploring designs of augmented reality systems for ship bridges in Arctic waters". In: *RINA, Human Factors 2018*.

Frydenberg, Synne et al. (2021). "Development of an Augmented Reality Concept for Icebreaker Assistance and Convoy Operations". In: *Journal of Marine Science and Engineering* 9.9. ISSN: 2077-1312. DOI: 10.3390/jmse9090996. URL: https://www.mdpi.com/2077-1312/9/9/996.

Furuno Product Solutions (2019). "Driving the Digitalization of Navigation: ENVISION". In: *Furuno Corporation*. URL: https://www.furuno.com/special/en/envision/.

Gerald Moulis, Vincent de Larminat (2015). "How Augmented Reality can be fitted to satisfy maritime domain needs – the case of VISIPROT® demonstrator". In: *VRIC '15: Proceedings of the 2015 Virtual Reality International Conference*. DOI: 10.1145/2806173.2806200.

Grabowski, Martha (2015). "Research on Wearable, Immersive Augmented Reality (WIAR) Adoption in Maritime Navigation". In: *THE JOURNAL OF NAVIGATION (2015)*, pp. 453–464. DOI: 10.1017/S0373463314000873.

Jaeyong OH Sekil PARK, Oh-Seok KWON (2016). "Advanced Navigation Aids System based on Augmented Reality". In: *International Journal of e-Navigation and Maritime Economy 5 (2016) 021 – 031*. DOI: 10.1016/j.enavi.2016.12.002.

Jung Min Lee Byeongwook Nam, Kyung Ho Lee Yuepeng Wu (2016). "Study on Image-based Ship Detection for AR Navigation". In: *2016 6th International Conference on IT Convergence and Security (ICITCS)*. DOI: 10.1109/icitcs.2016.7740373.

Kenneth Broad Anthony Leiserowitz, Jessica Weinkle and Marissa Steketee (2007). "Misinterpretations of the "Cone of Uncertainty" in Florida during the 2004 Hurricane Season". In: *2007 American Meteorological Society*, 651–668. DOI: 10.1109/SMC.2017.8123156.

Kjetil Nordby Etienne Gernez, Synne Frydenberg Jon Olav Eikenes (2020). "Augmenting OpenBridge: An Open User Interface Architecture for Augmented Reality Applications on Ship Bridges". In: *19th Conference on Computer and IT Applications in the Maritime Industries - COMPIT 2020*.

Koulieris, George et al. (June 2019). "Near-Eye Display and Tracking Technologies for Virtual and Augmented Reality". In: *Computer Graphics Forum* 38, pp. 493–519. DOI: 10.1111/cgf.13654.

Leite, Bruno G. et al. (2022). "Maritime navigational assistance by visual augmentation". In: *Journal of Navigation* 75.1, 57–75. DOI: 10.1017/S0373463321000795.

Liu, Le et al. (June 2015). "Visualizing Time-Specific Hurricane Predictions, with Uncertainty, from Storm Path Ensembles". In: *Computer Graphics Forum* 34. DOI: 10.1111/cgf.12649.

Marie-Christin Ostendorp Jan Charles Lenk, Andreas Lüdtke (2015). "Smart Glasses to support maritime pilots in harbor maneuvers". In: *6th International Conference on Applied Human Factors and Ergonomics (AHFE 2015) and the Affiliated Conferences, AHFE 2015*. DOI: 10.1016/j.promfg.2015.07.775.

Muhammad Raziq Kazura Mohammed Ismail Russtam Suhrab1, Che Wan Mohd Noor Che Wan Othman (2024). "Enhancing Maritime Education and Training (MET) through Virtual andAugmented Reality: A Voyage into Immersive Learning". In: *JOURNAL OF MARITIME RESEARCH*, pp. 339–348.

Odd Sveinung Hareide, Runar Ostnes (2017). "Maritime Usability Study by Analysing Eye Tracking Data". In: *Journal of Navigation · April 2017*. DOI: 10.1017/S0373463317000182.

Oever Morten Fjeld, Bastian Orthmann Abel van Beek Kjetil Nordby Bjørn Sætrevik1 Floris van den (2022). "A Field Study Shows that AR Can Facilitate Collaboration in Ship Navigation". In.

Padilla, Lace, Matthew Kay, and Jessica Hullman (Feb. 2021). "Uncertainty Visualization". In: pp. 1–18. ISBN: 9781118445112. DOI: 10.1002/9781118445112.stat08296.

Potter, Kristin, Paul Rosen, and Chris R Johnson (2012). "From quantification to visualization: A taxonomy of uncertainty visualization approaches". In: *Uncertainty Quantification in Scientific Computing: 10th IFIP WG 2.5 Working Conference, WoCoUQ 2011, Boulder, CO, USA, August 1-4, 2011, Revised Selected Papers*. Springer, pp. 226–249.

Riveiro, Maria et al. (2014). "Effects of visualizing uncertainty on decision-making in a target identification scenario". In: *Computers Graphics* 41, pp. 84–98. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2014.02.006. URL: https://www.sciencedirect.com/science/article/pii/S0097849314000302.

Safranoglou, Ioannis et al. (2024). "Augmented Reality for Real-time Decision-Making in Flood Emergencies". In: *2024 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*, pp. 110–116. DOI: 10.1109/ISMAR-Adjunct64951.2024.00032.

Steven C. Mallam Salman Nair, Sathiya Kumar Renganayagalu (2019). "Rethinking Maritime Education, Training, and Operations in the Digital Era: Applications for Emerging Immersive Technologies". In: *J. Mar. Sci. Eng. 2019*. DOI: 10.3390/jmse7120428.

Tadatsugi Okazaki Rei Takaseki, Ruri Shoji Kazushi Matsubara (2017). "Development of sea route display system by using augmented reality". In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. DOI: 10.1109/SMC.2017.8123156.

Teo Chee Hong, Ho Si Yong Andrew and Chua Wei Liang Kenny (2015). "Assessing the Situation Awareness of Operators Using Maritime Augmented Reality System

(MARS)". In: *Proceedings of the Human Factors and Ergonomics Society 59th Annual Meeting - 2015*, pp. 1722–1726. DOI: 0.1177/1541931215591372.