E.T.S. of Industrial Engineering,
Computer Science and Telecommunications

Aerial Twin VR: Real-time cloning and telemetry control of a physical drone (UAV) in a simulated environment using Unreal Engine 5

Degree in Computer Engineering
(International Program)

Final Degree Project

Iker Goñi Lara

Daniel Aláez Gómez

Pamplona, 15 January 2025

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Acknowledgments

In this section, I would first like to express my sincere gratitude to my supervisor, Daniel Aláez, for his constant support, technical insight, and thoughtful feedback throughout the project. His guidance has been key in helping me stay focused and pushing the overall quality of the work higher.

I also want to thank the volunteer testers who contributed during the development phase. Their observations and suggestions were extremely helpful in identifying interface issues and improving the user experience.

Finally, I would like to acknowledge the developers and maintainers of the open-source tools used in this project. The open source community continues to be one of the greatest assets in our field.

Thank you all.

Abstract

This Bachelor's Thesis presents an innovative application developed in Unreal Engine 5 for virtual reality, aimed at significantly improving the remote control of drones through the generation of a virtual clone positioned within a realistic three-dimensional environment created using Cesium. The system implements real-time communication via *WebSockets*, enabling efficient transmission of telemetry and GPS positioning with minimal latency. In addition, it features an advanced visual collision prevention system that provides the pilot with adaptive graphical alerts based on the risk of impact. The interface, specifically designed for virtual reality, offers floating screens with clear and interactive data, real-time graphs, and an intelligent notification system. This modular solution opens up future possibilities for integration, including AI-powered voice command control.

Glossary of Terms

VR (Virtual Reality): Technology that simulates computer-generated three-dimensional environments, allowing users to interact with them immersively using specialized headsets.

UAV (Unmanned Aerial Vehicle): Aerial vehicle without a human pilot on board, commonly known as a drone, used in civil, industrial, and military applications.

FPV (First Person View): A pilot mode in which the operator controls the drone as if onboard, viewing the scene in real time from a front-facing camera.

GCS (Ground Control Station): Ground-based station used to monitor, plan, and execute UAV missions, including telemetry, video, and flight parameters.

Cesium ion: Web platform that provides 3D terrain and photogrammetric city tiles via real-time streaming, compatible with graphics engines like Unreal Engine.

Tiles: Segments or sections of 3D terrain downloaded on demand by the graphics engine based on the camera's position and required level of detail.

LOD (Level of Detail): Optimization technique that adjusts the geometric complexity of a 3D object depending on its distance to the camera to maintain performance.

LIDAR (Light Detection and Ranging): Laser scanning technique used to capture highly accurate 3D terrain models from point clouds.

HMD (Head-Mounted Display): Head-worn device, such as the Meta Quest, that allows users to experience VR environments from an immersive perspective.

Pawn (in Unreal Engine): Object that represents the player or a controlled entity, in this case the VR avatar that moves through the simulated environment.

Blueprints (UE): Unreal Engine visual scripting system that allows designers to create logic and behaviors without writing code.

Widget Blueprint: Template for interactive graphical interfaces in Unreal Engine, used to create menus, info panels, and buttons in VR.

Hand-tracking: Technology that detects hand movements without physical controllers, enabling VR interaction through natural gestures.

Motion sickness: Nausea or discomfort caused by discrepancies between physical movement and visual perception, common in poorly optimized VR environments.

Vision tunneling: Technique to reduce motion sickness by dimming peripheral vision during rapid movements or sharp turns.

WebSocket: Real-time bidirectional communication protocol used to transmit drone telemetry without blocking VR rendering.

GameInstance (UE): Persistent class that runs throughout the entire game lifecycle, useful for storing global data and managing network connections.

Tick: Update event triggered every frame by the game engine; used to execute recurring logic such as telemetry reading or rendering new positions.

RPC (Remote Procedure Call): Mechanism that allows functions to be executed remotely on another device or thread, not used in this work but relevant in advanced client-server architectures.

Meta XR Plugin: Official toolkit for integrating Meta Quest devices with Unreal Engine, including support for hand-tracking and voice input.

Voice SDK (Meta): Development kit enabling voice command integration in VR applications, currently under evaluation for future project iterations.

Git LFS (Large File Storage): Git extension for managing large files, such as Unreal models and textures, without bloating the main repository.

Contents

Introduction **5**

1 Context **8**

1.1 Technological Context 8

1.2 Project Justification 10

2 Working Methodology **13**

2.1 Hardware Environment and Testing 13

2.2 Software Tools and Libraries 16

2.3 Project Planning 17

2.4 Incremental Development with Vertical Slicing 18

2.5 Version Control and Documentation on GitHub 20

3 Project Design **22**

3.1 Requirements 22

3.2 Global Architecture 24

3.3 Cesium Integration 28

4 Development **31**

4.1 Slice 0: VR Base & Cesium 31

4.2 Slice 1: Real-time Telemetry 34

4.3 Slice 2: Graphs Window 38

4.4 Slice 3: Collision Detector System 41

4.5 Slice 4: Notification System 48

4.6 Slice 5: Hand menu 52

4.7 Slice 6: Hand-tracking & Motion Sickness 56

4.8 Slice 7: 3D Physical Main Menu 59

5 Testing and Validation **65**

5.1 Methodology and Testing Environment 65

5.2 Participants and Profiles 66

5.3 Validation Criteria 66

5.4 Results Obtained 67

5.5 Identified Improvements and Fixes 68

5.6 Identified Limitations 68

6 Conclusion **71**

6.1 Differential Value of the Project 72

6.2 Future Work Directions 73

References **75**

Introduction

Humankind's fascination with flight has driven, since ancient times, the development of technologies aimed at expanding our physical and cognitive capabilities. In recent decades, drones or Unmanned Aerial Vehicles (UAVs) Figure 1 have emerged as key tools in fields such as precision agriculture, cinematography, surveillance, logistics, rescue operations and entertainment. Their increasing popularity has democratized their use, but has also revealed new technical challenges related to safety, control accuracy, flight data interpretation, and accident prevention.

Despite advances in sensors, cameras, and real-time transmission systems, traditional solutions still present significant limitations in terms of spatial perception, visual latency, operational complexity, and pilot ergonomics. In this context, virtual reality (VR) technologies and 3D simulation open up a promising path to completely redesign the remote piloting experience.

Figure 1: Flying drone [1]

This Bachelor's thesis proposes a new UAV piloting support tool based on virtual reality. An immersive system has been developed that replicates the real-time position and telemetry of a physical drone using WebSocket-based communica-

tion, placing it within a photorealistic 3D environment generated with Unreal Engine 5 and Cesium. The pilot can explore the environment from any angle, access floating visual interfaces, receive contextual alerts, and follow the flight using different camera modes, all without losing precision or smoothness.

In addition to replicating standard monitoring functions, the system explores natural interaction methods through hand-tracking, 3D physical buttons, and an open architecture designed to integrate AI-assisted voice recognition. It also includes specific measures to mitigate motion sickness, common in prolonged VR sessions, such as vision tunneling, trajectory smoothing, and progressive teleport transitions.

Regarding the working methodology, an iterative development approach was followed, based on functional deliveries (organized in *vertical slices*) with regular meetings between the author and the project supervisor. Each functional block was designed, implemented and validated independently, allowing improvements to be made, priorities to be redefined, and new features to be added based on progress and feedback received.

This document is structured as follows:

- **Chapter 1** presents the state of the art, current solution limitations, and the technical justification for the project.
- **Chapter 2** describes the resources used and the methodology followed.
- **Chapter 3** covers system design: objectives, requirements, architecture, data model, and integration with Cesium.
- **Chapter 4** addresses the implementation, divided into functional slices that reflect the progressive development of the project.
- **Chapter 5** documents laboratory tests and functional validation.
- Finally, **Chapter 6** presents the general conclusions and possible future work directions.

This document not only describes a functional system, but also highlights the technical and design decision-making process that turned an initial idea into an

immersive, efficient, and future-ready tool built to evolve alongside emerging technologies.

Chapter 1

Context

1.1 Technological Context

Drone piloting has evolved rapidly from its initial recreational and military applications to become a key professional and commercial tool in sectors such as engineering, security, logistics, and agriculture. Traditionally, these systems were operated within direct visual line of sight (VLOS) [2], relying solely on the operator's skill and radio frequency transmitters. However, the limitations of this model have led to new forms of remote piloting based on technology. BVLOS (Beyond Visual Line of Sight) flights Figure 1.1, where the operator does not see the drone directly but instead relies on sensors, telemetry, and video to guide it, have become one of the main operational paradigms in complex or large-scale contexts.

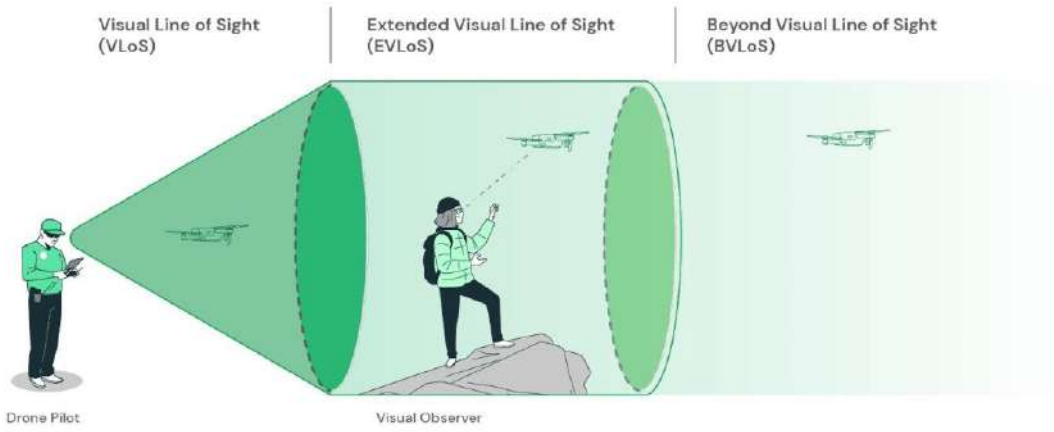

Figure 1.1: VLOS, EVLOS and BVLOS modes [3]

This shift has been accompanied by significant development in ground control systems (GCS), which allow operators to plan routes, receive telemetry, monitor the drone's status, and visualize its position on maps. Platforms such as Mission Planner, QGroundControl, and DJI Pilot are widely used examples. These tools offer graphical interfaces with multiple windows: one for the map, another for live video, and another for instrumentation, among others. However, studies such as those conducted by NASA Ames [4] have pointed out common limitations of many current GCSs: dense and overloaded interfaces, fragmented information across unrelated windows, difficulties in clearly representing spatial dimensions (altitude, obstacles) in 2D interfaces, and a steep learning curve for novice users.

To address these issues, the potential of a more immersive flight visualization is being explored, with the aim of providing a more intuitive understanding of the environment and the drone's relative position. Traditional spatial representation, based on 2D maps and symbols, is not always adequate for complex or urban environments, where the third dimension is especially important for both navigation and decision-making.

Modern digital FPV links (e.g. DJI, Walksnail, HDZero) have improved video resolution, range, and overall transmission stability compared to older analog systems [5]. These technologies can achieve low latencies and high frame rates under laboratory conditions, offering an immersive experience to the pilot. However, maintaining truly real-time video remains a challenge: any additional delay in the signal can hinder precise drone control, especially during fast maneuvers [6].

Latency is identified as a critical concern in drone teleoperation. Even a slight delay in video transmission can cause the pilot to react late to obstacles or changes, complicating safe flight [7].

In terms of interaction, recent trends point towards the development of more natural and immersive interfaces, including the use of HUDs (Head-Up Displays) that project information onto the visible environment and emerging technologies such as hand-tracking and voice control. However, these solutions are still in the early stages of adoption and their implementation in real operational environments, especially in critical applications such as rescue or industrial inspection, poses significant technical challenges in terms of reliability, precision, and com-

patibility with existing safety protocols.

The current technological context reveals a clear transition towards more integrated, immersive, and user-centered solutions. This Bachelor's thesis positions itself at that intersection, proposing a system that reimagines the pilot interface through virtual reality, natural interaction, and 3D spatial visualization, with the goal of anticipating the next generation of UAV control tools.

1.2 Project Justification

This Bachelor's thesis was conceived with the intention of exploring new technological possibilities applied to drone piloting and control, not with the immediate goal of replacing existing systems, but to provide an innovative perspective on how emerging technologies can transform this activity. The proposal originated from a previous departmental project that focused on visualization of flight data through a traditional 2D interface. Upon joining the project, and given my prior knowledge in immersive technologies, the opportunity was identified to extend the scope of that solution into a virtual reality environment, offering the pilot enhanced spatial perception, greater immersion, and more direct and intuitive interaction with flight information.

This approach is based on the real need to overcome certain technical and operational limitations currently present in conventional piloting systems. Below are some of the main critical points identified in the context of professional UAV operations.

- **Dependence on multiple operators:** in practice, UAV missions typically require at least two people: one for manual piloting and another for monitoring parameters such as telemetry, battery and signal. This separation increases human resource demands and can lead to operational miscoordination.

- **Limitations in pilot's spatial perception:** current systems offer limited and partial views of the environment, making it difficult to accurately understand the drone's position and orientation in three-dimensional spaces, especially in urban or obstacle-rich environments.

- **Visibility issues in outdoor environments:** the use of conventional screens in high-brightness conditions severely limits the readability of critical data, affecting real-time response capabilities.
- **Complex or unintuitive interfaces:** tools such as Mission Planner Figure 1.2, although very comprehensive, have dense designs that may be inaccessible to nonexpert users. The abundance of options without a clear hierarchy increases the learning curve and the risk of operational errors.
- **Communication security vulnerabilities:** the widely adopted MAVLink protocol does not include robust encryption or authentication mechanisms by default, exposing it to potential interceptation or malicious command injection attacks [8].
- **Limited collision detection and prevention:** many systems rely solely on basic sensors and the pilot's visual capacity, which is insufficient in dense or low-visibility environments [9].

Figure 1.2: Screenshot of Mission Planner GCS, with MAVLink protocol [10]

From these critical points, various possible development lines were considered. Initially, working on the security of MAVLink and WebSocket protocols was

evaluated, but the project ultimately focused on improving spatial perception and interaction through virtual reality, leveraging both my prior experience and the potential of this technology to simultaneously address multiple limitations.

The proposed solution would directly respond to the identified issues. First, it would allow the pilot to visualize the drone's environment from an immersive three-dimensional perspective, combining 3D simulation via Cesium with real-time synchronization using GPS data. This capability would significantly improve airspace awareness and reduce the likelihood of collisions.

Second, VR visualization would eliminate readability issues caused by direct sunlight on conventional screens, ensuring that all critical parameters remain visible under any lighting condition. Additionally, menus and information windows would be integrated into the 3D space as floating elements, improving ergonomics and reducing the pilot's cognitive load.

Third, simplified and intuitive graphical interfaces could be designed, prioritizing visual representation over raw numerical data. This would facilitate quick reading of information such as battery level, altitude, or speed, even for operators without advanced technical training.

Another key component would be the incorporation of a collision prevention system based on simulated sensors. This system would inform the pilot with directional visual cues about nearby obstacles, with a detection range adaptable to each mission's conditions.

Furthermore, the project could integrate emerging technologies that enhance natural interaction, such as hand-tracking for gesture-based function activation, or voice recognition through the Meta Voice SDK. The latter would reduce the need for manual input, preparing it for an operational model where a single pilot could manage both the drone and the VR interface with complete freedom of movement.

As we can see, this approach opens a promising path toward evolving current systems. It is not conceived as an immediate replacement, but rather as an innovative alternative that anticipates a new generation of more immersive, intuitive, and user-oriented remote piloting tools.

Chapter 2

Working Methodology

2.1 Hardware Environment and Testing

The development of the project required not only design and programming capabilities, but also a testing environment that was sufficiently powerful and flexible to verify each new functionality under the most realistic conditions possible. This section describes the evolution of the technical environment (computer and headsets) in which the application was built and validated, and justifies the decisions made from the perspective of performance, compatibility and user experience.

Main Hardware Selection

The project began modestly, with preliminary tests performed on a laptop equipped with an Intel i5 processor and integrated graphics. It was on this machine that the first load of a Cesium 3D scenario in 2D was carried out [11], which was enough to reveal the system's limitations: slow loading, jerky movement, and an inability to achieve a fluid experience; especially in VR.

In light of this, we decided to upgrade to a machine capable of handling high workloads without bottlenecks: a laptop with an Intel i9-14980HX processor and an NVIDIA RTX 4070 Studio GPU. Two reasons fully justified this decision:

- **Scenario load:** the system works with streaming of photogrammetric tiles, high-resolution textures, and complex geometry in real time, which places a constant demand on the GPU.

- **VR requirements:** rendering two simultaneous high-frequency views without perceptible latency requires a powerful GPU and an architecture capable of compiling, debugging, and executing without interruptions directly from the development environment.

The result was a robust and agile work environment. This configuration enabled direct headset debugging, stable frame rates even with effects such as *ambient occlusion* or *volumetric fog* enabled, and drastically reduced iteration times.

Headset Evolution: From Quest 2 to Quest Pro

For the VR headset, the first iterations were developed using Meta Quest 2. This model provided a fast onboarding curve, was lightweight, easy to connect via USB-C, and had a large, well-documented community. However, as the application grew, three practical limitations emerged:

- The **passthrough mode** to the real world is monochrome and low-resolution, forcing the user to remove the headset to perform manual tasks.
- The **head strap** poorly distributes weight, making it uncomfortable to wear during long sessions.
- **Limited memory and thermal capacity** forced quality reduction or geometry simplification in complex scenes.

For these reasons, the decision was made to migrate to a superior headset, the Meta Quest Pro Figure 2.1 [12]. This model includes substantial improvements:

- Low-latency **color passthrough**, allowing the headset to remain on while supervising physical controls.
- Sharper pancake optics and a **halo-style strap** that distributes the weight across the head and allows the headset to be lifted slightly from the face for increased comfort.
- More memory and an **upgraded XR2 processor**, avoiding bottlenecks in demanding scenes and allowing for future expansion.

Figure 2.1: Quest Pro design [13]

This headset allows quick inspection tasks without removing the device: Just lift your head slightly to look at the physical controls or redirect your gaze to a physical UAV in the room. This is especially useful in shared work environments, such as classrooms, laboratories or test zones.

Testing Strategy

All validations were performed in controlled environments and with departmental colleagues, without direct participation from external pilots. Even so, the combination of the high-performance laptop with the standalone headset offered a key advantage: it enabled replicating a complete development and testing environment anywhere, as if carrying the lab in a backpack.

This approach enabled short iteration and verification cycles. Each newly implemented functionality could be tested in under a minute under conditions similar to real-world usage: open the laptop, connect the Quest Pro, and launch the application. This agility was crucial for maintaining a steady development pace without sacrificing test realism.

2.2 Software Tools and Libraries

Unreal Engine 5 as a Continuity Strategy

When the project was conceived for virtual reality, a desktop prototype already existed in Unreal 5, with Cesium tile streaming integrated and functioning. Rewriting this foundation in Unity 6 would have meant redoing work and, more importantly, giving up the graphic advantages that were already yielding results. The *Lumen* and *Nanite* duo [14] allowed photogrammetric cities to be rendered with uniform lighting and meshes of millions of triangles without the need to precompute LODs, especially valuable since the drone camera travels long distances and any lack of detail becomes immediately evident. Unity offers comparable solutions, such as HDRP [15] and impostors, but they require more manual authoring and, in practice, do not match the seamless level of detail Nanite provides out of the box in VR.

Choosing Unreal also eliminated the learning curve, since I already had experience programming in Unreal using Blueprints. Maintaining that visual language accelerated every iteration and prevented common mistakes from paradigm shifts to C++ or C# depending on the platform.

Blender

During development, it was necessary to lighten the main menu environment and, more importantly, to restructure the scanned drone model to separate the propellers from the fuselage and animate them independently. All this work was done in Blender [16], a familiar license-free tool. The exported FBX format integrates seamlessly into the Unreal pipeline, and material repacking is performed in a single step, which reduced time and avoided reliance on commercial software.

Cesium

Reproducing the world required more than aerial textures stuck to a flat plane. With Cesium for Unreal, the engine receives 3D tiles representing real buildings and terrain, streamed in real time with automatic level-of-detail updates. This

data flow, already operational in the first prototype, proved stable even in VR and left open the possibility of injecting LIDAR point clouds [17] for critical zones, enriching local accuracy without breaking global continuity. The Unity SDK counterpart is still maturing; in 2025 it continues to require patches and specific engine versions to compile. Thus, the most reliable path for this project was to continue with the Unreal branch.

Meta XR and Voice SDK

Although Unreal includes generic OpenXR support, the available headset at the university is Meta Quest, and the Meta XR Plugin [18] offers two decisive advantages: more accurate hand-tracking, a valuable resource leveraged in this project, and integration of Voice SDK, which has begun to be integrated for future voice control without adding external libraries. Using the official toolkit ensures long-term compatibility and device-specific controller optimizations.

WebSockets

Telemetry arrives from the UAV encapsulated in WebSockets, a lightweight protocol that mitigates latency without blocking the render thread. The client-server architecture is part of the repository itself and pushes coordinates into the scene in real time. For the C++ code portion dedicated to WebSockets, Visual Studio 2022 was used, the IDE recommended by Epic, with a native debugger and static analysis to help prevent memory leaks in modules running in the same process as the VR rendering.

2.3 Project Planning

The complete work cycle lasted approximately ten months, combining an iterative delivery structure with a flexible approach that made it possible to maintain the pace of development without compromising the quality of the deliverable or long-term motivation.

Halfway through the process, a first semi-functional version was prepared that

included the Cesium map, the pawn's basic locomotion, the hand menu, and an initial approximation of the telemetry system. This version was used as a demonstration for the South Summit event in Madrid [19], presented at the stand of the company FuVeX. Beyond verifying technical stability, this deployment served as a usability benchmark in a real environment, receiving spontaneous reactions that proved very useful in guiding subsequent improvements. That event marked the first major milestone of the project. The second was the consolidation of the final version that accompanies this report.

Throughout the development process, regular contact was maintained with the TFG advisor through in-person meetings approximately once a month, adapted to the academic calendar. In each session, a functional demonstration was carried out, technical or conceptual problems were identified, and priorities for the next phase were redefined. This dynamic allowed for an agile way of working, adjusting objectives as progress was made and documenting everything using *issues* labeled by priority on the GitHub Kanban board.

In several of these meetings, departmental colleagues also participated, acting as test users and providing observations from diverse perspectives. Their contributions were key to detecting blind spots in the interface and validating whether the design decisions were effective in practice.

Regarding the initial scope, no relevant functionality was discarded, although it was decided to postpone the implementation of voice control. Despite investigating the Meta Voice SDK and beginning its integration, the complexity of the system and the need for an additional iteration to achieve real-time reliability made it unfeasible within the TFG's timeframe.

2.4 Incremental Development with Vertical Slicing

The development approach followed the technique known as *vertical slicing* Figure 2.2, which consists of building complete functional deliveries that cut through all layers of the application: user input, processing logic and visual presentation. Each slice delivers tangible value to the user and is tested as an autonomous unit before moving on to the next.

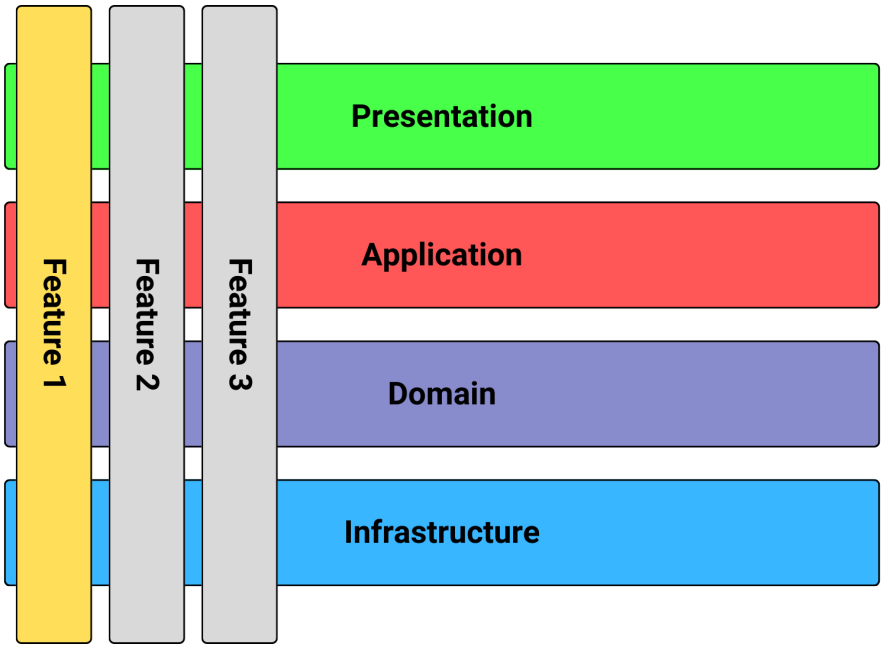

Figure 2.2: Vertical Slice Architecture [20]

This methodology helped maintain system cohesion, avoid premature cross-dependencies, and focus design efforts on a specific functionality per iteration. The developed slices were as follows:

- **Locomotion + Cesium:** VR Pawn with teleportation and dynamic 3D map loading
- **Real-time telemetry:** Repositionable HUD with real-time telemetry
- **Data charts:** Single or triptych window with telemetry data graphs
- **Anti-collision system:** 25 rays and spheres warning of obstacle direction and severity
- **Notification system:** Brief messages with icon and text (battery, collision, etc.) with no persistent history
- **Hand menu:** Panel anchored to the palm with UMG buttons to move the pawn and show/hide windows
- **Hand-tracking:** Full menu and object control without controllers
- **3D main menu:** Initial menu with 3D tactile controls like buttons and sliders

Each slice was considered complete only when it worked properly in the Meta Quest headset, without appreciable loss of performance or user comfort.

2.5 Version Control and Documentation on GitHub

All development was managed through the repository UAVisualizer_VR on GitHub (currently public). From the beginning, the repository was conceived as a "living memory" of the project, although its structure and level of sophistication evolved as my experience with the platform grew.

A simple but functional branching scheme was adopted: the main branch (`main`) always maintains a stable and executable version, while each new feature was developed in a short branch of the type `feature/...`. Once the implementation was complete, changes were merged into `main` using a `merge commit` accompanied by an explanatory summary.

The commit system served as an informal chronological record. It documented tasks completed, problems identified, solutions adopted, and technical decisions that would otherwise have been scattered in personal notes. Although basic, this practice proved valuable for tracking when a feature was introduced, why a certain approach was chosen, or what dependencies were added at a specific time.

For the most important milestones, such as the South Summit demonstration version [19] or the final TFG submission, lightweight tags (`tags`) were created accompanied by changelogs in Markdown format. This allows any reviewer to clone the exact functional state of those versions, facilitating their review or future reproduction.

Although advanced collaborative development techniques such as *pull requests* or continuous integration were not applied, the habit of uploading frequently and keeping the main branch free of unstable code prevented major errors in the final stages. Unreal binary files (models, materials, textures) were managed using Git LFS, which helped keep the repository size contained, which at project closure was around 200 MB.

Along with the code, the repository includes a README with instructions to open the project in Unreal Engine 5.5, a quick usage guide, and minimal techni-

cal documentation to guide new collaborators. Altogether, the repository fulfills its academic role: preserving the complete development history, allowing the reproduction of previous versions, and bringing together all necessary resources to continue evolving the tool beyond the TFG.

Chapter 3

Project Design

3.1 Requirements

Designing an immersive system for remote piloting assistance requires precisely identifying the functions the system must perform, as well as the conditions necessary for these functions to be reliable, comfortable, and sustainable in real usage environments. This section establishes a dual categorization: functional requirements, which describe what the system must do, and non-functional requirements, which determine how it must do it to ensure a high-quality experience.

Both tables are supported by a set of general objectives that define the mission of the project and serve as a reference to generate traceability between goals and functionalities. These objectives were developed based on the analysis of needs detected in current control systems and the opportunities that virtual reality technology offers to overcome them.

- OBJ-01: Provide a remote pilot support tool capable of displaying flight data in real time within an immersive 3D environment.
- OBJ-02: Improve operator spatial awareness and safety through photogrammetric views and alert mechanisms for potential collisions.
- OBJ-03: Design visual interfaces that are clear, intuitive, and aesthetically coherent with the context of use.
- OBJ-04: Increase interaction freedom through technologies such as hand-tracking or voice recognition.

- OBJ-06: Ensure usability during prolonged sessions by minimizing motion sickness associated with virtual reality.
- OBJ-07: Prepare the system architecture to facilitate maintenance and the incorporation of new features in the future.

These objectives reflect a balance between technical, functional, and user experience aspects that define the scope of the project. The goal is not just to offer a new way of visualizing telemetry, but to propose a more natural, robust, and forward-compatible interaction environment.

The functional requirements (Table 3.1) cover everything from basic visualization and environment navigation to interactive components such as menus, floating windows, and alert mechanisms. They are prioritized using a simple hierarchy: `MUST` (essential), `SHOULD` (recommended), and `COULD` (optional), to guide the implementation order based on their impact on the primary purpose of the system.

Table 3.1: Functional requirements of the system

ID	Brief Description	Priority	Obj.
RF-01	Load and display the Cesium 3D map with continuous streaming	MUST	01
RF-02	Move the pawn (free teleportation, jump to "home" or the drone)	MUST	01
RF-03	Hand menu to exit, center view, and toggle windows	MUST	03
RF-04	Repositionable floating telemetry window in real time	MUST	01
RF-05	Historical charts of selected variables	SHOULD	01
RF-06	Directional visual alert for imminent collisions	MUST	02
RF-07	Pop-up notifications based on monitoring rules (e.g., battery)	MUST	02
RF-08	Drone trajectory smoothing through interpolation	SHOULD	02
RF-09	Immersive main menu adapted to VR	SHOULD	03
RF-10	Full control via hand-tracking	COULD	04
RF-11	Anti-motion sickness measures (vignette, vision tunnel, speed limit)	SHOULD	06

The non-functional requirements (Table 3.2) establish conditions related to performance, ergonomics, code maintainability, and the system's compatibility with different devices. These aspects do not define direct functionalities, but are

essential for ensuring that the user experience is viable in real-world scenarios and that the system can evolve in the medium term without losing its integrity.

Particular attention should be paid to visual performance parameters such as frame rate and latency, as they have a direct impact on pilot comfort and prevention of symptoms associated with visual fatigue or motion sickness. Similarly, criteria related to code structure, documentation, and the use of tools such as Git LFS reflect a commitment to good development practices and to the potential reuse of the project by third parties.

Table 3.2: Non-functional requirements of the system

ID	**Brief Description**	**Priority**	**Obj.**
RNF-01	Stable FPS in headset and low latency	MUST	06
RNF-02	Sessions of $\geq$ 30 minutes without VR sickness symptoms	MUST	06
RNF-03	Modular and commented Blueprint code to facilitate maintenance	SHOULD	07
RNF-04	Folder structure and naming conventions aligned with Unreal guidelines	SHOULD	07
RNF-05	Use of Git LFS for assets and a README with deployment instructions	SHOULD	07
RNF-06	Compatibility with Quest 2 and Quest Pro via Meta XR Link	SHOULD	04
RNF-07	Coherent visual design aimed at semi-professional users	COULD	03

This set of requirements defines the quality standard expected of the system. It will serve as a reference for evaluating the results presented in later chapters and as a foundation for determining whether the implemented functionalities adequately address the needs that gave rise to the project.

3.2 Global Architecture

The design of the system's global architecture is based on two fundamental principles: the separation of responsibilities through well-defined logical layers, and the efficient transmission of real-time data via a WebSocket channel. This combination keeps the system modular, extensible, and with minimal latency, ideal for

immersive environments like virtual reality.

Layered Architecture

At the core of the system lies the `GameInstance` class, which acts as the central hub for data. It has a dual function: to keep the most recent telemetry state in memory and to manage the WebSocket connection with the Python server that redistributes drone data. This centralization turns GameInstance into the single source of truth, preventing inconsistent states among subsystems and simplifying the update model.

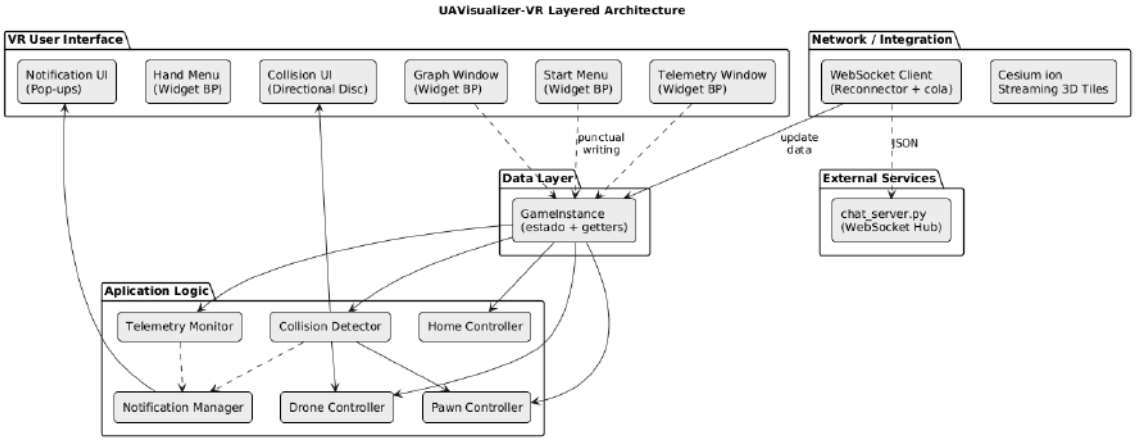

Figure 3.1: Layered Architecture Diagram

On this foundation, the application logic modules are built Figure 3.1, each specialized in a specific task related to flight:

- **Pawn Controller:** interprets HMD movements, controllers, or gestures to move the VR avatar and allows teleportation to the drone or home position.
- **Drone Controller:** interpolates between positions to smooth model movement and synchronizes animations such as rotor spinning.
- **Home Controller:** maintains the takeoff point position based on received information.
- **Collision Detector:** casts rays around the drone and detects obstacle proximity, generating alerts if thresholds are exceeded.

- **Telemetry Monitor:** continuously checks telemetry values to detect anomalies and trigger warning events.
- **Notification Manager:** gathers events from the above systems and turns them into visual alerts or warning icons.

The upper layer corresponds to the VR presentation, where the results of all this logic are reflected in real time:

- A **start menu** allows for configurations such as date and time and initial connection to the server.
- A **hand menu** anchored to the palm enables toggling views or controlling information windows.
- **Floating telemetry and chart windows** update their content with each new incoming message.
- **Collision warnings** (such as a directional disk on the headset or environmental spheres) and *pop-up notifications* alert the pilot of relevant events.

All these elements are implemented as *widget blueprints*, encapsulating both the visual components and light interaction logic. The core intelligence resides in the intermediate modules, which facilitates reuse in other environments.

At the level of external services, the system relies on just two components:

- **Cesium ion**, responsible for streaming the 3D tiles that make up the real-world environment.
- **chat_server.py**, a WebSocket server written in Python that transforms drone data into JSON format and distributes it to connected clients.

WebSocket Client-Server Flow

Data flow is unidirectional and low-latency Figure 3.2. The drone sends its telemetry to a ground laptop (which may also run the application), where the Python server processes it into a compact JSON format and publishes it on a WebSocket port. When the user presses `Start` in the start menu, GameInstance opens the

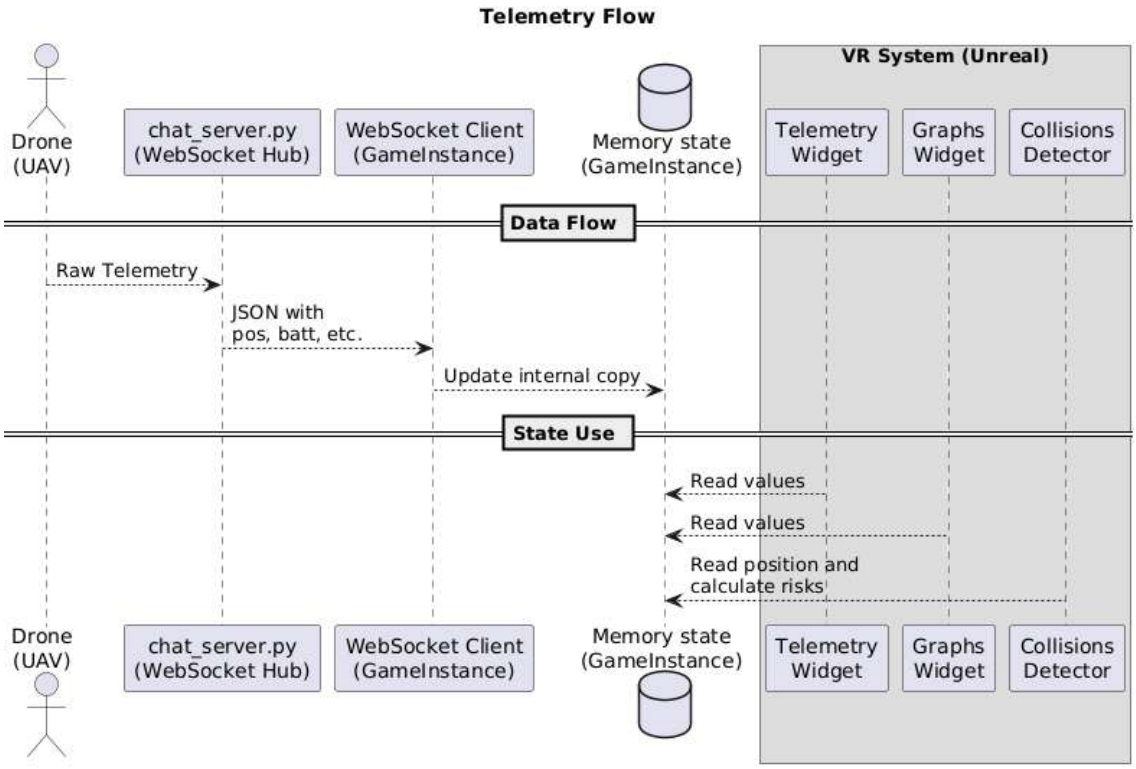

Figure 3.2: Telemetry Data Flow Diagram

connection, listens to the channel, and enqueues incoming messages from a background thread. The main thread consumes this queue without blocking the rendering process, ensuring a smooth experience.

Each subsystem reacts to the new data: the Drone Controller interpolates the position, the Collision Detector updates its calculations, and the Telemetry Monitor evaluates alert conditions. If an anomaly is detected, the Notification Manager triggers the appropriate visual response. In this way, the pilot receives near-instantaneous feedback from any change in the drone's data.

This design supports scalability: the protocol can be expanded to incorporate new message types, such as control commands or authentication mechanisms, without altering the system core.

Centralizing telemetry in GameInstance simplifies synchronization, avoids discrepancies between subsystems, and enables playback testing with recorded data without requiring a physical drone connection. Additionally, maintaining a clear separation between logic and presentation guarantees code portability to other devices, such as traditional screens or different headsets. Finally, limiting external dependencies to Cesium and the WebSocket server streamlines installation and

ensures long-term continuity through future migrations or expansions.

3.3 Cesium Integration

The main scene in the headset is powered by *Cesium for Unreal*, a plugin that streams continuous flows of photogrammetric tiles and terrain meshes into the engine in real time. The decision to use Cesium was not only based on its visual quality (other providers like Google or Mapbox also offer photogrammetry), but rather on the plugin's maturity, demonstrated stability, and direct integration with Unreal Engine 5.

At the start of development, Cesium already had over two years of stable releases and included VR-specific examples. Although its official demo has some compatibility issues with recent engine versions, the basic guide provides clear instructions for instantiating a pawn, enabling navigation, and applying teleportation in just a few steps [21]. This combination of stability and documentation was key in reducing the project's ramp-up time and focusing efforts on functional development.

Another advantage was Cesium ion's free usage model. The quota included in its academic plan easily covers the traffic generated by test sessions, without requiring special configurations. In the event of public distribution of the tool, an educational agreement could be negotiated, but in the current state, access is managed via a personal token securely included in the project.

Dynamic Tile Management

Cesium delegates all level-of-detail (LOD) management and tile selection to its runtime. On a machine equipped with an RTX 4070 Studio, the application can sustain over 90 fps while rendering scenes at the highest available quality. However, a *Screen Space Error* parameter and slider are exposed to degrade mesh resolution if the graphical load is shifted to less powerful devices, such as standalone headsets.

To optimize performance in open scenes, a strategy based on *Exponential Height Fog* is used to limit draw distance based on altitude. This reduces GPU load and

bandwidth consumption by avoiding the download of distant tiles, without sacrificing visual naturalness from the drone's point of view.

The application does not download or preload local maps. All information is managed in real time, which imposes certain requirements on network quality but keeps the executable lightweight and ensures that the terrain visualized is always up to date without recompilation.

High-Precision LIDAR Zones

In strategic areas of the project, such as the testing field, LIDAR scans have been incorporated [17] to improve terrain fidelity. Cesium allows custom models to be uploaded through ion, where they are processed as proprietary tiles accessible only with the author's token.

This mechanism enriches the scene locally without needing code modifications. As soon as the user approaches the corresponding coordinates, the viewer automatically receives the private tiles and replaces the generic photogrammetry with the high-resolution mesh. The system integrates both sources seamlessly, blending precision with visual continuity.

Lighting Consistent with Date and Time

The lighting system also adjusts dynamically via the *SunSky* actor, which calculates the solar position based on the day and time selected by the user. This data is set from the start menu, and Cesium automatically adapts the solar elevation and light tone.

This mechanism prevents texture overexposure, provides shadows consistent with the real-world location, and improves depth perception in low-altitude scenes. Although its effect is subtle, it directly impacts immersion and the overall credibility of the environment.

Future Flexibility

Cesium's integration provides immediate realism, allows quality scaling based on available hardware, and enables localized scene enhancement through custom

data. All of this fits the project's modular design: if in the future a more suitable photogrammetric tile provider is identified, it would be enough to reconfigure the plugin without altering the rest of the system's logic or visual architecture.

Chapter 4

Development

4.1 Slice 0: VR Base & Cesium

Adaptation from Desktop

As previously mentioned, this project originated from an earlier desktop application whose core functionality was to display 3D maps using the Cesium library. In that initial version, movement through the map was achieved by switching between various strategically positioned cameras: one at the drone's location, another at the "home" point, and a third free camera that allowed the user to navigate the scene. The first goal when migrating the application to virtual reality (VR) was to transfer this same logic in order to provide a coherent experience with respect to the original design.

Initial VR Implementations

In the early development phase, I created a basic VR pawn within Unreal Engine, taking advantage of the documentation provided by Cesium. A particularly useful tutorial demonstrated locomotion by teleportation via raycasting. This technique is especially effective in VR environments as it significantly mitigates motion sickness, allowing the user to move between locations without apparent physical displacement.

Since the intention was to replicate the behavior of the original desktop version, the “object possession” technique in Unreal Engine was initially explored.

However, unexpected complications quickly arose: when "possessing" different objects, Unreal transfers not only the camera but all control logic associated with the object, which meant that the state and controls had to be fully replicated in each new object. This unnecessary complexity made it clear that a much simpler and more scalable approach was needed.

Paradigm Shift: Moving a Single Pawn

The final solution was to keep a single pawn active throughout the session, dynamically repositioning it to relevant locations, such as home or the drone. This approach greatly simplified the implementation and avoided additional complications.

To achieve this movement, Unreal Engine's Arrow component was used. This seemingly simple component allows you to fix a relative position and orientation with respect to the parent object, making it easy to follow the drone. By parenting the pawn to an Arrow attached to the drone, it could automatically follow the drone's relative position without requiring constant calculations. The same method was used to anchor the pawn to the home position. During this process, some issues with object hierarchies appeared due to improper handling of parenting, but careful management of the parenting order ultimately resolved the problems.

Progressive Locomotion Improvements

The initial teleportation-based locomotion system was improved through several enhancements. A more intuitive and representative visual marker (an arrow) was added to clearly indicate the location of the target teleport. Later, to further reduce motion sickness caused by abrupt position changes, a fade effect was implemented before and after each teleportation.

Additionally, virtual rotation via external input was added, allowing the user to change their viewing direction artificially and thus avoid uncomfortable physical turns with the VR headset.

Problems with Drone Tracking

A serious technical issue emerged when integrating drone telemetry via WebSockets. Due to the low frequency and irregular timing of the position update messages, the drone's movement in the VR environment appeared jerky, quickly inducing severe motion sickness symptoms in users, even those experienced with VR.

To quickly mitigate this problem, the Spring Arm component was initially used. It smooths movement through a visual effect similar to an elastic cord connecting the drone to the user's pawn. While initially effective, it soon created further issues, particularly with the Widget Interaction system (similar as virtual fingers), which became uncoordinated with the hands' visual positions due to the smoothing applied by the Spring Arm.

After multiple failed attempts to resolve this incompatibility, a new technical decision was made: instead of smoothing the user's movement, the drone's visual position was slightly delayed, interpolating between previously received messages. Although this introduced a small increase in latency in drone flight visualization, it produced smooth, seamless movement, and definitively resolved the earlier problems.

Additional Solutions to Mitigate Motion Sickness

Given that tracking the drone in flight involves continuous movement and changes in altitude (two major contributors to VR nausea) an additional solution was implemented: the "tunneling vision" technique. This method artificially limits peripheral vision during critical flight moments, significantly reducing discomfort caused by continuous motion.

Initial attempts to implement this solution encountered technical challenges due to how Unreal Engine renders VR images. However, after research and testing, an effective solution was found using a plane with a specific texture that creates a gradual peripheral vision reduction effect, resulting in a more comfortable experience.

Brief Note on Cesium Integration

As detailed elsewhere, integrating with Cesium did not pose additional technical difficulties beyond the requirement for a stable Internet connection to load tiles in real time. However, as a preventive measure for future hardware limitations, a fog effect (Exponential Height Fog) was included to limit draw distance and reduce graphical load in more constrained scenarios.

4.2 Slice 1: Real-time Telemetry

At the beginning of the project, the desktop version featured a rudimentary telemetry visualization system implemented through a basic widget that simply displayed numerical data in a large area of the screen. This system fulfilled its purpose in a basic manner, providing key drone information during flight, but without advanced processing or visualization.

When migrating to virtual reality (VR), the logical first step was to reuse this same visualization Figure 4.1, while a more adapted and visually advanced system was developed in parallel to fully exploit the capabilities of VR technology.

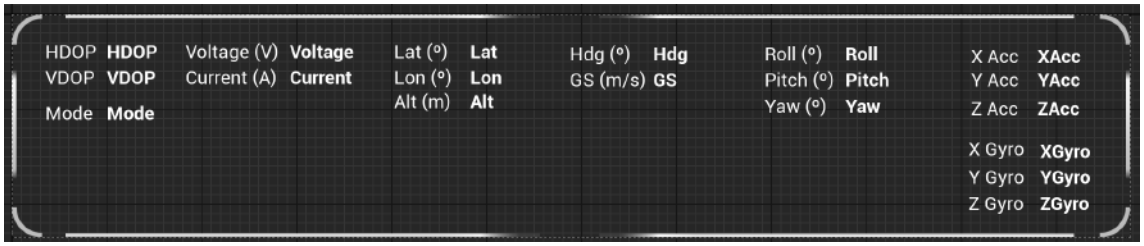

Figure 4.1: Telemetry window old design

Initial Adaptation to the VR Environment

The first adaptation involved creating a virtual floating window that could be adjusted in location and distance within the user's view. This design aimed to provide a familiar experience for users accustomed to traditional screens, striking a balance between innovation and usability.

To facilitate interaction, a simple animation was implemented to highlight the window when the user aimed the virtual pointer (widget interaction) at it, clearly

indicating when it could be moved. This interaction initially presented issues due to channel conflicts, which were resolved after a careful analysis and specific adjustments.

The initial implementation for moving the window relied on built-in tools from Unreal Engine's VR template. These tools, originally designed for grabbing physical objects, had to be adapted to allow "grabbing" from a distance without applying gravity. This interaction was also ensured to affect only the telemetry window, preventing accidental movements of other objects such as the drone or map.

A problem soon emerged: when the pawn moved through the map, the window remained static in the position where it had been released. This was easily resolved using the previously developed object parenting system, allowing the window to remain anchored at a relative position to the pawn.

Visual and Functional Evolution of the Telemetry

After several iterations, it became clear that the initial telemetry presentation did not meet the goal of providing a significantly superior visualization compared to traditional formats. To address this, work sessions were held with the thesis advisor, who contributed operational experience in drone flight. From this collaboration, clear criteria emerged for how each data type should be displayed to maximize usefulness and speed of interpretation.

For instance, the battery level, crucial for evaluating the remaining drone autonomy, did not require exact numerical precision. Thus, a simple visual bar was used to indicate an approximate percentage. Similarly, data like angular accelerations, less relevant numerically but critical when showing sudden changes, were represented with graphic sliders that clearly highlighted any anomalies.

For fundamental data such as tilt or speed, more prominent and precise numerical representations were maintained. Others, like coordinates, were kept in the background due to their only occasional importance. Finally, GPS accuracy was visually indicated through a color-changing icon based on connection quality, allowing quick user interpretation.

Figure 4.2: Telemetry window advanced design

Internal Data Processing

Implementing the system involved not only visual improvements but also specific internal processing of certain data. For example, the battery percentage shown is not received directly from telemetry but is calculated from the voltage and the number of battery cells specified by the user in the main menu.

Other data, such as GPS precision, are processed using geometric means of the received values, establishing thresholds to visually show the connection image color status. Likewise, to display the drone's relative direction to north and the home position, internal calculations determine the angle formed by the drone's direction vector and the vector connecting the drone and home positions. This angle indicates the direction needed to correctly position the home signal.

This processing required extensive testing to adjust and validate the methods used, ensuring both accuracy and clarity in the displayed information.

Experimentation with Different Display Formats

Since telemetry data is fundamental for piloting, alternative ways of presenting it in VR were explored. Initially, the adjustable floating window was used, but a fixed visualization format, such as a virtual helmet HUD, was also tested, keeping the data always visible in front of the user.

After implementing both and collecting detailed feedback from various users, it became evident that the fixed visualization had significant drawbacks: the abundance of peripheral information was ineffective due to low resolution on the edges of the VR field of view, and users found it uncomfortable and unnatural to turn their heads rather than just their eyes to read data.

The conclusion was clear: the original interactive floating window Figure 4.2 proved to be the best option, even after weeks of work on the fixed display. This process, although initially delaying other features, provided valuable insights into the interaction and visualization particularities of VR environments.

Refactoring for Modularity and Scalability

In the later stages of development, after confirming the effectiveness and potential of the interactive floating window, the code was refactored to enhance modularity. A base class for floating windows was created, containing all the essential logic, which greatly simplified the creation of new windows designed to display different types of information.

This refactoring significantly improved the internal organization of the code, simplified future program extensions, and allowed faster and more efficient adaptation to new needs identified during the project's development.

4.3 Slice 2: Graphs Window

From the beginning of the project, one of the key objectives was to implement a functional graphing system that would enable advanced tracking of telemetry data, visually getting over existing alternatives, and providing a clear historical view of critical flight variables. However, developing a complete visual graphing system from scratch in Unreal Engine represented a highly complex task that could have required months of intensive work, including design, development, and debugging.

To address this time constraint, a basic charting system compatible with Unreal was purchased from the official Fab asset store (Charts Pro Scatter Plot Line). While acquiring a pre-developed asset might seem like it would greatly simplify the task, the reality turned out to be more complicated than expected.

Initial Problems with the Acquired Asset

Once acquired, it was discovered that the charting system did not function exactly as advertised. It had minor but significant bugs, and although some variables were documented for customization, the documentation was clearly insufficient for direct integration.

The first challenge was understanding and reorganizing an extremely chaotic and poorly documented Blueprint-based code structure. Blueprint graphs, if not

clearly organized and annotated in functional blocks, are very difficult for external developers to understand. The initial task involved meticulously reorganizing these graphs, adding essential comments to clarify the structure and facilitate understanding.

This process required deep immersion into the asset's various classes and functions to identify exactly where errors occurred, how the different components interacted, and how they could be adapted to the specific needs of the project. While arduous, this exercise emphasized the critical importance of maintaining clean, well-documented, and properly commented code for future developers.

Adaptation and Initial Implementation

After fixing the initial bugs and gaining a thorough understanding of how the charting system worked, it was integrated with the floating window base class developed in earlier stages of the project Figure 4.3. This quickly allowed for the reuse of already implemented dynamic behavior for telemetry windows, resulting in a floating, adjustable graph window that displayed the historical data of a specific variable in real time.

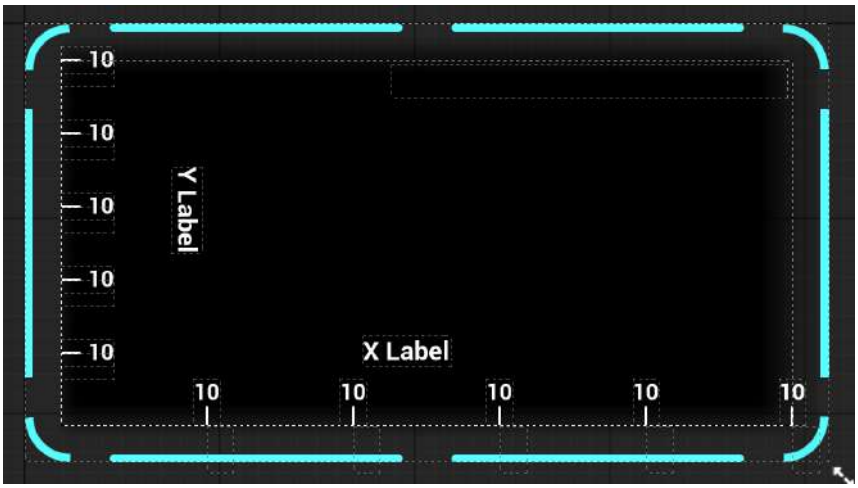

Figure 4.3: Graph window design screenshot

New Requirements and the Tryptich Graph Proposal

In a later iteration of the project, the idea emerged to display multiple graphs simultaneously, each representing different variables. Initially, the simplest solu-

tion was considered: generating multiple independent instances of the already developed graph object. However, several issues quickly became apparent:

- How to dynamically select the variable to display in each generated graph?
- How to allow users to intuitively close individual graphs?
- How to manage the efficient and comfortable placement of multiple graph windows in the user's field of view?

These problems revealed major limitations in the initial approach, leading to the exploration of a more comprehensive and robust alternative solution.

After careful analysis and reflection, the tryptich graph solution was proposed. Although initially more complex in terms of design and development, it offered a complete and perfectly fit response to the identified needs:

- It made conceptual sense because the most relevant variables for graphical tracking were speed (a simple variable) and linear accelerations along the three axes (complex but highly relevant for detecting flight anomalies).
- It enabled clear, simultaneous visualization while maintaining visual organization and avoiding additional complications in managing and positioning individual windows.

After discussions with the TFG advisor, it was decided that the default single graph would show speed, and when multi-graph viewing was enabled, the system would display the three axes of linear acceleration simultaneously.

Technical Implementation of the Tryptich

Implementation began by establishing a clear hierarchy among graph objects, thus creating the concept of a 'parent' graph and 'child' graphs. This hierarchy allowed centralized handling of key functionalities like hover detection and group movement.

Initially, several technical issues arose:

- When trying to move a child graph, it became unlinked from the tryptich. This was resolved by fully centralizing control of movements and animations in the parent graph.
- When applying inherited dynamic behavior from the base window class (automatic rotation to face the user), another visual problem emerged. Each graph had its pivot point at the center, causing the tryptich to visually deform when turning to face the user, breaking cohesion.

To resolve this issue, the pivot points of the child graphs were repositioned laterally. This turned the tryptich into a layout similar to a classic triple-monitor setup, with each graph slightly angled toward the user, optimizing visibility and providing a smooth, natural transition between screens Figure 4.4.

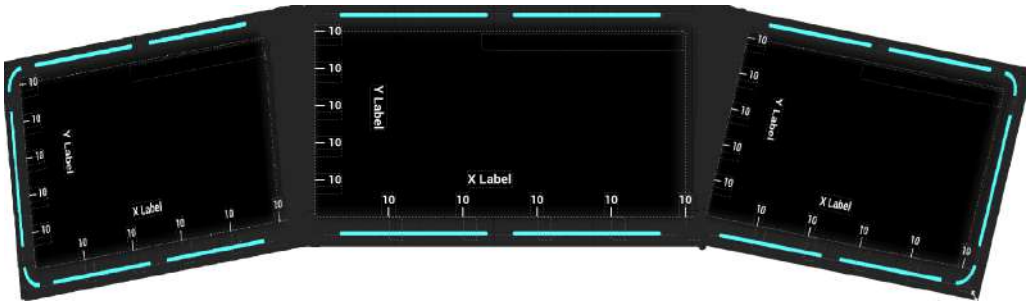

Figure 4.4: Tryptich graphs reconstruction

This final solution provided an excellent visual experience and enabled intuitive user interaction, efficiently resolving previously identified problems. The process also reinforced the importance of designing complete solutions from the outset, even if they require more initial development time, as they prevent future complications and offer greater scalability.

As an additional result, clear in-code documentation was created to facilitate future modifications, including precise instructions on how to change the default displayed variables and how to manage future interactions easily.

4.4 Slice 3: Collision Detector System

During one of the intermediate meetings, upon observing the strong development progress of the project, it was proposed to expand the initial objectives by incorpo-

rating new functionalities that would add real value to the flight experience and enhance operational safety. One of the most ambitious proposals was the development of a real-time collision prevention system. Although initially overwhelming due to its technical complexity, this functionality was adopted as a motivating challenge that would give the system a higher level of sophistication.

Preliminary Approaches and Detected Issues

The first idea was simple but quickly discarded: to generate a single physical sphere around the drone capable of detecting collisions. However, this solution posed serious performance and scalability problems, especially in a complex 3D environment with multiple objects loaded via Cesium.

Next, an intermediate solution was explored using three concentric collision spheres, each with a different radius. The logic was clear: if an object entered the innermost sphere, the collision should be considered more severe than if it occurred in an outer sphere. This hierarchy would allow for the establishment of visual alert priorities and urgency. However, a relevant technical obstacle emerged during implementation: Unreal Engine does not natively provide the exact point of contact for a collision with a sphere, making it impossible to determine the direction from which the obstacle approached. Attempts were made to resolve this using the center of the collided object as a reference, but since all collidable objects were Cesium tiles (which are large in size), the results were inaccurate and ultimately useless.

New Solution: 3D Ray Matrix

Frustrated by the limitations of the previous methods, the system was completely redesigned and a new approach was proposed: to cast multiple rays in all directions around the drone. Each of these rays would simulate a “detection antenna” that would return both the existence of a collision and the distance to it. Based on this information, the direction of the nearest object and the urgency of the alert could be precisely determined.

The final system included 25 rays strategically distributed Figure 4.5, uniformly

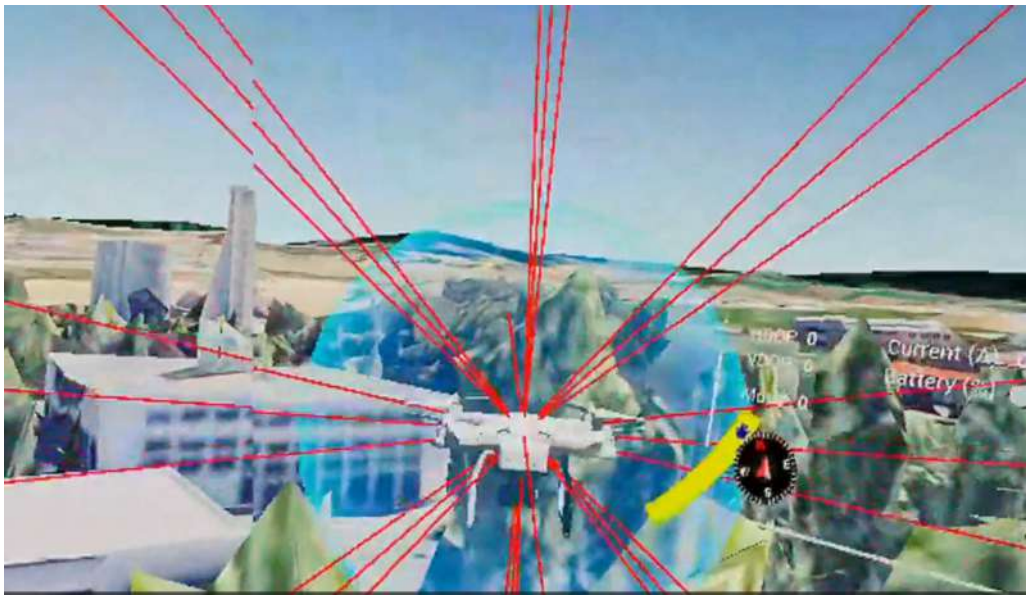

Figure 4.5: Collision detection rays

covering the three-dimensional sphere surrounding the drone. Each of these rays, represented by a unit direction vector, was stored in an array along with an identifying tag that facilitated its traceability within the alert system. To optimize performance, the rays were cast twice per second instead of on every frame.

During each iteration, an individual collision check was performed for every ray. If an impact was detected, a priority value was calculated based on the distance to the collision, weighted by the original length of the ray. This calculation allowed for distinguishing between nearby collisions in less critical directions and more distant ones in high-risk areas, such as the front. The full logic of this process is described in algorithm algorithm 1, which outlines how all detections are evaluated and the highest-priority one is selected as the final warning. This priority score triggered the visual alert system, providing the pilot with clear information about the direction and urgency of the detected threat.

Algorithm 1: Collision Detection and Prioritization

Input: $R = \{r_1, r_2, \ldots, r_{25}\}$: set of rays, each with direction vector $\vec{d_i}$ and maximum length L_i,
$T = 0.5\,\text{s}$: update interval
Output: Direction and alert level of the highest-priority collision, or `None` if no collisions

while *the application is running* **do**
    `sleep(`T`);`
    $C \leftarrow \emptyset$; `% detected collision list`
    foreach $r_i \in R$ **do**
        $((hit, d)) \leftarrow$ `RayCast(`$\vec{d_i}, L_i$`);`
        if *hit* **then**
            $C \leftarrow C \cup \{(i, L_i, d)\}$; `% store index, length, and distance`
    if $C \neq \emptyset$ **then**
        Define $P = [\, p_i \mid (i, L_i, d_i) \in C\,]$, $p_i = L_i / d_i$;
        $k \leftarrow \arg\max_j P[j]$, $(i^*, L^*, d^*) \leftarrow C[k]$;
        `ShowAlert(`$i^*, L^*/d^*$`);`
    else
        `ClearAlerts();`

Visual Design of the Alert System

To display the direction of the detected threat to the user, a directional disc was designed to be visible in the VR headset. In the event of a collision, the segment of the disc corresponding to the direction of impact would light up, enabling quick and intuitive understanding of the environment. Additionally, the priority of the collision, calculated based on the proximity of the obstacle to the drone, was visually encoded using color: the more intense and warmer the color, the higher the danger Figure 4.6.

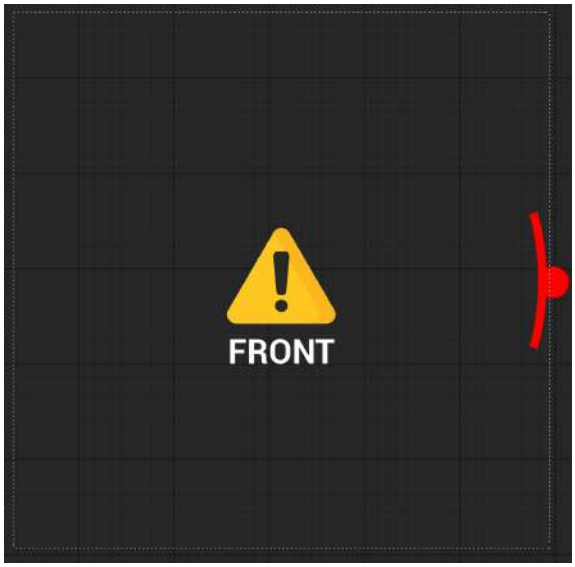

Figure 4.6: Collision widget design screenshot

In cases of frontal or rear collisions, where a segment of the disc might be less intuitive, independent and clearly visible triangular markers were added. These indicators, placed in the center of the alert disc, clearly warned when the collision came from those extreme angles.

Frontal System Refinement: Priority and Coverage

In a later stage of development, an additional issue was identified: due to the natural inertia of flight, frontal collisions needed to be detected earlier than lateral ones. Simply increasing the length of the frontal rays was not a complete solution: as the distance increased, so did the space between rays, resulting in blind spots. Furthermore, calculating priority solely based on distance introduced the risk that a nearby lateral collision might be prioritized over a more critical frontal one.

The adopted solution is detailed in algorithm algorithm 2. First, a “directional importance factor” was added to the priority calculation, boosting the weight of rays labeled as frontal by multiplying their final score. This adjustment helped prioritize collisions along the likely flight path.

Additionally, to enhance coverage in the forward direction and detect threats situated between rays, a progressive detection sphere system was implemented. If none of the frontal rays detected a collision, up to three concentric spheres were generated in front of the drone, with exponentially increasing radius and distances from the center. These spheres expanded the detection angle without increasing the computational load of new rays.

When one of these spheres detected a collision, a virtual alert equivalent to that of a ray was triggered, assigning it a priority based on relative distance and applying the same directional factor. This approach ensured early, continuous, and more robust detection even in complex flight paths or partially obstructed environments.

Algorithm 2: Frontal Priority with Progressive Sphere Fallback

Input: $F = \{i_1, \ldots, i_f\}$: indices of frontal rays within R,
$R = \{(\vec{d}_i, L_i)\}_{i=1..25}$: ray set, direction $\vec{d}_i$, length L_i,
α: frontal importance multiplier,
$N_s = 3$: fallback sphere count,
D_0, R_0: initial distance and radius,
$T = 0.5\,\text{s}$: update interval
Output: Direction index i^* and priority p^* for alert, or `None`

while *the system is running* **do**
  `sleep(`T`);`
  $p^* \leftarrow 0$, $i^* \leftarrow$ `None`;
  for $i \leftarrow 1$ **to** *25* **do**
    $(hit, d) \leftarrow$ `CastRay(`$\vec{d}_i, L_i$`)`;
    if *hit* **then**
      $p \leftarrow L_i / d$;
      if $i \in F$ **then**
        $p \leftarrow \alpha \cdot p$
      ;
      if $p > p^*$ **then**
        $p^* \leftarrow p$, $i^* \leftarrow i$
      ;
  if $i^* \neq$ *None* **then**
    `ShowAlert(`i^*, p^*`)`; **continue**
  for $k \leftarrow 1$ **to** N_s **do**
    $D_k \leftarrow D_0 \cdot 2^{k-1}$, $R_k \leftarrow R_0 \cdot 2^{k-1}$;
    $C_k \leftarrow \text{origin} + D_k \cdot \text{forwardVector}$;
    $(hit, d_c) \leftarrow$ `CheckSphere(`C_k, R_k`)`;
    if *hit* **then**
      $p \leftarrow \alpha \cdot (D_{N_s} / d_c)$;
      `ShowAlert(`*frontal,* p`)`; **break**;
  if *no sphere hit* **then**
    `ClearAlerts()`

Visual Reinforcement: Dynamic Alert Sphere

Although the directional disk proved functional, it was not always visually striking enough, especially during complex flights or when the pilot was focused on the surrounding environment. For this reason, a new complementary visualization was introduced: a semi-transparent sphere around the drone, whose scale and color vary according to the level of danger detected by the ray system.

This solution, highly visual and seamlessly integrated into the 3D scene, was effective in both functionality and development, as it directly leveraged the priority value calculated by the system. Furthermore, its computational cost was low, allowing smooth updates without compromising the overall performance of the program.

4.5 Slice 4: Notification System

Just like with the collision detection system, the idea of incorporating an advanced visual notification system emerged as part of the project's progressive expansion, once the core elements began to consolidate. From the outset, it was approached with a rigorous and planned methodology, prioritizing modularity, component decoupling, and system scalability with a view toward potential future expansions.

The design was based on a clear and structured model from the beginning (Figure 4.8), which made it possible to maintain a clean and flexible architecture throughout the development process.

System Structure and Main Classes

The system is composed of five main classes:

- **BP_Check:** Logical class that encapsulates various types of value checks (greater than, less than, equal to, within ranges...). Each Check object operates with a value to verify, a limit or set of limits, and the type of comparison. Although it is currently focused on basic checks, its design easily scales to more complex validations.

- **BP_CheckManager:** Class that orchestrates checks. It maintains a list of active checks and evaluates them on each tick. When an alert condition is met, it triggers a signal through an interface, delegating the visual handling to other components.

- **BPI_Notification:** Interface that acts as a bridge between the check logic and visualization. It allows warnings to be emitted without maintaining direct class references, which enhances system decoupling and simplifies maintenance and expansion.

- **WBP_Notification:** Visual class that represents a specific notification (Figure 4.7). It contains graphic elements (background, icon, text) and manages its own lifetime and entry/exit animations.

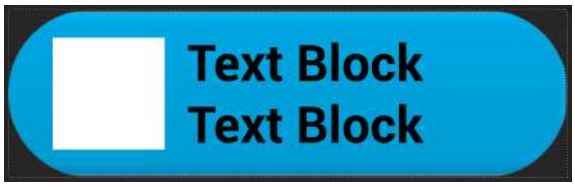

Figure 4.7: Base notification design screenshot

- **WBP_NotificationManager:** View manager class. It maintains a pool of 5 inactive notification instances, which are activated when new alerts are received. This object pooling approach avoids constant object creation/destruction, reduces system load, and provides better control over active resources.

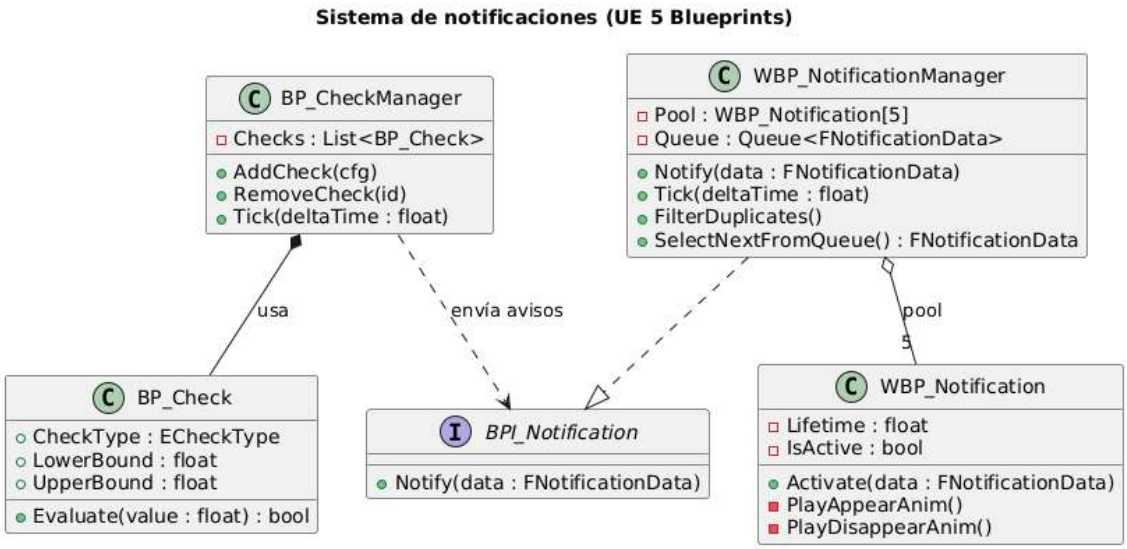

Figure 4.8: Notification system Class Diagram

General System Flow

The complete flow begins when the CheckManager detects an alert condition. At that moment, a warning is sent via the Notification interface.

Upon receiving this alert, the NotificationManager applies a series of filters before displaying it:

- **Avoids duplicates:** if an identical notification is already active, its duration is extended instead of creating a new one.
- **Checks availability:** if one of the five notifications is free, it is assigned the appropriate parameters and activated for 2.5 seconds with an appearance animation.
- **Queue management:** if no notifications are available, the new one is queued with its priority and accumulated waiting time.

When an active notification expires, two processes are triggered:

- **Visual elevation:** lower notifications move up to occupy the vacated space, simulating continuity.
- **Queue injection:** the queued notification with the highest weighted value between priority and wait time is selected, ensuring that even low-priority alerts are eventually displayed.

This dual system ensures a smooth experience, without cluttering the screen or losing important alerts.

Visual Iterations: Animations and Refinements

In the first phase, the system was designed and developed with the full logical and functional flow described above, but without incorporating secondary behavior animations, aside from a basic entry and exit.

In later iterations, and as part of the product's visual polish, several key animations were added:

- **Elevation animation:** when a notification expires, those below it rise with a visual transition simulating continuity, avoiding abrupt jumps.
- **Idle animation:** each active notification adopts a subtle slow pulsing effect (expansion and contraction), like a constant bubble that draws attention without being intrusive.
- **Appearance/disappearance refinement:** the initial fade in/out and scaling animations were fine-tuned to be smoother and more natural.

These improvements significantly increased the readability of alerts and their integration with the rest of the VR interface.

All of this made the notification system one of the most robust and reusable components of the entire application. The clear separation between logic, interface, and visual representation allowed for rapid iteration without rewriting large code blocks.

Moreover, its scalability-oriented design makes it an ideal candidate for future expansions, such as integration with voice alerts, haptic feedback, or adaptive prioritization based on flight context. The initial planning effort, though costly, resulted in a solid, smooth, and visually appealing implementation perfectly suited for the virtual reality environment.

4.6 Slice 5: Hand menu

The need for an effective control system within virtual reality arose directly from the transition from the original desktop program, which was controlled with a keyboard and mouse. Initially, the idea was to assign the main system functions to the physical buttons on the VR controller, but this approach was quickly discarded due to its poor scalability and incompatibility with hand-tracking, one of the functionalities planned for later stages of the project.

The proposed solution was to develop a contextual menu anchored to the user's hand, thus integrating the controls naturally and intuitively within the virtual environment. This proposal reduced reliance on physical buttons and opened the

door to a smooth transition to controller-free implementation while maintaining immersion and ergonomics.

Menu development with VR controllers

During the early stages of development, the VR test scene provided by Unreal Engine was reused as a base, including a small widget with buttons for exiting or pausing. This snippet served as inspiration to explore a possible forearm-based interface.

The system started with the pointer based on Widget Interaction Components and joystick control, although the original system allowed a cursor to slowly move across the interface, which was impractical. The solution was to transform this menu into a radial structure: the buttons were distributed in a ring around a central button Figure 4.10, and an angular selection system was implemented using the joystick's orientation, dividing $360°$ into five equal zones, each associated with a button. When the joystick was left in its neutral position, the central button was selected by default. This design preserved a consistent aesthetic and a clear organization of functions.

Figure 4.9: Base button hover design screenshot

Each button was designed with a circular shape Figure 4.9, incorporating a small icon, a descriptive label, and a visual animation when hovered. This resulted in a general-purpose design that centralized functionality and maintained

interface consistency.

Integrated functionality and modular logic

With this system, users could quickly access functions such as:

- Teleport to home
- Teleport to the drone
- Show or hide the telemetry window
- Activate one or three tracking graphs
- Exit the application

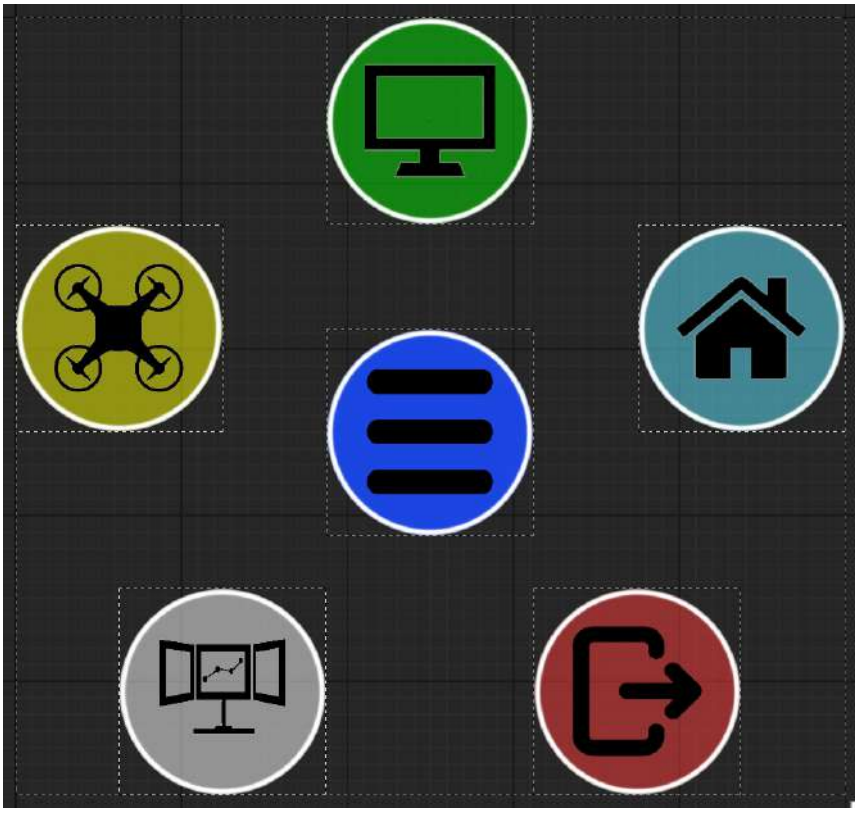

Figure 4.10: Hand menu design screenshot

These functionalities could be easily assigned thanks to the modular menu architecture. Each button represented a reusable instance with individualized behavior, all managed from a central structure.

Transition to hand-tracking and menu redesign

Once the hand-tracking system was integrated, it became clear that maintaining the same floating forearm menu would be insufficient. A complete redesign was carried out to take full advantage of the new finger-tracking capability.

A direct correspondence was identified between the circular buttons and the five fingers of one hand, with a central button (now represented in the palm). This observation led to the conceptualization of a new menu in which each button was placed directly over a fingertip Figure 4.11, allowing the user to activate a function by simply touching that finger with the index finger of the other hand. This solution was more natural, faster, and preserved full immersion in the environment.

Figure 4.11: Hand-tracking menu screenshot

Inherited architecture and interaction management

To implement this new structure, a “menu base class” was created, from which both the traditional menu and the individual finger mini-menus inherited. This inheritance pattern made it possible to maintain common logic (positioning, animations, reference control) without duplicating functionality.

One of the most significant technical challenges was coordinating events between buttons. Since the fingers are close together, it was common for a single gesture to trigger more than one button at the same time. To prevent this, a

higher-level interaction controller object was created to validate or block impulses based on the time since the last activation.

Additionally, an internal interface was designed to allow the buttons to communicate with the main menu, managing shared references such as the visibility of the telemetry window. This decision was key to enabling dynamic switching between controller and hand modes without losing data or functional links.

Visual adjustments and usability

Due to the limited distance between fingers in hand-tracking mode, the button scale was reduced to prevent overlapping. Nevertheless, to maintain legibility and action clarity, a special animation was implemented: when hovering over a button, it would temporarily enlarge and clearly display its function. This effect significantly improved the user experience by visually confirming the action before execution.

The final result was a dual menu system, one for controllers and one for hands, sharing the same base logic and adapting to the detected control type, while always maintaining visual and functional consistency within the environment.

4.7 Slice 6: Hand-tracking & Motion Sickness

As the project progressed and the core functionalities related to drone control and telemetry monitoring were consolidated, a lack of features focused on enhancing user immersion in the VR environment became apparent. During various review meetings, several proposals emerged to address this imbalance, among which three priority lines of action stood out (some of them previously explained):

- Redesign the main menu to adapt it to a more physical and immersive interaction.
- Take active measures against motion sickness, which is common in poorly optimized VR environments.
- Implement innovative control technologies such as hand-tracking (HT) and voice recognition.

These initiatives aimed to reinforce the user's sense of presence and natural interaction within the simulation, going beyond a purely functional experience.

Measures Against Motion Sickness

The first group of solutions focused on combating motion sickness, a common discomfort for VR users, especially in experiences involving movement or abrupt interaction.

Figure 4.12: Tunneling Vision Screenshot

Three major potential sources of nausea were identified, and specific solutions were designed for each:

- **Teleportation:** a short delay was introduced, accompanied by fade-out and fade-in effects, to soften the instant transition from one position to another, avoiding sudden disorientation.
- **Continuous drone tracking:** as explained in previous slices, drone movement was smoothed using temporal interpolation between telemetry messages, eliminating the “jumping” effect that was especially uncomfortable.
- **Fast first-person movement:** the *vision tunneling* technique Figure 4.12 was implemented, which progressively darkens peripheral vision during prolonged or elevated movements, reducing the perception of speed.

These three measures, applied together, significantly improved user comfort without sacrificing visual fidelity or environmental control.

First Steps in Hand-Tracking Integration

The second line of work focused on implementing gesture-based control, leveraging the hand-tracking system included in the official Meta XR plugin. The goal was to allow navigation and interaction without physical controllers, offering a more intuitive and futuristic experience.

The project was first adapted to accept dual input (controllers + hands), followed by a thorough review of documentation and available examples of predefined system gestures. Once the most reliable and stable gestures were identified, the two basic control actions were implemented: teleportation and camera rotation. Both systems were successfully adapted, confirming the technical practicability of the approach.

Adapting the Hand Menu to Hand-Tracking

One of the most complex challenges was adapting the hand menu, previously controlled with a joystick, to the new gesture-based control paradigm. As detailed in the corresponding slice, the menu was completely redesigned so that each button was anchored to one of the user's fingers, with a central button located in the palm.

The interaction was designed as a natural gesture: the user would touch the desired finger's button with the index finger of the other hand. This solution proved to be highly immersive and aligned with the user's body perception, who can intuitively sense the position of their fingers.

Technical Obstacles and Solutions

Implementation was not without challenges. One of the first issues was accurately retrieving the bones from the hand model generated by Meta's plugin, which is quite encapsulated and poorly documented. After intensive investigation, it was possible to access the necessary bones to:

- Anchor the menu buttons to the corresponding fingers.
- Position the Widget Interaction on the index finger to trigger events.

Another significant issue was the need to add collisions to the virtual hand to allow interaction with the main menu, which was also resolved through studying the internal structure of the tracking system.

A more complex challenge arose when attempting to implement the movement of floating windows (like telemetry or graphs) via hand-tracking. The intention was to combine pointing with the index finger and a “grab” gesture to move the object. However, current gesture recognition technology does not always accurately detect finger positions if they move outside the headset's camera view. This caused the grab action to be unintentionally interrupted, leading to loss of control over the windows.

Although alternative solutions were considered, it was concluded that this limitation is inherent to the current hardware and is expected to improve in future SDK versions or through AI integration. For now, this interaction remains an experimental feature with room for improvement.

Voice Control: Prepared But Not Implemented

In parallel, research began on integrating AI-powered voice commands using Meta's Voice SDK. The project was configured to support this extension, but due to time constraints, it was not implemented. However, the design and architecture are ready to incorporate it as a natural evolution of the system.

4.8 Slice 7: 3D Physical Main Menu

The design of the main menu, like many other functionalities in the project, originates from the original desktop application. This initial menu Figure 4.13 consisted of a 2D interface interactable with a mouse, where the user could change the date and time of the Cesium scenario as well as start the flight session.

At the beginning of the VR development, this interface was reused by instantiating the widget in an empty room, where the user could operate the menu using the VR pawn’s Widget Interactions (WI), in a way similar to the desktop version. This solution was kept during a large part of the development, as it was not a priority in the early stages.

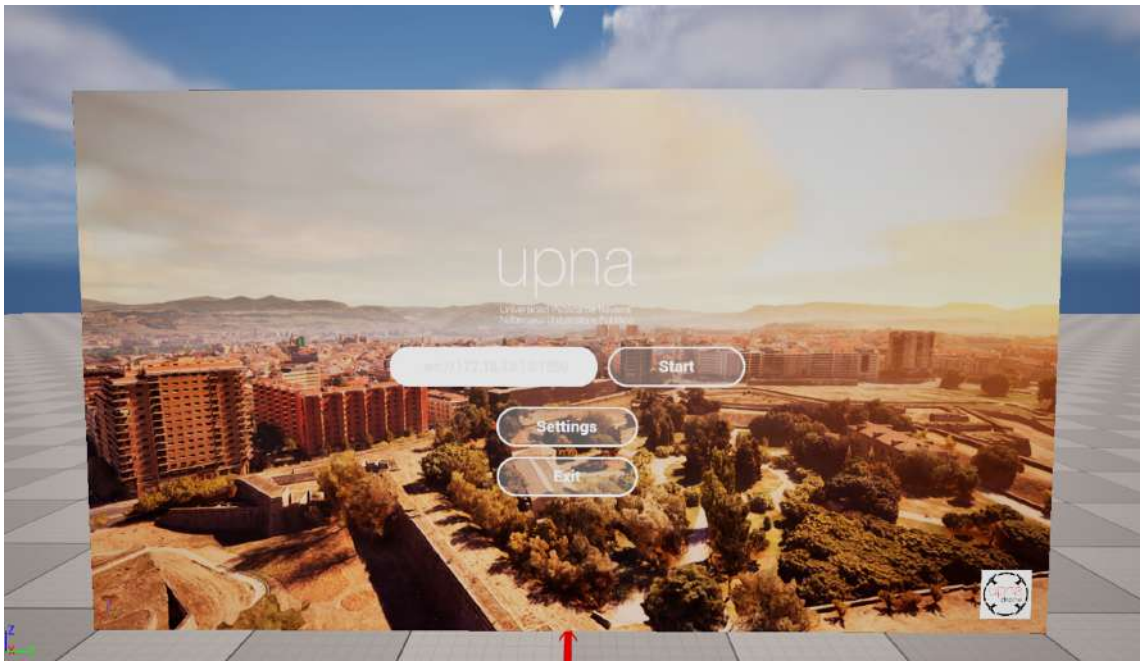

Figure 4.13: Initial main menu 2D screenshot

However, as the rest of the project advanced and improved visually and functionally, this menu began to feel out of place. Its simplicity and poor integration with the immersive experience contrasted with the level of polish of the rest of the application. It became clear that the user's first experience had to match the overall quality of the system, prompting a complete redesign of the main menu from scratch.

Environment and Visual Style

A search was initiated for scenarios and assets that evoked a technological and futuristic environment, such as a drone hangar or an air control tower. After several tests, a floating control tower overlooking a suspended runway was selected Figure 4.14, with a sci-fi aesthetic. The scenario was adapted to the VR environment by modifying incompatible materials (glass and others) and editing the collision mesh to allow the user to move inside the space.

The new menu adopted a style based on transparency, holograms, and blue tones, conveying modernity without overloading the environment. This palette aligned with the visual design of the entire application, reinforcing the overall coherence of the product.

Figure 4.14: Main menu surroundings screenshot

Physical Interaction: 3D Buttons

Inspired by the success of the hand-tracking menu, it was decided that the new menu should rely on direct physical interaction to enhance immersion. This led to the idea of replacing traditional flat buttons with physical 3D buttons, inspired by the real behavior of mechanical keys: a body that depresses under finger pressure and activates at the lowest point.

Figure 4.15: Button 3D design screenshot

Semi-transparent cylinders with floating rings and central text were designed Figure 4.15, detecting proximity of the controllers or hand and physically sliding to the bottom of their path. This interaction was highly intuitive and satisfying. Thanks to their modular structure, these buttons could be easily replicated to create different commands, all sharing the same base logic.

Physical sliders for value input

One of the challenges when transitioning to a 3D environment was implementing the selection of values like day, month, and time. In the classic interface, this was done through numerical input fields. Other conventional approaches in virtual environments include increment/decrement buttons. The first option was unfeasible in VR due to the high development cost of implementing a virtual keyboard, while the second proved too limited, expensive, and unintuitive; completely contrary to the main objective of improving this menu.

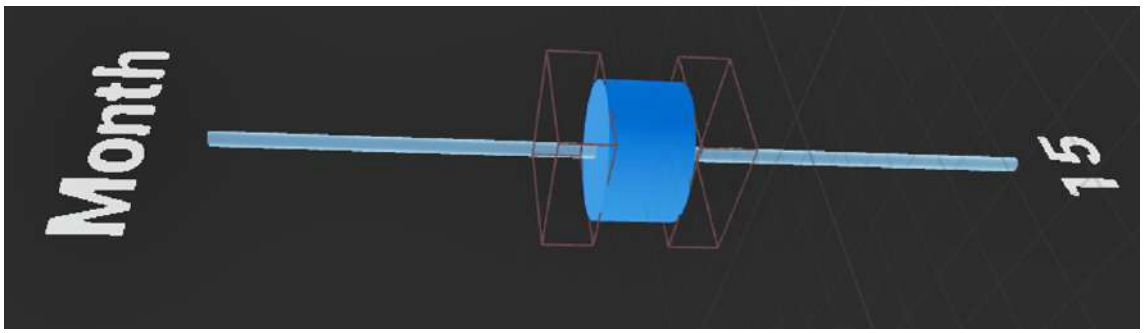

Figure 4.16: Slider 3D design screenshot

The solution was to create a physical 3D slider: a rail and a movable disc Figure 4.16, which shifted laterally depending on the finger's approach direction. The resulting value was calculated as a proportionally interpolated position along the axis and dynamically displayed on screen.

This component was also designed to be generic and reusable, allowing future application to any parameter requiring continuous numerical selection.

Menu structure and screens

The final menu included several functional screens:

- **Main screen:** with three physical buttons, the central one ('Start') being visually dominant. Around it, the 'Settings' and 'Exit' buttons complete the layout Figure 4.17.
- **General settings screen:** with three physical sliders for Cesium time selection, a button to return to the previous screen, and two additional buttons later added to navigate to the next two screens Figure 4.18.
- **User settings screen:** added in later development stages to let the user input essential telemetry parameters, such as the number of drone battery cells and other future extensions.
- **Help/documentation screen:** includes a brief visual guide to controls, accessible without external documentation. It is operated via a pair of advanced buttons.

Figure 4.17: New main menu, main screen design screenshot

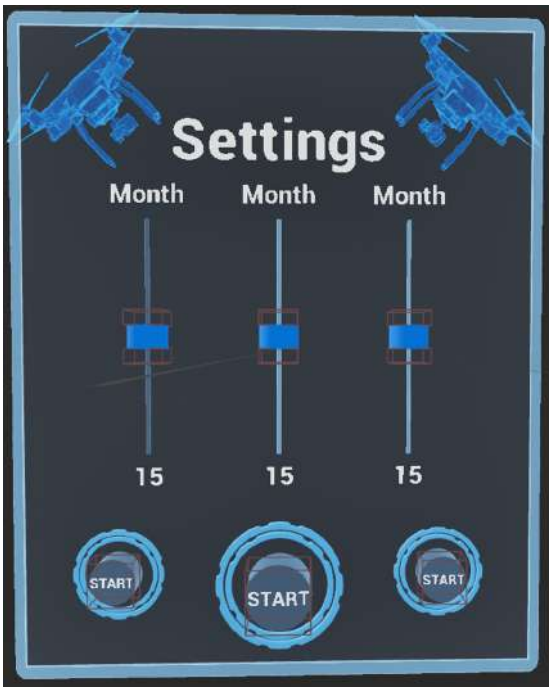

Figure 4.18: New main menu, settings design screenshot

To address the coexistence of both control modes (controllers and hand-tracking), a unified interface was designed to display both schemes, toggleable via stateful 3D buttons (inheriting from the generic button to retain functionality). These new

buttons could be activated or deactivated, and when pressed, would automatically signal the opposite button to deactivate, avoiding overlap or errors.

Sensory Feedback and Final Animations

In the final stages of development, several user experience improvements were introduced:

- Smooth and stylized **transition animations** between menu screens.
- **Haptic feedback** when pressing buttons or moving sliders, with varying intensities depending on the type of interaction.
- **Auditory feedback** with brief sound effects: confirmation clicks, sliding sounds, etc.

All these responses were encapsulated within the internal logic of buttons and sliders, so that inheriting from them preserved consistent and unified behavior without code duplication.

The new physical main menu not only addressed the shortcomings of the original one but became one of the most immersive experiences in the entire application. From the visual ambiance to the ergonomic design of its physical components, along with the precise logic of its controls, this section sets the overall tone of quality and immersion from the very first moment the user activates the system.

Chapter 5

Testing and Validation

The validation process of a technological application is essential to ensure its proper functioning, performance, and usability. For this application, a structured and exhaustive manual testing approach was chosen, with a strong focus on validating the system's overall functionality, usability, and performance, as well as that of each individual module.

5.1 Methodology and Testing Environment

Testing was divided into three main stages, each aimed at ensuring different critical aspects of the system. The first stage involved manual unit testing during the development of each *slice*. Thanks to the project's modular structure based on slices and the centralization of telemetry in the *GameInstance* class, it was possible to isolate and independently validate each component of the system before final integration. This method greatly facilitated the early detection and correction of bugs, preventing issues from propagating to later development phases.

The second stage consisted of intermediate validation tests, primarily conducted during monthly meetings with the TFG supervisor and other department colleagues. These sessions were held in a controlled environment (laboratory) and simulated real operational scenarios, including the robust presentation delivered during the South Summit event [19]. For these tests, a telemetry simulator based on previously recorded real data was used, providing a near-real usage context without needing to connect a physical drone.

The final stage involved more intensive testing using complete simulations of real missions, which made it possible to assess the system's stability, performance, and usability in extended sessions, typically around 30 minutes long. These tests were carried out both in the university lab and in my personal workspace, using the project's main hardware setup (Intel i9-14980HX laptop with an RTX 4070 Studio GPU).

5.2 Participants and Profiles

Tests were mainly conducted by the project supervisor and department colleagues, representing a variety of user profiles in terms of VR experience and drone piloting skills. Most users were VR novices; an essential factor for evaluating ease of use, learning curve, and the effectiveness of anti-*motion sickness* measures. Some participants, however, had prior experience operating drones, which allowed for accurate feedback on the intuitiveness and effectiveness of the HUD design and alert systems.

5.3 Validation Criteria

Several criteria were established to consider a system iteration or version valid:

- **Absence of critical errors:** The system had to be free from bugs that significantly impacted user experience or application performance.
- **Intuitiveness and ease of use:** A quick learning curve was required, measured by the user's ability to master basic controls without external assistance.
- **Graphics performance:** A stable framerate of at least 50 fps was required, although over 90 fps was consistently achieved on the available hardware.
- **Acceptable latency:** System responsiveness had to be kept as low as possible without compromising drone movement stability and visual smoothness; small controlled delays were acceptable if they improved the viewing experience.

- **Low incidence of *motion sickness*:** Sessions had to last up to 30 minutes with minimal symptoms of dizziness or discomfort.

5.4 Results Obtained

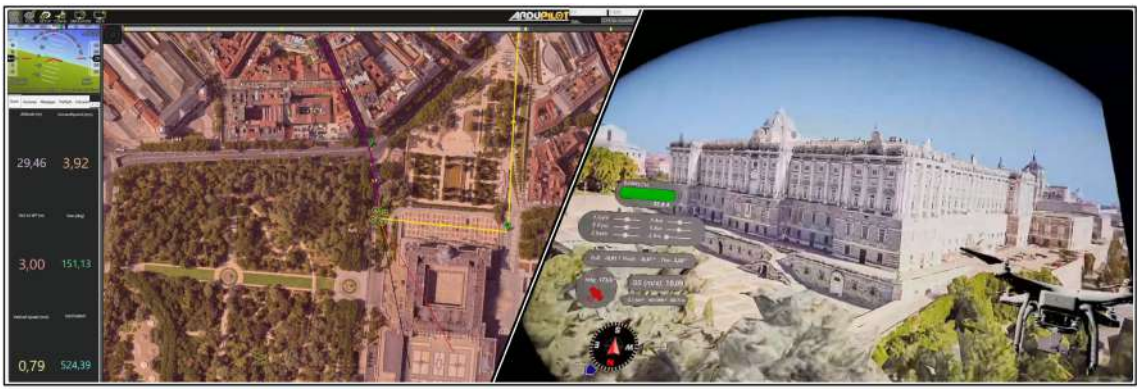

Figure 5.1: Direct comparison between Mission Planner and UAVisualizer_VR

The tests carried out confirmed that the system broadly meets the established goals and requirements. The interface proved to be highly intuitive, allowing even users without prior experience to quickly adapt to the virtual environment and effectively master the basic functions. The system's stability during extended sessions was remarkable, maintaining high and consistent framerates.

In terms of usability, suggestions provided by users during testing led to significant improvements, such as repositioning widgets to enhance visibility and accessibility, increasing the resolution of graphical elements to avoid visual degradation, and enhancing the clarity and presentation of pop-up notifications.

To illustrate and demonstrate the system's final result, a full test session was recorded during the final stages of development using a simulated mission over Plaza de España in Madrid. This is the same scenario presented initially at the South Summit event [19], clearly showcasing the significant progress achieved since that early demonstration. The full video of this test is available at the following link.

Figure 5.2: Small sample of the UAVisualizer_VR application running

5.5 Identified Improvements and Fixes

Testing allowed the identification of several areas for improvement, which were addressed during the development process:

- Increased resolution of interactive widgets to prevent loss of visual quality.
- Repositioning and redistribution of floating windows based on user feedback.
- Adjustment of the range of rays in the collision detection system and the generation of additional spheres to improve the reliability and accuracy of alerts.
- Ongoing improvements to *motion sickness* prevention techniques, optimizing elements such as *vision tunneling* and camera transition smoothness.

5.6 Identified Limitations

Although the developed system has reached a satisfactory functional level across its main components, it is important to recognize and document the limitations observed during the development, integration, and validation processes. These limitations do not detract from the value of the prototype but do highlight potential areas for future improvement.

Technical Limitations

Firstly, although the application was tested with a complete set of functionalities, these tests were conducted using simulated telemetry, as it was not possible to establish a direct connection with a real drone. This means that while the system has been validated under representative conditions, its behavior with live data in real flight conditions remains to be tested.

Regarding the collision prevention system, tests revealed that although the radial coverage using rays and spheres was effective in most situations, limitations remain in edge cases. For example, extending the rays to improve frontal collision anticipation introduced gaps between them that could create blind spots. While this was mitigated with additional spheres, full accuracy is not guaranteed.

In terms of performance, although the system consistently exceeds 90 FPS on high-end headsets (such as Quest Pro connected via Link to a laptop with an RTX 4070 Studio), no extensive performance tests were conducted on fully standalone headsets. This currently limits validation in environments with lower graphical capacity.

Validation Limitations

All system testing was performed manually, and no automated testing framework was implemented due to the complexity of development in Blueprints and the number of classes involved. An iterative approach was followed, based on unit testing per functionality (slices), and partial validations during each monthly meeting with the thesis supervisor and departmental colleagues.

Although qualitative results were positive, the group of evaluating users was limited to an academic setting. All participants were novices in virtual reality, which is useful for testing usability and effects such as *motion sickness*, but professional drone operators were not included as representative end users. Furthermore, no systematic evaluation of physiological comfort was carried out, such as quantitatively measuring fatigue or dizziness levels after extended sessions, due to the inherent difficulty of such classification.

Additionally, although a complete mission was simulated using recorded teleme-

try (e.g., over Plaza de España in Madrid), the analysis focused on subjective observations, without collecting objective metrics such as measured latency, GPU usage, or error rates in collision detection. Nonetheless, this test was recorded on video and is presented as visual support to validate the system's maturity.

These limitations should not be interpreted as shortcomings but rather as opportunities for improvement. Their systematic identification enables the definition of a realistic roadmap towards an increasingly solid, robust, and professional product.

Chapter 6

Conclusion

Throughout the development of this Bachelor's Thesis, the vast majority of the functionalities outlined in the different slices were successfully implemented. Initially, an effective migration was achieved from a traditional desktop application to a virtual reality (VR) environment, utilizing Cesium to create an immersive environment featuring an interactive pawn equipped with basic locomotion such as teleportation, rotation, and direct interaction capabilities with widgets.

Subsequently, an intuitive interaction mode via a hand-based menu was developed, consolidating it into a modular solution that facilitates scalability and maintenance. Similarly, other functionalities envisaged in the modular design were successfully completed, such as an advanced visualization and monitoring system utilizing floating windows for telemetry, interactive charts, and an efficient notification system for critical event tracking.

The only initial objective not completed within the development period was the implementation of voice command control powered by artificial intelligence. Although extensively researched and prepared for integration, it was not possible to implement this functionality due to time constraints and the need for subsequent validation and fine-tuning phases.

The delivered functionalities have been successfully tested using simulated realistic flight data, resulting in smooth, stable, and consistent performance. Although no tests were performed in real conditions with physical drones, the simulated environment provided sufficiently representative results to validate the robustness of the system in realistic operational scenarios. The success of these tests

highlights the effectiveness of the modular design and the system's adaptability to diverse operational scenarios, thereby demonstrating the feasibility of the proposed concept.

6.1 Differential Value of the Project

This project stands out in several key aspects compared to existing market solutions. Primarily, it offers innovative immersive visualization through virtual reality, significantly improving the pilot's spatial perception and situational awareness compared to traditional solutions based on 2D visualization or limited cameras. The created VR experience allows pilots to respond more intuitively and quickly to complex situations, minimizing operational errors and enhancing flight safety.

Another distinguishing aspect is the intelligent preprocessing of telemetry data, allowing clearer and more intuitive visualization of critical information. Compared to commercial solutions like Mission Planner or QGroundControl, which usually display raw data with less user-friendly interfaces, this system presents visually processed information through bars, sliders, and graphs, greatly facilitating interpretation and reaction by the pilot. This approach reduces the cognitive load on operators and enables faster, more informed operational decisions.

Additionally, the integration of real-time historical charts provides valuable analytical insights, typically absent from many commercial alternatives that only show instantaneous data. This capability for visualizing historical trends facilitates early detection of anomalies or subtle changes in drone performance.

Moreover, the implementation of a three-dimensional visualization of drone flight, independent from physical cameras, significantly reduces operational costs and provides greater flexibility in complex scenarios. This system allows pilot training in simulated environments before real missions, improving training effectiveness and reducing risks during critical operations.

The proposed solution also integrates advanced software-only technologies, such as collision prevention systems through virtual sensor simulation and visual alerts, eliminating the need for additional hardware on the drone, representing significant cost and weight savings. This solution is particularly valuable in small

and medium-sized drones, where adding physical sensors could significantly limit their autonomy and payload capacity.

6.2 Future Work Directions

The present project opens various avenues for future development, both to improve existing functionalities and to explore new features that significantly increase the practical and operational value of the application.

Firstly, a priority functionality for future iterations would be the complete implementation of voice command control through artificial intelligence (AI). This technology significantly enhances the accessibility and ergonomics of the system, allowing the operator to control the virtual environment without manual interaction. The integration of advanced language models could recognize complex commands, facilitating a more natural and efficient management of flight and internal systems.

Another immediate AI application to address is automated telemetry data monitoring. An intelligent system could perform predictive analysis and proactively issue reports or alerts in response to risky situations or anomalies detected in flight data, potentially preventing hazardous scenarios and optimizing real-time operations.

Security of message flow via WebSockets also represents a crucial aspect to strengthen in future versions. Incorporating robust encryption techniques, such as TLS (Transport Layer Security), along with authentication and authorization mechanisms, would ensure the confidentiality and integrity of transmitted information, essential in sensitive and professional operations.

Additionally, the simultaneous integration and visualization of multiple drones represents a significant improvement to considerably expand the operational utility of the system. The ability to manage multiple telemetry streams would allow the operator to supervise and coordinate joint or collaborative flights, particularly in missions requiring precise formations or spatial coordination. Direct integration with APIs from air traffic control systems or flight planning platforms could significantly enhance airspace safety management, visually representing other active

drones nearby in the virtual environment.

From the interaction perspective, advanced haptic technologies could be explored to provide tactile feedback complementing the visual experience, significantly increasing immersion and enhancing precision in delicate operations such as approaches or landings. These technologies would also help reduce the learning curve for novice pilots, considerably improving professional training.

On the other hand, technical optimization of the system to run directly on autonomous HMD devices is an urgent need. Such optimization would allow for more agile and portable operation, removing dependence on external computers and providing complete freedom of movement for the pilot. Reducing latency and maximizing graphical performance in these standalone environments would represent a critical advancement towards practical adoption in real professional contexts.

Finally, the application could greatly benefit from increased customization and configuration capabilities by the end-user. Allowing advanced settings adapted to different drone types and missions, as well as deep customization of the interface and environment, would be key to ensuring wider and more effective adoption across various industrial and academic sectors.

These future work directions offer a promising development horizon, reinforcing the current system's potential and opening doors to new applications that could radically transform the field of UAV piloting and remote supervision.

References

[1] Freepik. “Drone flying against blue sky background [premium image].” (2025), [Online]. Available: https://www.freepik.es/fotos-premium/drone-volando-sobre-fondo-cielo-azul_7682427.htm.

[2] A. Perlman. “Bvlos: The future of commercial drone operations.” (2024), [Online]. Available: https://uavcoach.com/inside-bvlos/.

[3] Fly Eye. “What is bvlos (beyond visual line of sight) & how does it work?” (2025), [Online]. Available: https://www.flyeye.io/drone-acronym-bvlos/.

[4] A. Hobbs, “Human factor challenges of remotely piloted aircraft,” NASA Ames Research Center, Tech. Rep., 2014. [Online]. Available: https://ntrs.nasa.gov/api/citations/20190001770/downloads/20190001770.pdf.

[5] DJI, *Dji o3 air unit and goggles 2: Digital fpv system*, 2022. [Online]. Available: https://www.dji.com/o3-air-unit.

[6] J. Benjak, D. Hofman, J. Knezović, and M. Žagar, “Performance comparison of h.264 and h.265 encoders in a 4k fpv drone piloting system,” *Applied Sciences*, vol. 12, no. 13, p. 6386, 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/13/6386.

[7] A. Stornig, A. Fakhreddine, H. Hellwagner, and P. Popovski, “Video quality and latency for uav teleoperation over lte: A study with ns-3,” in *Proceedings of the 2021 IEEE 93rd Vehicular Technology Conference (VTC2021-Spring)*, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9448676.

[8] Y.-M. Kwon, J. Yu, B.-M. Cho, Y. Eun, and K.-J. Park, "Empirical analysis of mavlink protocol vulnerability for unmanned aerial vehicles," *IEEE Access*, vol. 6, pp. 43 371–43 385, 2018. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2863237.

[9] C. V. Dolph, M. J. Logan, L. J. Glaab, *et al.*, "Sense and avoid for small unmanned aircraft systems," in *AIAA SciTech Forum*, Grapevine, TX, 2017. [Online]. Available: https://ntrs.nasa.gov/api/citations/20180003473/downloads/20180003473.pdf.

[10] J. da Silva Arantes. "Mission planner interface." (2017), [Online]. Available: https://commons.wikimedia.org/wiki/File:Mission_Planner.png.

[11] Cesium GS, Inc. "Learning center – cesium." (2025), [Online]. Available: https://cesium.com/learn/.

[12] Qualcomm Technologies, Inc. "Snapdragon xr2+ transforms vr experiences on meta quest pro." (2022), [Online]. Available: https://www.qualcomm.com/news/releases/2022/10/snapdragon-xr2--transforms-vr-experiences-on--meta-quest-pro.

[13] Meta Platforms, Inc. "Meta quest pro – premium mixed reality headset." (2025), [Online]. Available: https://www.meta.com/es/quest/quest-pro/.

[14] Epic Games, Inc. "Nanite virtualized geometry." (2024), [Online]. Available: https://docs.unrealengine.com/5.0/en-US/nanite-in-unreal-engine/.

[15] Unity Technologies. "High definition render pipeline (hdrp) documentation." (2025), [Online]. Available: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@latest.

[16] Blender Foundation. "Blender – a 3d creation suite." (2025), [Online]. Available: https://www.blender.org/.

[17] Cesium GS, Inc. "Tiling point clouds with cesium ion." (2025), [Online]. Available: https://cesium.com/learn/3d-tiling/ion-tile-point-clouds/.

[18] Meta Horizon OS Developers. “Meta xr for unreal engine documentation.” (2025), [Online]. Available: https://developers.meta.com/horizon/documentation/unreal/.

[19] Gacetín Madrid. “South summit madrid takes to the streets with plaza de españa as epicenter (june 1–7).” (2024), [Online]. Available: https://gacetinmadrid.com/2024/05/30/south-summit-madrid-sale-a-la-calle-con-la-plaza-de-espana-como-epicentro-1-al-7-de-junio/.

[20] M. Jovanović. “Vertical slice architecture.” (2023), [Online]. Available: https://www.milanjovanovic.tech/blog/vertical-slice-architecture.

[21] Cesium GS, Inc. “Vr series for geospatial apps – cesium for unreal.” (2025), [Online]. Available: https://cesium.com/learn/unreal/vr-introduction/.