



Wave Propagation in Augmented Reality

Tom Thys

`tom.thys.1863@student.lu.se`

Wannes Paesschesoone

`wannes.paesschesoone.5160@student.lu.se`

Department of Electrical and Information Technology
Lund University

Supervisor: Anders J. Johansson

Co-supervisor: Sara Willhammar

Examiner: Buon Kiong Lau

June 25, 2026

Contributions

This thesis is the joint work of Wannes Paesschesoone and Tom Thys. All design decisions were taken together.

Wannes Paesschesoone developed the backend (Part I): the preprocessing pipeline, the ray classifier, and the User Datagram Protocol (UDP) and Hypertext Transfer Protocol (HTTP) communication layer, with Tom contributing to the HTTP calibration handshake. On the Unreal Engine side, Wannes built the first Niagara-based visualization of the synthetic Comma-Separated Values (CSV) data (Section 3.1), the initial UDP receiver, and the Actor representing the moving User Equipment in the rendered scene.

Tom Thys developed the Augmented Reality (AR) application (Part II): all Meta Quest 3 integration, including scene mesh capture and headset-side coordinate mapping, and all the Unreal Engine implementation covering the frame buffer and playback system, the ray rendering architecture, and the user interface.

Abstract

Wireless communication systems like Wi-Fi and 5G rely on radio waves whose behavior is difficult to interpret in indoor environments without spatial context. Raw channel measurements processed by parameter estimation algorithms like Space-Alternating Generalized Expectation-Maximization (SAGE) extract multipath components described by their angle of arrival, propagation delay, and attenuation. However, numerical tables fail to convey the intuitive spatial context of the measured data within the physical environment. While existing tools mostly rely on simulated data, to our knowledge, no system currently combines empirical multipath component data with physical room recordings for immersive Augmented Reality (AR) visualization.

This thesis builds an end-to-end pipeline to address this gap. A Light Detection and Ranging (LiDAR) scan of the room is reconstructed into a 3D mesh. Multipath components are cast against this mesh and geometrically classified as line-of-sight, single-bounce, multi-bounce, or out-of-bounds. These classified frames stream over User Datagram Protocol (UDP) to a Meta Quest 3 headset, where an Unreal Engine 5 application renders the rays in their correct world positions. A dedicated Hypertext Transfer Protocol (HTTP) service handles the initial setup by aligning the LiDAR scan with the headset's scene mesh.

The system successfully transforms empirical measurements into immersive 3D visualizations. Operating within the headset's performance budget, it achieves a reasonable spatial placement accuracy. Ultimately, this provides researchers with a visually plausible, intuitive tool to analyze wireless channels directly within the measured environment.

Popular Science Summary

Seeing the invisible: Visualizing radio waves in Augmented Reality

Wireless engineers usually study radio waves as tables of numbers. We built an Augmented Reality tool that lets them see the waves instead, drawn as 3D paths through the actual room, viewed through a headset.

Wi-Fi and 5G keep us connected, but the radio waves that carry our data are invisible. When engineers measure how these signals travel through an indoor environment, the result is a table of numbers. Knowing that a signal bounced at 30 degrees after traveling 12 meters tells you little about what happened in the room itself.

Our goal was to bridge that gap by turning the static numbers into a 3D scene a researcher can step inside and explore. The pipeline takes a 3D laser Light Detection and Ranging (LiDAR) scan of a real room and combines it with the recorded signal data. A computer reconstructs the path each radio wave took, labeling it as line-of-sight, single-bounce, multi-bounce, or out-of-bounds (when no intersection was found). The reconstructed paths are then streamed wirelessly to a Meta Quest 3 headset, which overlays them onto the user's view of the room.

The hardest part was spatial alignment: making a ray that bounced off a real wall look like it is hitting that same wall in the headset. Our first version asked the user to calibrate the room by placing their controllers in the corners, but small placement errors caused the virtual rays to misalign. We replaced this manual step with an automatic one. The headset builds its own 3D mesh of the room in real time, and our software aligns the LiDAR scan with the headset's scene mesh to snap the two coordinate systems together and calculate a transformation matrix. Placement errors dropped to one to five centimeters.

The result is a tool that replaces numerical tables with spatial intuition. Researchers can walk through a measured environment and see exactly where each

signal went and what it bounced off, giving them a clearer foundation for designing the next generation of wireless signals.

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Anders J. Johansson, for his continuous guidance, valuable feedback, and support throughout this thesis. We are especially thankful to him for lending us his personal Meta Quest 3 headset, which made the practical development and testing of this project possible.

We also wish to thank our co-supervisor, Sara Willhammar, for her support, encouragement, and constructive discussions during the development of the project.

Furthermore, we would like to thank Liesbet Van der Perre from KU Leuven as our contact person at KU Leuven, she connected us with Anders and followed up on the progress of our work.

We are especially grateful to Yingjie Xu for providing the measurement datasets used in this thesis. The availability of this data, originating from his prior research [1], [2], made the development and evaluation of our system possible.

Finally, we would like to thank our colleagues in the Communications (COM) division at Lund University. We deeply appreciated the COM-division meetings, the shared fika breaks, and the division events that allowed us not only to learn from the work of others but also to share our own progress in such a welcoming and stimulating environment.

Declaration

We hereby affirm that this Master thesis was composed by ourselves, that the work herein is our own except where explicitly stated otherwise in the text. This work has not been submitted for any other degree or professional qualification except as specified, nor has it been published. Where other people's work has been used (either from a printed source, internet, or any other source), this has been carefully acknowledged and referenced.

During the preparation of this thesis, we used AI tools such as Gemini and Claude Code to assist in the writing and development process. These tools were used to improve language, flow, readability, and to support parts of the software implementation and code development. After using this tool/service, we have reviewed and edited the content as needed. We take full responsibility for the content of the whole thesis.

Table of Contents

Acknowledgements	vii
Declaration	ix
List of Acronyms	xviii
1 Introduction	1
1.1 Background	1
1.2 Problem Statement	2
1.3 Aim	2
1.4 Scope and Limitations	2
1.5 Contributions	3
1.6 Previous Work	3
1.7 Thesis Structure	4
2 Methodology	5
2.1 Hardware	5
2.2 Measurement Data	8
2.3 Design Goals	8
2.4 Pipeline Stages	10
I Backend Development	13
3 First Iteration and Design Choices	15
3.1 Synthetic Data and Early Visualization	15
3.2 Getting Data to the Headset	17
3.3 From Python to Rust	18
4 Preprocessing	21
4.1 Reading in the Measurement Data	21
4.2 Building the Environment Mesh	23
4.3 Aligning the Mesh to the Headset Frame	29

5	Ray Computation	35
5.1	Per-Ray Pipeline	35
5.2	Validation Visualizer	40
6	Communication	43
6.1	Transport and Encoding	43
6.2	Wire Format	44
6.3	Backend Server	46
6.4	Diagnostic Sender and Receiver	48
6.5	Persisted Alignment Transform	49
II	AR Development	51
7	Game Engine Selection	53
7.1	Architecture and Memory Management	53
7.2	Performance	54
7.3	Development Workflow	55
7.4	XR and Meta Quest Integration	55
7.5	Conclusion	57
8	Coordinate Mapping	59
8.1	Background: Spatial Data on Meta Quest 3	59
8.2	Approach 1: Manual Spatial Anchors	60
8.3	Approach 2: Scene Mesh Alignment	62
8.4	Summary	65
9	Signal Data Reception and Buffering	67
9.1	Wire Protocol	67
9.2	UDP Reception	67
9.3	Frame Buffer and Memory Limitations	68
9.4	Playback Modes	68
10	Ray Rendering Architecture	71
10.1	Evaluation of Rendering Approaches	71
10.2	Implementation	74
10.3	Summary	78
11	User Interface and Interaction	79
11.1	HUD Architecture	79
11.2	Configuration Controls	79
11.3	Playback Controls	80
11.4	Laser Pointer and Interaction	81
11.5	Communication Pattern	81
12	Hardware Platform Retrospective	83
12.1	Apple Vision Pro	83
12.2	HTC VIVE Focus	84

12.3	Pico 4 Ultra	84
12.4	Summary	84
III	Evaluation and Conclusion	86
13	Results and Discussion	87
13.1	Streaming and Classification Throughput	87
13.2	On-Device Rendering and Spatial Alignment	88
13.3	Visual Plausibility and Geometric Constraints	89
14	Future Work	91
14.1	Live Data Integration	91
14.2	Network Protocol	91
14.3	Engine and Hardware Expansion	92
14.4	Environment and Geometric Processing	92
14.5	Interaction and Parameter Tuning	92
15	Conclusion	93
15.1	Summary of Findings	93
15.2	Broader Implications	94
15.3	Limitations	94
A	Classification Throughput Benchmark Data	107

List of Figures

2.1	The mobile User Equipment (UE). Photographed by the authors.	6
2.2	The Multiple-Input Multiple-Output (MIMO) antenna panel used in the measurement campaign. Photographed by the authors.	6
2.3	The Meta Quest 3 VR headset. Photographed by the authors.	7
2.4	System architecture: Python pre-processing pipeline, Rust backend, and Meta Quest 3 client.	12
3.1	Local 3D viewer (Matplotlib) used to verify the synthetic rays. Line-of-sight rays are shown in green, single-bounce rays in orange, and the transmitter in red.	16
3.2	First in-engine visualization of the synthetic rays using Unreal Engine's Niagara particle system. The light blue beams represent the MPCs.	17
3.3	Backend cost of the PyVista prototype on a single snapshot of 159 rays, with rendering excluded. The red segment is the avoidable per-call index rebuild discussed in the text.	20
4.1	Local coordinate system of an antenna panel viewed from the front face. The y-axis points outward toward the user equipment (out of the page). Photographed by the authors.	24
4.2	Point cloud from dataset 2 before filtering.	25
4.3	Point cloud from dataset 2 after filtering.	26
4.4	Mesh reconstructed by Poisson surface reconstruction.	27
4.5	Mesh smoothing by planar segmentation and projection.	29
4.6	Spatial registration of the LiDAR and headset meshes. The LiDAR mesh is shown in blue, the headset mesh in orange, and the matched walls that drive the coarse alignment are highlighted in green. The final transform T maps the LiDAR mesh onto the headset mesh, so the two coincide after applying T to the LiDAR vertices.	30
5.1	Visualization of the math behind the ray classification.	39

5.2	Desktop visualizer used to validate the ray classification. The mesh is rendered semi-transparent, so rays inside the room remain visible. Rays are colored by type (line-of-sight, single bounce, multi bounce, out of bounds), the antennas and the UE are drawn as wireframe markers. The side panel exposes playback, filters, the five classification tolerances, and an optional ground plane.	40
8.1	Manual spatial anchor placement in the room.	60
8.2	Scene mesh and axis frame export.	63
10.1	Thread-safe double-buffering architecture showing data transfer across the render command boundary.	76
11.1	The in-headset settings panel.	80
13.1	Propagation rays overlaid on the measurement room as seen through the Meta Quest 3.	90

List of Tables

2.1	Key specifications of the Meta Quest 3. [10]	8
2.2	Structure of the first measurement dataset.	9
2.3	Structure of the second measurement dataset.	10
3.1	Structure and parameter descriptions of the synthetic CSV dataset. .	17
6.1	Wire header structure. Transmitted once per datagram.	44
6.2	Wire ray entry structure. Appended <code>num_rays</code> times following the header.	45
8.1	Comparison of coordinate mapping approaches.	66
10.1	Comparison of rendering approaches for ray visualization.	75
12.1	Cross-platform compatibility assessment.	85
A.1	Speedup of Rust backend relative to Python baselines for full 780-timestamp dataset.	109

List of Acronyms

AoA	Angle of Arrival
AoD	Angle of Departure
API	Application Programming Interface
AR	Augmented Reality
ASCII	American Standard Code for Information Interchange
BVH	Bounding Volume Hierarchy
CSV	Comma-Separated Values
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
D-MIMO	Distributed Multiple-Input Multiple-Output
GC	Garbage Collector
GIL	Global Interpreter Lock
HDF5	Hierarchical Data Format 5
HTTP	Hypertext Transfer Protocol
HUD	Head-up Display
ICP	Iterative Closest Point
IL2CPP	Intermediate Language to C++
IoT	Internet of Things
IPC	Inter-Process Communication
ITU-R	International Telecommunication Union - Radiocommunication Sector

JSON JavaScript Object Notation

LiDAR Light Detection and Ranging

LoS Line of Sight

MIMO Multiple-Input Multiple-Output

MPC Multipath Component

MR Mixed Reality

MRUK Meta Mixed Reality Utility Kit

PCA Principal Component Analysis

Protobuf Protocol Buffers

QBVH Quaternary Bounding Volume Hierarchy

RANSAC Random Sample Consensus

RSSI Received Signal Strength Indicator

SAGE Space-Alternating Generalized Expectation-Maximization

SoC System on Chip

SVD Singular Value Decomposition

TCP Transmission Control Protocol

UDP User Datagram Protocol

UE User Equipment

UE5 Unreal Engine 5

UI User Interface

VR Virtual Reality

VTK Visualization Toolkit

XR Extended Reality

Introduction

Wireless communication underpins much of modern infrastructure, and understanding how radio waves propagate in indoor environments is essential for research on optimizing systems such as Wi-Fi and 5G/6G. Channel measurements enable the extraction of per-path parameters of individual Multipath Components (MPCs) such as Angle of Arrival (AoA), propagation delay (equivalently, path length) and attenuation, using parameter estimation algorithms such as Space-Alternating Generalized Expectation-Maximization (SAGE) [3]. However, raw numerical data often lacks intuitive spatial context. To make these signals intuitive, this thesis develops an Augmented Reality (AR) application that allows users to physically navigate a room while visualizing measurement-derived multipath components.

1.1 Background

Indoor wave propagation is studied through two complementary approaches: simulations using physics-based ray-tracing and empirical measurements. Simulations are repeatable and parameterizable, but they depend on accurate geometry and material models that are difficult to obtain for real rooms. Measurements capture the actual propagation in a specific room at a specific time, but the output is a numerical table of Multipath Component (MPC) parameters that says little about the geometry that produced it. The gap between measurement output and physical intuition is a recurring obstacle in indoor channel research: a propagation delay of 12 m and an azimuth of 30 degrees tell the analyst little without a mental model of where those bounces actually happened.

Mixed Reality (MR) headsets are an ideal platform for closing this gap. A standalone MR headset can render the rays in their world-space positions while the user walks through the same physical room in which the measurements were taken. Path geometry, antenna panel placement, and obstructing furniture appear together in the same spatial frame, and the user's own viewpoint determines what is shown. Recent consumer hardware such as Meta Quest 3 makes this kind of visualization possible without the tethered workstations or external trackers that

earlier Extended Reality (XR)-based work depended on. The precise distinction between AR, MR, and XR is elaborated in Section 2.1.2.

1.2 Problem Statement

Despite recent advances in visualizing wireless channels, a gap remains. Most existing tools focus on deterministic simulation platforms rather than empirical measurements, and those that do consume empirical data tend to aggregate it into 2D signal-strength views rather than expose individual paths in 3D. This thesis closes that gap by combining purely empirical multipath data with a Light Detection and Ranging (LiDAR)-reconstructed environment and visualizing individual MPC parameters as 3D rays inside the measured room, viewed through a standalone MR headset. To our knowledge, no existing work integrates all three aspects into a single pipeline.

1.3 Aim

This thesis aims to make wave propagation intuitive by overlaying measurement-derived multipath rays directly onto the physical environment in which they were recorded. The system targets a Meta Quest 3 as a standalone MR headset. By turning a static measurement into an interactive scene the user can walk through, the system allows researchers to intuitively map SAGE-estimated MPC parameters against the physical geometry that produced them.

Concretely, the work addresses three research questions:

1. Can measurement-derived MPCs be classified and streamed at a stable frame rate to a standalone AR application?
2. Can the resulting rays be rendered on the Meta Quest 3 in alignment with the physical room, given the device’s performance constraints?
3. Do the rays classified by the geometric ray-mesh pipeline produce visually plausible propagation paths against the measured room?

1.4 Scope and Limitations

The scope of this thesis is bounded by several hardware and data constraints. The two datasets used are indoor Distributed Multiple-Input Multiple-Output (D-MIMO) measurements, executed at Lund University. SAGE post-processing was performed by the data provider and is not a contribution of this thesis. The target hardware is the Meta Quest 3, portability to other AR and MR headsets is out of scope. The room mesh is captured at the moment of the measurements and does not dynamically adapt to changes in the environment. Finally, the visualization is qualitative: the system supports the visual inspection of propagation paths against physical geometry, but it does not support formal verification of channel models.

1.5 Contributions

This thesis makes the following contributions:

- An end-to-end pipeline that converts raw Distributed Multiple-Input Multiple-Output (D-MIMO) measurements processed by SAGE into spatially registered AR ray visualizations on a standalone headset.
- A Rust backend that classifies ray mesh intersections against a Quaternary Bounding Volume Hierarchy (QBVH)-accelerated room mesh and streams the classified frames over User Datagram Protocol (UDP), evaluated quantitatively against an earlier Python prototype (Chapter 3).
- A custom Unreal Engine 5 (UE5) rendering component that draws the streamed rays under the Meta Quest 3 performance constraints and supports per-ray selection and filtering.
- An interactive validation visualizer enabling reproducible debugging of the classification stage independently of the AR application.

1.6 Previous Work

Several research systems have addressed the gap between MPC data and physical context by rendering propagation in 3D, but most rely on synthetic or simulated data rather than empirical measurements. CupCarbon [4] is a simulation platform that predicts radio propagation in Internet of Things (IoT) networks using ray-tracing-based channel models over digital 3D city data, estimating signal behavior and interference before a network is actually deployed. Tang et al. [5] bring a simulated pipeline into an immersive endpoint. They extract 3D map information from Sketchup and simulate radio propagation through Matlab-based ray-tracing and Maxwell's equations. The resulting path parameters are then integrated into Unreal Engine 4, allowing users wearing an HTC VIVE to inspect channel properties and multipath effects within a virtual office. Koutitas et al. [6] move the display from Virtual Reality (VR) to AR, but their signals are still simulated. They reconstruct a vector facet model of the room from 2D smartphone images or AR spatial scans, and run a ray-tracing algorithm over it to predict signal strength. The predicted signal strength is then overlaid on the physical environment as a planning aid rather than as a window onto recorded measurements.

Wang et al. [7] push closer to the empirical side by combining a LiDAR-scanned 3D environment with GPU-accelerated ray tracing. They assign International Telecommunication Union - Radiocommunication Sector (ITU-R) material parameters to the scanned objects and validate the simulated output against onsite Received Signal Strength Indicator (RSSI) measurements. The room is real, but the signal is still simulated. ESPARGOS [8] is the only measurement-driven system in this group, but it overlays real-time channel data as a 2D heatmap on a camera view, without exposing individual paths as inspectable 3D rays.

The measurement data this thesis builds on comes from the indoor distributed-Multiple-Input Multiple-Output (MIMO) campaign at Lund University by Xu et

al. [1], [2], [9]. Their work provides the per-MPC angles, delays, and attenuations that feed the visualization pipeline, recorded against a real room with antenna panels in known positions. Reusing this dataset means the rays this thesis renders are tied to actual hardware in a real environment rather than to a simulated scenario.

1.7 Thesis Structure

Following the methodology and system architecture presented in Chapter 2, the remainder of this thesis is divided into three main parts. Part I details the backend data pipeline, moving from the preprocessing of SAGE measurements and LiDAR meshes to the core ray computation and UDP streaming protocol. Part II focuses on the AR application for the Meta Quest 3, explaining the Unreal Engine 5 rendering architecture, spatial coordinate mapping, network buffering, user interface, and cross-platform hardware portability. Finally, Part III presents the overall evaluation of the visualization system, a discussion of the results, directions for future work and a conclusion.

Methodology

This chapter describes the methodology and overall architecture of the system: the pipeline that turns recorded wave-propagation measurements into an augmented-reality visualization on the Meta Quest 3. First we introduce the hardware needed for our pipeline, then we describe the measurement data format. Then the design goals that shaped the architecture are laid out, and finally the three stages of the pipeline are described at a high level.

2.1 Hardware

2.1.1 Measurement Hardware

The User Equipment (UE) is the mobile transmitter of the measurement campaign. It consists of an antenna, mounted together with a LiDAR sensor on a small wheeled robot (Figure 2.1). During the measurements the robot drives slowly across the room. The LiDAR sensor on the robot captures the point cloud of the environment.

The D-MIMO antenna panels (Figure 2.2) are the receivers of the measurement campaign. Each panel has a known position in the room. The antennas on the panel capture the incoming radio waves, and the SAGE algorithm is used to estimate the MPCs. Each MPC is described by an AoA, a propagation distance, and an attenuation. The AoA is given in the local coordinate system of the panel, so the position and orientation of the panel are needed to transform it into the global room coordinates.

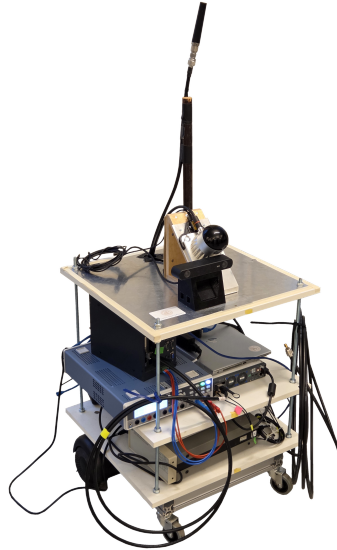


Figure 2.1: The mobile UE. Photographed by the authors.

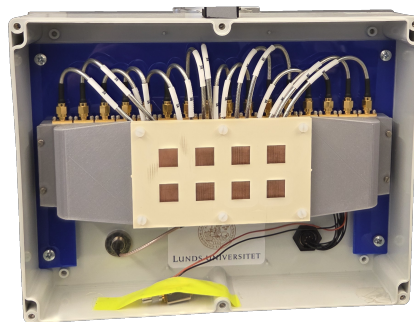


Figure 2.2: The Multiple-Input Multiple-Output (MIMO) antenna panel used in the measurement campaign. Photographed by the authors.



Figure 2.3: The Meta Quest 3 VR headset. Photographed by the authors.

2.1.2 Meta Quest 3

In this thesis the Meta Quest 3 (Figure 2.3) is used as the target AR platform for the visualization [10]. It is a standalone device: instead of rendering on a tethered workstation and streaming frames to a display, it runs the UE5 application directly on its own mobile system. This allows a researcher to walk freely through the measurement room with no cabling.

Although marketed primarily as a VR headset, the Quest 3 supports MR pass-through: two forward-facing RGB cameras, together with a depth sensor, reconstruct a view of the physical surroundings onto which virtual content is overlaid. This is the mechanism by which the classified rays are drawn over the user’s actual view of the room. The same sensors drive Meta’s Scene Application Programming Interface (API), which exposes a mesh and a set of anchor points of the room.

This video passthrough mode means the Quest 3 is technically a MR device rather than a traditional optical AR headset, in which the user views the physical world directly through transparent lenses while digital content is overlaid. XR is the umbrella term spanning VR at one extreme (fully synthetic environments), optical AR at the other, and MR in between. The rendering pipeline and spatial-registration methodology of this thesis are equivalent for both optical AR and video-passthrough MR.

Internally the headset is built around a Qualcomm Snapdragon XR2 Gen 2 System on Chip (SoC) with 8 GB of RAM, driving two displays of 2064×2208 pixels each at refresh rates up to 120 Hz over a $110^\circ \times 96^\circ$ field of view. These specifications define the performance budget the on-device renderer must respect to run smoothly. Key specifications are summarized in Table 2.1.

Table 2.1: Key specifications of the Meta Quest 3. [10]

Component	Specification
System on Chip (SoC)	Qualcomm Snapdragon XR2 Gen 2
Memory	8 GB RAM
Storage	512 GB
Display	2064×2208 px per eye
Max. Refresh rate	120 Hz
Field of view	110° (H) $\times$ 96° (V)
Optics	Pancake lenses, IPD 53-75 mm
Pass-through	2 RGB cameras + depth projector
Connectivity	Wi-Fi 6E, USB-C
Weight	515 g
Eye tracking	Not featured

2.2 Measurement Data

The input datasets are recordings of indoor D-MIMO measurements, organized as a sequence of timestamps. At each timestamp every receiving antenna panel reports a set of MPCs. The UE carrying the transmitter moves through the room while these measurements are taken, and its position is recorded at every timestamp.

The two datasets used in this thesis were not collected as part of this work, they were provided by Yingjie Xu, one of the authors of [1], [2]. The measurements were conducted in the E-building at the Faculty of Engineering (LTH), Lund University. The first dataset [1] (Table 2.2) contains a local coordinate system for each antenna panel, which makes it easy to transform the MPCs into the global room coordinates. The second dataset [2] (Table 2.3) does not contain this information, so pre-processing is needed to estimate the panel orientations from the MPCs themselves. In addition, the second dataset uses delay and amplitude instead of distance and attenuation. The propagation distance can be calculated from the delay by multiplying it by the speed of light. The attenuation (dB) can be calculated from the amplitude by using the formula $20 \log_{10}(\text{amplitude})$.

2.3 Design Goals

The architecture is shaped by four requirements that follow from the aims stated in Chapter 1.

Spatial registration. The visualization must remain anchored to the physical room in which the measurements were recorded. A ray drawn at a given AoA

Table 2.2: Structure of the first measurement dataset.

Variable name		Data dimension	Description
Antenna Panels	Pos	$[3 \times 8]$	3D positions of the eight D-MIMO antenna panels (xyz coordinates $\times$ panel number).
	xaxis / yaxis / zaxis	$[3 \times 8]$	Each antenna panel has its own x' - y' - z' local coordinate system. (x' , y' , z' $\times$ panel number)
UE	UE_truth	$[3 \times 780]$	3D UE trajectory: 780 positions, one per timestamp.
Channel	z_t	$[780 \times 8]$ cells, each $[4 \times P]$	SAGE-estimated MPCs per (timestamp, panel). Each cell has one column per path, the four rows are: propagation distance, azimuth AoA, elevation AoA, and channel attenuation.
Point cloud	ptL	N/A	Point cloud data of the room, captured by the LiDAR scanner.

has to leave its sensor in the direction the SAGE estimator reported, with no perceptible drift relative to the walls of the room as the user walks around. This requirement drives both the LiDAR-derived environment mesh and the start-up handshake between the backend and the headset, in which the two coordinate systems are aligned.

Standalone operation on the Meta Quest 3. The headset must render the visualization and track the room on its own hardware, so that a researcher can move freely with no cable connecting the device to a workstation. Unlike a PC-tethered setup, where a desktop GPU renders every frame and streams the finished video to the display, the Quest 3 here receives only a compact stream of ray data over wireless UDP and draws it locally. The heavy per-frame work ray classification and serialization still runs on a separate backend machine, but because it transmits lightweight data rather than rendered frames, it imposes no tether and leaves the on-device rendering budget as the binding constraint.

Decoupled backend and headset development. The two halves of the system must evolve independently: the backend testable against recorded data with no headset in the loop, and the headset able to render frames from any source that

Table 2.3: Structure of the second measurement dataset.

Variable name		Data dimension	Description
Antenna Panels	Pos	$[3 \times 8]$	3D positions of the eight D-MIMO antenna panels (xyz coordinates $\times$ panel number).
UE	UE_truth	$[3 \times 50]$	3D UE trajectory: 50 positions, one per timestamp.
Channel	Mpc_info	$[7551 \times 6]$	Flat table of all multipath components. Each of the 7,551 rows is one path, the six columns are: timestamp index, panel index, azimuth AoA, elevation AoA, delay, and amplitude.
Point cloud	pcMapDS	N/A	Point cloud data of the room, captured by the LiDAR scanner.

speaks a defined wire format. This is achieved by narrowing the interface between them to a single contract: the UDP frame of Chapter 6. The decoupling pays off twice: it lets us develop Part I and Part II in parallel, and it lets the same backend output feed an validation visualizer as well as the headset.

Forward-compatibility with live measurements. Although the measurement datasets used in this thesis are static and could in principle be pre-computed offline, the live data path is built around a per-frame loop in which each timestamp is classified, encoded, and transmitted independently. Classification carries no cross-frame state, and the streaming protocol tags every datagram with its source timestamp, so the receiver is agnostic to whether a frame originated from a recording or from a live measurement. Replacing the static-dataset loader with a real-time source therefore leaves the classification, encoding, and rendering stages untouched: the only changes required are in the data-ingestion stage and in the outer driver loop that feeds one frame at a time through the pipeline. This forward-compatibility property is revisited in the future-work (Section 14.1) discussion.

2.4 Pipeline Stages

The pipeline has three stages, illustrated in Figure 2.4.

Pre-processing (Chapter 4) is performed once per measurement campaign. A LiDAR point cloud of the room is filtered, reconstructed into a 3D mesh, and lightly

smoothed. The resulting mesh is used by every later stage as the geometric ground truth.

The backend is a Rust service that operates in two phases. At start-up it receives the Meta Quest 3 scene mesh from the headset, and returns the rigid transform that aligns the two coordinate frames. In the per-frame loop, for every frame in the recording it casts each MPC against the QBVH accelerated room mesh, classifies the resulting ray as Line of Sight (LoS), single-bounce, or multi-bounce (Chapter 5), and emits the bundle as a single UDP datagram (Chapter 6).

The Unreal Engine application on the Meta Quest 3 also has two phases. At start-up it captures its scene mesh through the Meta Scene API, registers anchor points relative to the Unreal world origin, and forwards both to the backend. At runtime it decodes the incoming datagrams and draws the classified rays overlaid on the user's view of the room (Part II).

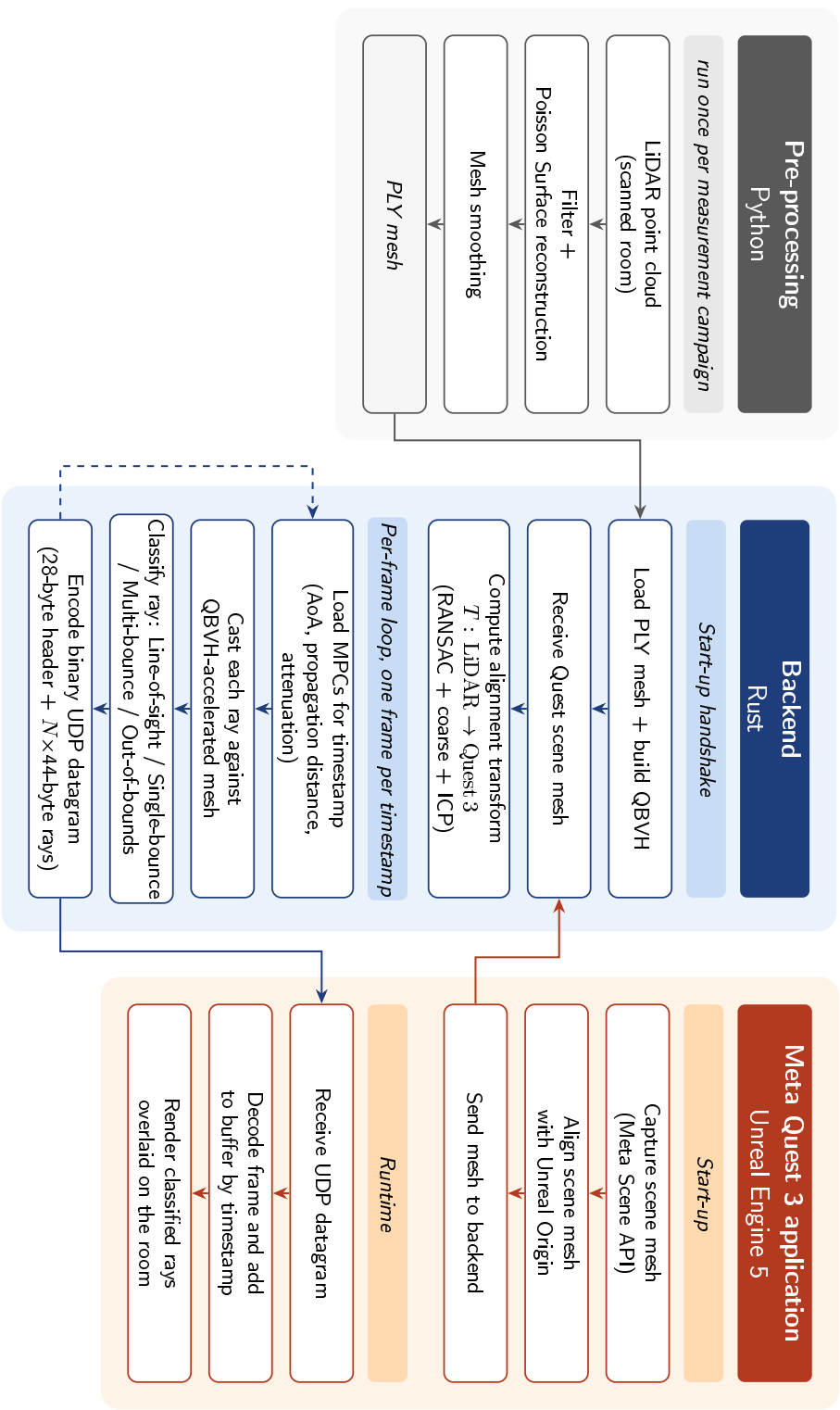


Figure 2.4: System architecture: Python pre-processing pipeline, Rust backend, and Meta Quest 3 client.

Part I

Backend Development

First Iteration and Design Choices

In this chapter a first prototype was built to visualize the rays in Unreal Engine. This prototype settled three questions: what and how to visualize the rays, how to get it to the headset, and what to compute on the backend. This first iteration was mainly built on synthetic data, this kept the focus on the pipeline itself, and produced the skeleton that later chapters will refine.

3.1 Synthetic Data and Early Visualization

For the initial phase, synthetic data was generated to simulate the rays. This allowed for rapid iteration on the visualization logic without dealing with the complexity of the real dataset. Each synthetic ray originates from a single fixed transmitter. To simplify the initial geometric generation, random Angles of Departure (AoDs) (azimuth and elevation) were used to project the paths outward toward a terminal receiver point, though the final system framework is built to ingest AoAs from the real dataset. Each ray is also described by a propagation delay, and a power level, all sampled at random within reasonable bounds. Every ray is labeled either line-of-sight or single-bounce. These spherical parameters are then converted into 3D Cartesian coordinates, with the delay scaled by the speed of light to obtain the path length (propagation distance). For a single-bounce ray an intermediate reflection point is added, this point is illustrative rather than physically derived, since the prototype only needed plausible geometry to render.

The resulting values are stored in a Comma-Separated Values (CSV) file, one row per MPC (Table 3.1). CSV was chosen because Unreal Engine imports it natively as a Data Table, which made it the most efficient pipeline for initial ingestion. Each row holds the transmitter, optional bounce, and end points in Cartesian coordinates, together with the power level and a ray-type flag. LoS rays leave the bounce columns empty. This format was never intended for the final real-time streaming, but two of its properties carried over into the later pipeline: the flat, one-row-per-ray layout and the explicit ray-type encoding both reappear in the binary wire protocol of Chapter 6.

To verify the geometry prior to integration with the game engine, a local 3D viewer

was written using Python and Matplotlib [11]. It parses the CSV and plots each ray from the transmitter to its endpoint, passing through the reflection point for single-bounce rays (Figure 3.1). This made it straightforward to confirm that the spherical-to-Cartesian conversion and the reflection geometry were correct in a controlled setting before dealing with Unreal Engine’s rendering.

The same CSV was then imported into Unreal Engine and rendered with the Niagara particle system [12], producing the first in-engine visualization (Figure 3.2). Each ray is drawn as a beam from the transmitter to its endpoint. This Niagara prototype, and the reason it was eventually abandoned, is discussed in Chapter 10.

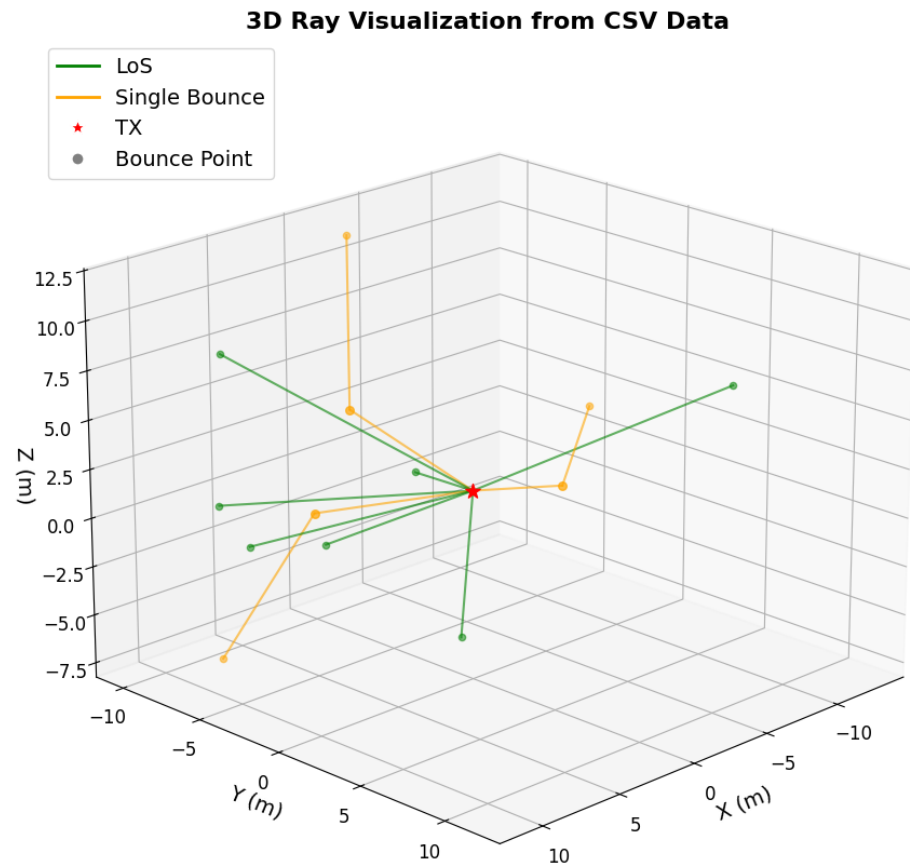


Figure 3.1: Local 3D viewer (Matplotlib) used to verify the synthetic rays. Line-of-sight rays are shown in green, single-bounce rays in orange, and the transmitter in red.



Figure 3.2: First in-engine visualization of the synthetic rays using Unreal Engine’s Niagara particle system. The light blue beams represent the MPCs.

Table 3.1: Structure and parameter descriptions of the synthetic CSV dataset.

Column Name	Description
ray_id	Unique identifier assigned to each generated ray.
tx_x, tx_y, tx_z	3D Cartesian coordinates of the transmitter location.
bounce_x, bounce_y, bounce_z	3D Cartesian coordinates of the reflection point, set to NaN for line-of-sight (LoS) rays.
end_x, end_y, end_z	3D Cartesian coordinates of the calculated endpoint.
power_dbm	Assigned transmission power level in dBm.
type	Ray classification: 0 denotes LoS and 1 denotes a single-bounce reflection.

3.2 Getting Data to the Headset

With the per-ray data structure defined, the rays still had to be transported to the renderer. In the prototype this renderer was Unreal Engine running on a desktop, while the eventual target is the Meta Quest 3. Several transport and serialization options were tried before settling on the final one, and in each step the overhead that the previous option had introduced was removed.

The first attempt used a WebSocket connection, with a small client requesting a batch of rays and the server replying with the data. This request-response model

coupled the update rate to a full round-trip and was a poor fit for a visualization that should simply receive new frames as fast as they are produced. WebSocket also runs over Transmission Control Protocol (TCP), whose reliability guarantees turn out to work against real-time streaming: a single delayed segment blocks every later one until it is retransmitted, an effect known as head-of-line blocking [13], and by the time a lost frame is recovered it is already stale, since the user only cares about the most recent set of rays. The prototype was therefore changed to push frames over a raw UDP socket. UDP sends each datagram independently and never waits for an acknowledgement, so a lost frame simply does not arrive and is replaced by the next one. This trades reliability for bounded latency, which is the right trade-off for this visualization.

With the transport fixed, the remaining architectural decision was how to encode the rays inside each datagram. A single ray fits in 44 bytes, but each frame contains numerous rays, so any per-field encoding cost multiplies quickly. Protocol Buffers (Protobuf) [14] was considered first. It defines a schema-driven wire format in which every field is preceded by a tag and wire type, and the receiver reconstructs the message through generated parsing code rather than reading the buffer directly. For a payload that is already a dense array of fixed-size records this adds overhead without buying flexibility, and the format also handles LoS rays awkwardly, since their bounce fields have no natural representation when no reflection occurred. JavaScript Object Notation (JSON) was used briefly during development for the convenience of inspecting packets in plain text, but was replaced once the format had stabilized. The final pipeline uses a fixed binary layout whose field order and alignment match the in-memory layout on the receiver, so the network buffer can be reinterpreted as a typed array of ray records without per-field parsing. The byte-level structure of that format is described in Chapter 6

It is worth being clear about what real-time means here. The current system is not real-time in the strict sense, as the backend streams from a recorded dataset rather than from a live measurement. The pipeline was nonetheless designed with real-time operation in mind. Each frame is classified and sent independently and carries a timestamp index and a sequence number in its header (Chapter 6), so the recording could later be replaced by a live source without changing the rest of the pipeline. Making the full measurement chain real-time is a goal of the wider project and is left as future work (Chapter 14).

3.3 From Python to Rust

The backend that produces these frames was first written in Python. Python was a deliberate choice for the prototype: it allowed the data parsing, the spherical-to-Cartesian conversion, the ray classification, and the UDP streaming to be assembled and changed quickly while the pipeline was still taking shape. Ray casting against the environment mesh was handled with PyVista [15], which wraps the Visualization Toolkit (VTK) and was already in use for the local 3D viewer. This kept the whole prototype, computation and visualization, inside a single familiar library.

Although the prototype was rapidly developed, its runtime performance proved to be a limiting factor. Figure 3.3 presents a profiling analysis of a single snapshot containing 159 rays. The results show a one-time initialization cost arising from library imports, mesh loading, and acceleration-structure construction. After initialization, the execution time is largely determined by the per-frame geometric processing required for ray classification, which consists of two primary operations. Casting the 159 rays against the mesh costs only 0.37s, because PyVista builds its ray-tracing acceleration structure once and reuses it for every ray. The single-bounce test, which requires the normal of the reflecting surface, costs 1.69s. Almost all of this, 1.59s, is PyVista rebuilding a spatial index from scratch. The prototype calls `find_closest_cell` once per ray, and that method builds a throw-away cell locator on each call rather than caching one on the mesh as it does for the ray-tracing tree, so the index is rebuilt for every lookup. The lookup it serves takes only 4ms, and the small remainder is the reflection-geometry arithmetic and the library call overhead around it. This repeated build is avoidable even within PyVista, either by passing every query point to a single `find_closest_cell` call, which builds the locator once, or by reusing the underlying VTK cell locator directly across calls. The dominant cost is therefore neither Python nor ray casting, but the way the prototype used a single library operation. PyVista is a visualization toolkit rather than a compute backend, and a library intended for batched ray casting, such as `trimesh` [16], would make this build-once pattern the default rather than something the caller must arrange. Python was, in short, a perfectly adequate language for this backend, and the figure should be read as the cost of the prototype as built rather than as a ceiling on what Python can do. The reasons for moving to Rust lay elsewhere.

The rewrite to Rust was therefore an engineering decision rather than a verdict on Python, and it rested on properties of the final system rather than raw speed. The strongest of these is parallelism. The classification of one ray is independent of every other, so the workload is naturally parallel, yet the per-ray work is branching control flow (a LoS test, a mesh intersection, a single-bounce check involving a surface normal and two angle comparisons, and a fallback for rays that leave the room) that does not benefit from NumPy vectorization and runs as a plain Python loop. Under CPython's Global Interpreter Lock (GIL) such Python-level work cannot run on more than one core. This is not an absolute barrier: the work can be spread across processes with multiprocessing, at the cost of Inter-Process Communication (IPC) and the awkwardness of sharing the mesh and its acceleration structure between processes. The GIL itself is being removed, with free-threaded builds experimental in Python 3.13 [17] and officially supported, though still optional, in Python 3.14. Neither route was attractive here: the free-threaded interpreter was not available in the Python 3.12 used for the prototype and the process-based route trades the GIL for serialization overhead. Rust sidesteps the question entirely: the `rayon` crate parallelizes the per-ray classification across all cores with no GIL and no dependence on an evolving interpreter, by flattening every frame into its individual rays.

Two further properties reinforced the choice. Ray casting uses the `parry3d` [18] crate, whose QBVH [19] is built once when the mesh is loaded and reused for every

ray of every frame (the reuse the prototype’s surface lookup lacked), and because the cast already returns the identifier of the triangle it hit, the normal needed for the single-bounce test becomes a direct lookup into a table computed once at load time, so the prototype’s most expensive operation disappears rather than merely becoming faster. Geometry is expressed with the statically typed `nalgebra` [20] library, and the binary wire format maps directly onto fixed-layout `#[repr(C)]` structs that are cast to and from bytes without per-field serialization. This is primarily a type-safety and maintainability gain (the wire layout is defined once in the type system and checked by the compiler) rather than something Python could not do, since `numpy` structured arrays achieve a comparable zero-copy encoding. The classification logic and acceleration structure are described in Chapter 5, and the measured throughput of the Rust backend is reported in Chapter 13.

The rewrite was an engineering decision, not a language benchmark. The PyVista prototype made Python look slower than it is, and a fairer Python baseline, namely a compute-oriented geometry library together with either process-based parallelism or a free-threaded interpreter, would have narrowed the gap considerably. What Rust provided was not raw arithmetic speed but a better fit for the final system: parallel per-ray classification free of the GIL and of any runtime-specific limitations. An acceleration structure built once and reused, type-safe geometry and a type-checked zero-copy binary protocol, and a single self-contained binary to deploy in place of a Python environment with several native dependencies. These are the properties that matter as the ray count grows and as the recorded dataset is eventually replaced by a live data source.

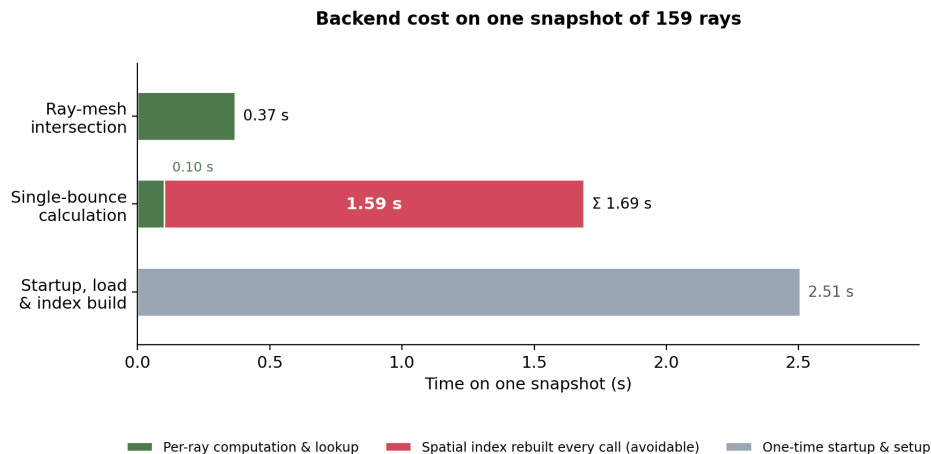


Figure 3.3: Backend cost of the PyVista prototype on a single snapshot of 159 rays, with rendering excluded. The red segment is the avoidable per-call index rebuild discussed in the text.

Preprocessing

Before a single ray can be classified, two raw inputs have to be turned into the form the backend actually consumes. The first is the measurement file produced by the SAGE processing of the D-MIMO recording, which holds the antenna-array positions and their local coordinate systems, the UE trajectory, and the estimated MPCs for every timestamp. The second is a LiDAR scan of the physical room, from which the environment mesh used for ray casting is reconstructed. A third input is supplied by the headset side of the project: a mesh of the same room captured by the Meta Quest 3, which serves only as a reference frame for alignment and anchor points from the UE5 application. How that mesh is captured on the device and transported to the backend is explained in Part II.

Preprocessing therefore has three responsibilities: reading the measurement file into a structured in-memory model (Section 4.1), reconstructing the environment mesh from the LiDAR point cloud (Section 4.2), and aligning that mesh, and with it every coordinate the backend emits, to the headset’s reference frame (Section 4.3). This stage produces a measurement model, watertight mesh and a transform that maps the coordinates onto the headset’s mesh.

4.1 Reading in the Measurement Data

The measurement file whose structure was introduced in Table 2.2 is a MATLAB v7.3 file which is based on the Hierarchical Data Format 5 (HDF5) format, so any HDF5 client can open them. The Python prototype used `h5py` and the Rust backend uses the `hdf5` crate. At v0.8.1 the high-level `hdf5` [21] crate has no safe API for following the HDF5 object references in the `z_t` field, where each cell is an 8-byte reference pointing at the actual MPC table rather than the table itself, so this is handled through a single contained `unsafe` block that calls the underlying HDF5 C API directly to dereference each reference into a dataset handle, avoiding a per-cell name lookup and bringing the load time down from tens of seconds to well under one.

Once read, the file is decomposed into a small set of structs: a `Sensor` (id, position, and a rotation matrix), a `Mpc` (propagation distance, azimuth and elevation angle

of arrival, and channel attenuation), an `MpcCell` (the list of `Mpcs` for one sensor at one timestamp), and a `MeasurementData` that owns all sensors, the UE trajectory, and every cell across sensors and timestamps. Note that `sensor` is used to describe the antenna panel since `sensor` is the term used in the datasets. The only transform applied at this stage is precomputing each sensor’s rotation matrix $\mathbf{R}_s$ once from the local-axis vectors stored in the file, so that the ray stage can map a measured direction into the world frame with a single matrix multiply rather than rebuilding the matrix per ray.

4.1.1 Preprocessing Dataset 2

The second dataset is structured differently, as was shown in Table 2.3. The main difference that affects processing is that the local coordinate systems of the antenna panels are not given. Inspecting the point cloud together with the first dataset shows that each panel’s z-axis points upwards, so it equals $[0, 0, 1]$, the only unknown per panel is the yaw. Recovering it requires the point cloud and the panel positions.

For every panel, a crop is taken from the point cloud, the points inside it are clustered, and the cluster that belongs to the panel is fit to extract its facing direction. The crop is a box centered on the panel position, its horizontal half-size is set to 45% of the distance to the nearest neighboring panel, capped at 0.6 m, so the crop can never reach an adjacent panel even when the panels are densely packed. Two height bounds relative to the panel position drop the floor and ceiling from the crop. Inside the crop, Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is run with its ϵ (maximum distance between two points) scaled to the local point spacing, and the resulting cluster nearest to the known panel position is kept. Principal Component Analysis (PCA) is then applied to that cluster: the panel is a thin flat surface, so the eigenvector belonging to the smallest eigenvalue is the face normal. The normal is oriented to point towards the centroid of the UE trajectory, which lies on the room-facing side of every panel, and its horizontal projection gives the panel’s y-axis. The x-axis follows from the cross product of the y-axis and z-axis. Figure 4.1 shows the axes in the point of view in front of an antenna panel. The code for extracting the axes was verified against the first dataset, which ships ground-truth axes, the recovered boresight lies within ten degrees of truth on every panel, with a median error around four degrees.

Once the local frames are known, the delay column is converted to a propagation distance through $d = c \cdot \tau$, with τ scaled by its declared unit, and the normalized amplitude is converted to a dB attenuation through $\alpha_{\text{dB}} = 20 \log_{10}(a)$, matching the convention used by the first dataset.

A JSON converter was written for each dataset, the second is designed so that its output schema matches the first, which leaves the rest of the backend unchanged. The schema is shown below (the `//` markers are annotations, not valid JSON):


```

{
  "sensors": [
    {
      "id": 0,
      "pos": [x, y, z],
      "xaxis": [x, y, z],
      "yaxis": [x, y, z],
      "zaxis": [x, y, z]
    }
    // ...one entry per panel
  ],

  "ue_truth": [
    [x, y, z]
    // ...one entry per timestamp (UE position over time)
  ],

  "z_t": [
    [
      { "paths":
        [
          [distance_m, az_rad, el_rad, attenuation], /* ... */
        ]
      }
      // ...one entry per sensor
    ]
    // ...one entry per timestamp
  ]
}

```

4.2 Building the Environment Mesh

To reconstruct the geometry of the physical room, we use a LiDAR scan from the provided dataset. The raw output is a point cloud, which is processed into a watertight mesh suitable for ray-intersection calculations. The mesh reconstruction pipeline is designed to handle the inherent noise and sparsity of the LiDAR data while preserving the critical geometric features of the environment.

4.2.1 Filtering the Point Cloud

The raw LiDAR scan contains everything in the scanner's view, not only the room geometry the ray engine needs. Reflections from windows or reflective surfaces are also present resulting in a noisy point cloud. The filtering stage reduces the cloud to points that lie inside the room and belong to its actual surfaces, while preserving disconnected interior objects such as desks and chairs so that they survive into the final mesh. Figure 4.2 shows the raw input. The filtering exists as a sequence of

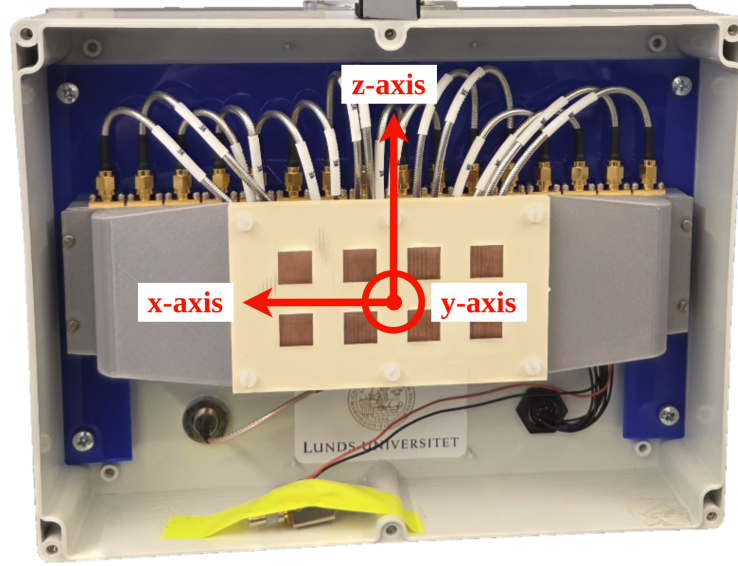


Figure 4.1: Local coordinate system of an antenna panel viewed from the front face. The y-axis points outward toward the user equipment (out of the page). Photographed by the authors.

steps explained below.

Core isolation. Points with intensity above the 70th percentile are clustered with DBSCAN [22] ($\epsilon = 0.2$ m, minimum cluster size 100 points). The largest of these resulting clusters is designated as the room core. This core is used exclusively as a height reference for the subsequent cropping stage.

Z crop. The full point cloud is filtered along the Z-axis using the vertical limits of the established core. A 2 m vertical safety margin is added to the core's upper and lower bounds to prevent aggressive clipping of the true floor and ceiling.

Ceiling alignment. To perform a more precise vertical crop, a 100-bin histogram [23] is computed along the Z-axis. Because the scanned room does not include the floor, the densest bin corresponds to the ceiling height. The point cloud is then clipped to a vertical range of $[Z_{\text{ceiling}} - 4.5 \text{ m}, Z_{\text{ceiling}} + 1.0 \text{ m}]$. The generous upward allowance ensures that coved ceiling edges, which may rise above the ceiling plane, are retained rather than truncated.

Wall extraction. The point cloud is restricted to a horizontal slab below the ceiling, excluding the top 0.5 m to avoid distortion from coved edges. Within this subset, an iterative Random Sample Consensus (RANSAC) [24] procedure extracts dominant planes ($ax + by + cz + d = 0$) using a 5 cm inlier distance threshold. RANSAC is a robust estimator used for geometric primitive fitting in noisy point

clouds [25]. The implementation here builds on Open3D’s `segment_plane` [22]. A candidate plane is accepted as a wall only if its normal vector is near-horizontal ($|c| \leq 0.2$) and it contains at least 500 inliers. To ensure distinct surfaces are found, the inliers of each validated wall are removed from the point cloud before the next iteration.

Horizontal crop via half-spaces. If at least three walls are extracted, they are used to geometrically enclose the room. Each wall plane is oriented inward toward the spatial median of all wall inliers. The entire dataset is then evaluated, retaining only points whose signed distance places them inside the intersecting half-spaces of every wall. A 0.2 m outward buffer is applied to this intersection to preserve recessed architectural features such as door jambs.

Fallback crop via bounding box. If fewer than three walls are recovered, the system falls back to cropping the point cloud to the bounding box of the room core. The full point cloud is then evaluated against this rectangular footprint, deleting any points that do not fall within the box.

Radius outlier removal. With radius outlier removal [26], a point is kept only if at least 10 neighbors lie within 0.2 m. This step removes isolated noise points that survived the previous geometric filters.

Figure 4.3 shows the cloud after the filtering. The difference is most visible at the side of the room where the windows are located. The unfiltered cloud (Figure 4.2) contains many points that belong to the reflections on the windows.

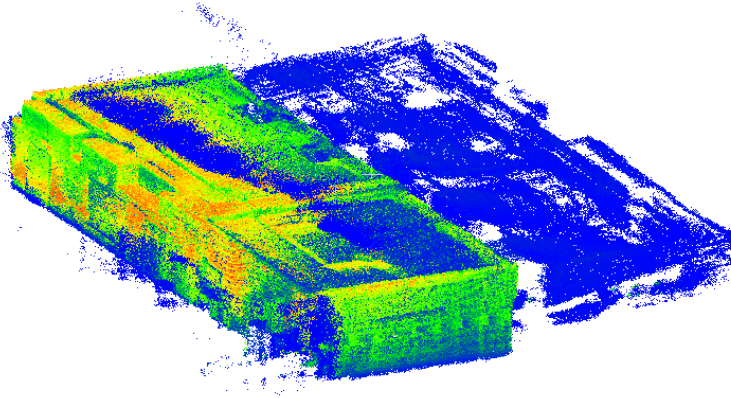


Figure 4.2: Point cloud from dataset 2 before filtering.

4.2.2 From Point Cloud to Mesh

The filtered point cloud is converted to a watertight triangle mesh by Poisson surface reconstruction, using the Open3D implementation [27]. Because this algo-

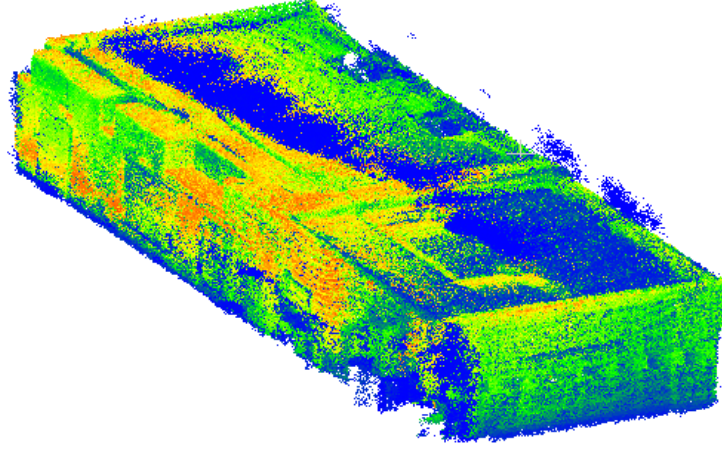


Figure 4.3: Point cloud from dataset 2 after filtering.

rithm requires an oriented surface normal at every input point, the pipeline first estimates and aligns these normals before extracting the surface.

Normals are estimated per point using Open3D’s principal-component approach, in which a small neighborhood around each point is gathered and the eigenvector corresponding to the smallest eigenvalue of its covariance matrix is taken as the local surface normal [22]. The neighborhood is defined through a hybrid KD-tree search that caps the query at 30 points within a 0.1 m radius. The resulting normals are not necessarily oriented consistently across the cloud, so Open3D’s `orient_normals_consistent_tangent_plane` routine is then applied, which builds a minimum spanning tree over the point graph and propagates a consistent orientation along its edges [27].

With the cloud properly oriented, the points are passed to the surface reconstruction algorithm. Initial tests for this pipeline explored Alpha Shapes and the Ball Pivoting Algorithm. Both methods construct the output mesh directly from input samples without adjusting their positions, which leaves any sensor noise present in the cloud visible as roughness on the reconstructed surface [27]. To achieve a smoother result, the pipeline instead uses Poisson surface reconstruction. This method casts surface recovery as a global Poisson equation whose solution is the indicator function of the implicit surface, and is therefore inherently resilient to noise because every input sample contributes only locally to a globally regularized field [28]. The Open3D implementation exposes an octree-depth parameter that controls how finely this implicit field is sampled and, consequently the level of detail in the final mesh [27]. The pipeline uses a depth of 9. Because Poisson reconstruction also fills in low-coverage regions by extrapolation, the resulting mesh contains triangles in areas with little supporting data. To remove these, the bottom 5% of the per-vertex density values returned by the algorithm are

dropped, and any degenerate triangles (triangles that reference the same vertex more than once [29]), duplicated vertices, or unreferenced vertices that result are removed [29].

While Poisson surface reconstruction successfully captures the overall geometry of the scanned environment, it also reproduces any residual noise that remains after the filtering stages. As a result, planar surfaces such as walls and ceilings often exhibit visible surface irregularities, as shown in Figure 4.4. Since these artifacts can negatively affect the accuracy of the ray-tracing stage in the proposed pipeline, an additional smoothing step is applied, described in Section 4.2.3.

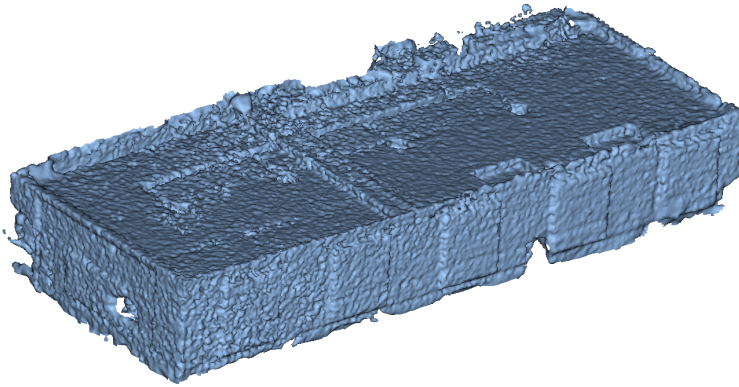


Figure 4.4: Mesh reconstructed by Poisson surface reconstruction.

4.2.3 Mesh Smoothing

Standard mesh smoothing algorithms compute new vertex positions using a weighted average of their first-order neighbors. The uniform Laplacian (or umbrella) operator forms the basis of Gaussian smoothing and Taubin’s non-shrinking low-pass filter [30], and was later combined with implicit integration for improved stability and efficiency on large meshes [31]. While these isotropic filters successfully attenuate high-frequency sensor noise, they apply smoothing equally across the entire mesh and therefore deform or remove high-curvature features [32]. Diffusion-based operators also cause vertices to drift tangentially along the surface, even in locally flat regions [31]. Consequently, applying these operators globally to architectural models inadvertently dulls the sharp depth steps that delimit critical structural features, such as doorways and recessed windows.

For wave-propagation simulation, the opposite geometric trade-off is required: planar architectural surfaces should be flattened to remove residual sensor noise, while the sharp edges and depth discontinuities between them should remain as crisp as possible. We achieve the flat surfaces with a region-growing planar segmentation algorithm followed by per-region orthogonal projection, applied to the reconstructed Poisson mesh. The procedure consists of five steps: the first three are repeated for each of a fixed number of random seed vertices, and the last two run once globally over all accepted regions.

Seed plane estimation. A local tangent plane is fitted by PCA to the 15 spatial nearest neighbors of the seed vertex, following the tangent-plane construction of Hoppe et al. [33]. The neighborhood centroid is taken as a point $\mathbf{c}$ on the plane, and the smallest-eigenvalue eigenvector of its covariance matrix gives the plane normal $\mathbf{n}_s$. Since PCA only fixes $\mathbf{n}_s$ up to a sign, it is flipped if needed to agree with the pre-computed vertex normal at the seed, which makes the signed comparison in the normal gate (Equation 4.2) well-defined.

Dual-gated flood fill. Starting from the seed, a depth-first traversal walks the mesh vertex adjacency graph. A candidate vertex v_i is admitted to the region only if it is still unclaimed by any previously grown plane and satisfies both

$$|\langle v_i - \mathbf{c}, \mathbf{n}_s \rangle| < \tau_d, \quad (4.1)$$

$$\langle \hat{\mathbf{n}}(v_i), \mathbf{n}_s \rangle \geq \cos \alpha_n, \quad (4.2)$$

where $\mathbf{c}$ and $\mathbf{n}_s$ are the centroid and unit normal of the seed plane fit, $\hat{\mathbf{n}}(v_i)$ is the unit vertex normal at the candidate, τ_d is the plane-distance threshold, and α_n is the maximum permitted angular deviation. The skip condition for already-claimed vertices guarantees that the planes recovered by successive seeds form a partition of the accepted vertices rather than overlapping regions.

Equation (4.2) on its own is the smoothness constraint used in normal-based region growing for unstructured 3D data [34]. In architectural scenes that constraint alone is insufficient: a planar wall and a recessed window inset share approximately the same surface normal direction, so a flood gated only on normals would freely cross the depth step at the frame. Equation (4.1) blocks that crossing by additionally enforcing an inlier distance to the seed plane, a criterion commonly paired with the normal-angle gate in region-growing plane segmentation [35]. The depth steps at window and door frames are typically of order 50-200 mm and therefore exceed τ_d , terminating the flood at the frame. Combined, the two gates segment two parallel but depth-offset surfaces into independent planes without any explicit edge detection.

Plane refinement. Regions with fewer than $\max(6, f_{\min} |V|)$ vertices are rejected, where $|V|$ is the total vertex count and $f_{\min}$ a user-set fraction. The vertices of a rejected region are released, so a later seed can still claim them. For accepted regions, the plane is refitted by re-running PCA on all accepted vertices, and the refined normal is flipped if needed to agree with the seed vertex normal. The refit improves the plane orientation, since it uses many more samples than the initial seed neighborhood.

Vertex projection. Every accepted vertex v_i is projected orthogonally onto the refined plane of its region,

$$v_i \leftarrow v_i - \langle v_i - \mathbf{c}, \mathbf{n} \rangle \mathbf{n}, \quad (4.3)$$

where $\mathbf{c}$ and $\mathbf{n}$ are the refined centroid and normal. After this step, the vertices in each region are exactly coplanar.

Boundary blending. Projection leaves accepted vertices exactly coplanar, but vertices the gates rejected keep their original noisy positions: they stick out from the flat planes as small spikes and pits, or form visible seams between adjacent planes. A blending pass fixes this: the d -ring neighborhood of each set of projected vertices is identified, and n iterations of unweighted Laplacian (umbrella) smoothing [30], [31] are applied only there. Most neighbors of a blended vertex lie on a projected plane, so outliers get pulled back to plane level and seams become smooth transitions. Vertices on the planes and vertices far from any plane are untouched. Because the smoother runs only on this narrowband, its shrinkage and feature-rounding stay within the band, and the planes themselves remain exactly flat.

The pipeline is deterministic: seed vertices are drawn from a fixed-seed random-number generator, so the same input mesh always produces the same output. The primary tuning knob is the plane-distance threshold τ_d from Equation (4.1): lower values give a stricter cut at depth edges and yield more, smaller planes, higher values merge more aggressively at the cost of bridging small discontinuities. The values used are $\tau_d = 0.05$ m, $\alpha_n = 30^\circ$, 500 seed vertices, a minimum region size of 0.3% of the total vertex count, a boundary ring depth of 4, and 8 Laplacian iterations on the boundary zone. These parameters were selected through visual inspection of the output mesh.

The results of the smoothing techniques can be seen in Figure 4.5.

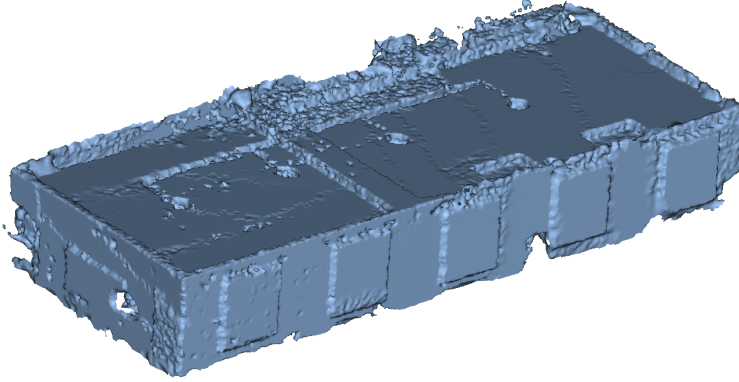


Figure 4.5: Mesh smoothing by planar segmentation and projection.

4.3 Aligning the Mesh to the Headset Frame

The reconstructed LiDAR mesh and the Meta Quest 3 mesh describe the same physical room but are expressed in unrelated coordinate frames. The two meshes were captured independently, each with its own world origin and no shared reference between them. The transform must therefore be recovered from the geometry alone. The goal is a single rigid transform T , a rotation followed by a translation, that maps every LiDAR coordinate into the headset frame. Both meshes are

gravity-aligned, so they share a common vertical axis (Z -up). The rotation then reduces to a single yaw, the angle that makes the dominant walls of the two meshes parallel. Alignment proceeds as a three-stage coarse-to-fine pipeline. First, dominant vertical walls are recovered from both meshes with RANSAC. Second, the dominant walls fix the rotation up to a 180° yaw ambiguity, producing two candidate transforms. For each candidate, the translation is solved in closed form from the matched walls. Third, both candidates are refined with point-to-plane Iterative Closest Point (ICP) [36], [37], and the one with the lowest residual is kept.

- Meta Quest 3 Scene Mesh
- LiDAR Mesh before alignment
- LiDAR Mesh after alignment

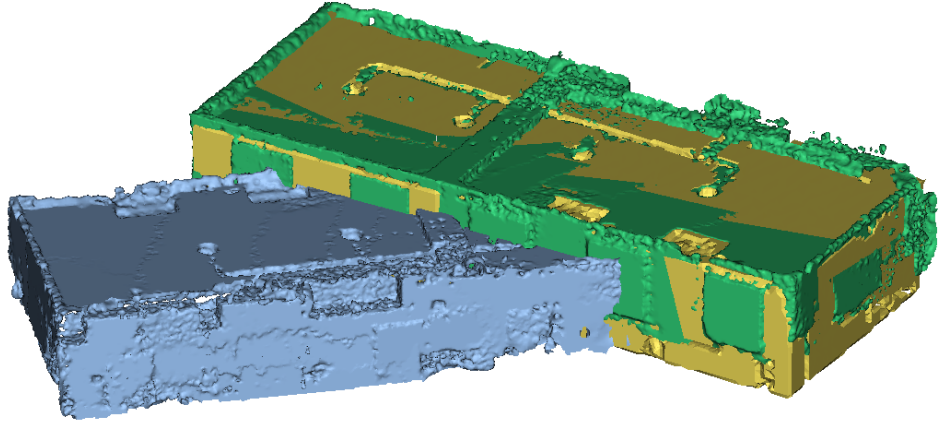


Figure 4.6: Spatial registration of the LiDAR and headset meshes. The LiDAR mesh is shown in blue, the headset mesh in orange, and the matched walls that drive the coarse alignment are highlighted in green. The final transform T maps the LiDAR mesh onto the headset mesh, so the two coincide after applying T to the LiDAR vertices.

Vertical wall extraction. A clean labeling of walls is not available for either mesh: the headset export carries no Quest object-id metadata that would separate anchor surfaces from the scanned geometry, and the LiDAR mesh contains thousands of small triangles per wall with several centimeters of sensor noise. Both meshes are therefore fed to the same RANSAC routine. Triangle centroids and face normals are precomputed from the input mesh. Any triangle whose normal satisfies $|\mathbf{n} \cdot \mathbf{u}| > 0.20$ with respect to the up-axis $\mathbf{u}$ is rejected as floor or ceiling. In each iteration two centroids $\mathbf{p}_0$ and $\mathbf{p}_1$ are drawn from the remaining pool. The

horizontal component of their displacement,

$$\mathbf{t} = (\mathbf{p}_1 - \mathbf{p}_0) - \mathbf{u} ((\mathbf{p}_1 - \mathbf{p}_0) \cdot \mathbf{u}), \quad (4.4)$$

gives the candidate wall's tangent direction, and the candidate plane normal is $\mathbf{n} = (\mathbf{u} \times \mathbf{t}) / \|\mathbf{t}\|$. This two-point sampling with gravity prior reduces the wall normal to one degree of freedom (yaw), so per-triangle noise cannot tilt the plane out of vertical the way a generic three-point fit would allow. Centroids within an inlier band of 0.05 m of the candidate plane are counted, the best candidate over 400 random trials is kept, and its inlier set is refitted by area-weighted PCA: the smallest-eigenvalue eigenvector becomes the refined normal, after which the full pool is re-scored and the plane is refitted again, with two or three such rounds enough in practice. Inliers are then removed and the procedure repeats up to a cap of six planes. A minimum inlier count of 1,000 triangles is set for both meshes. The result is a list of `PlaneFit` records, each carrying a normal, signed offset, centroid, planar extent, and total area, sorted by area-descending, so the longest wall is always at index zero.

Yaw seeding and the 180° ambiguity. The yaw seed is taken from the longest-wall pair. The source's longest wall normal is projected onto the horizontal plane perpendicular to the shared up-axis $\mathbf{u}$ to form unit vector $\mathbf{a}$. The target's longest wall normal is projected the same way to form unit vector $\mathbf{b}$. Their signed angle around $\mathbf{u}$ follows from the standard arctan form,

$$\sin \theta_{\text{seed}} = \mathbf{u} \cdot (\mathbf{a} \times \mathbf{b}), \quad \cos \theta_{\text{seed}} = \mathbf{a} \cdot \mathbf{b}, \quad (4.5)$$

$$\theta_{\text{seed}} = \text{atan2}(\sin \theta_{\text{seed}}, \cos \theta_{\text{seed}}). \quad (4.6)$$

Because the PCA refit canonicalizes every wall normal to a fixed half-space, the wall geometry alone cannot distinguish θ_{seed} from $\theta_{\text{seed}} + \pi$. A second hypothesis $\theta_{\text{seed}} + \pi$ is therefore always emitted. The eventual ICP residual ranks the two, and the wrong one typically grows by an order of magnitude during refinement.

Wall correspondences and translation. For each yaw hypothesis $\theta \in \{\theta_{\text{seed}}, \theta_{\text{seed}} + \pi\}$ the full source-to-target rotation is $R = R_{\text{yaw}}(\theta)$, which is applied to every source wall normal. A greedy one-to-one match is then formed by walking source walls in area-descending order: each source normal claims the target wall that maximizes $|R \mathbf{n}_l \cdot \mathbf{n}_q|$, where $\mathbf{n}_l$ and $\mathbf{n}_q$ are the source (LiDAR) and target (headset) wall normals, provided the dot product exceeds 0.9 (about 25° tolerance, which absorbs the residual RANSAC wobble). The translation that maps the rotated source onto the matched target walls is then solved in closed form. Each correspondence contributes one linear equation in the translation $\mathbf{t}$,

$$\mathbf{n}_q \cdot \mathbf{t} = d_q - \mathbf{n}_q \cdot (R \mathbf{c}_l), \quad (4.7)$$

where $\mathbf{c}_l$ is the source wall centroid and d_q the target wall's plane offset (defined so that $\mathbf{n}_q \cdot \mathbf{p} = d_q$ for any point $\mathbf{p}$ on the plane). Vertical walls have a negligible up-axis component in their normals, so the component of $\mathbf{t}$ along the shared up-axis

$\mathbf{u}$ is left unconstrained by (4.7) alone. One extra row pins the mean source-wall centroid projected onto $\mathbf{u}$ to its target counterpart, which fully determines the translation. Stacking the per-correspondence rows together with this up-axis row gives a linear system $A \mathbf{t} = \mathbf{b}$, where the rows of A are the normals $\mathbf{n}_q$ (and one $\mathbf{u}$ row) and the entries of $\mathbf{b}$ are the right-hand sides of (4.7) (and the corresponding up-axis mean). The system is solved via its 3×3 normal equations $A^\top A \mathbf{t} = A^\top \mathbf{b}$ rather than by Singular Value Decomposition (SVD). This ensures that if the geometry lacks enough intersecting walls to fully lock the room’s position (e.g., parallel walls in a hallway that allow infinite sliding), the $A^\top A$ matrix cannot be inverted and the alignment explicitly fails. In contrast, an SVD solver would silently guess a zero-translation for the unconstrained axis, producing a mathematically valid but physically incorrect alignment.

Point-to-plane ICP refinement. Each coarse candidate is refined separately using point-to-plane ICP. Because room geometry consists mostly of flat walls and ceilings, point-to-plane ICP is more effective than the standard point-to-point variant. It allows source points to slide along matched target surfaces rather than penalizing them for lateral offsets, leading to faster and more reliable convergence. A fixed set of 5,000 source triangle centroids is sampled once, deterministically. In every iteration each sampled centroid is transformed by the current T into a world position $\mathbf{p}_{\text{world}} = T \mathbf{c}_l$, and projected onto the target mesh via `parry3d`’s nearest-surface query, which returns the projected point $\mathbf{p}_{\text{proj}}$ together with the target triangle’s normal $\mathbf{n}_q$ (here the per-triangle normal at the projection, not the wall-plane normal). Correspondences are rejected when $\|\mathbf{p}_{\text{world}} - \mathbf{p}_{\text{proj}}\|$ exceeds 0.5 m or when the source and target normals disagree by more than 60° , the second filter matters when a source sample drifts onto the wrong face of a thin partition. Each surviving correspondence contributes one row $\mathbf{j}$ to a 6×6 Gauss-Newton system in the linearized pose increment $(\delta\omega, \delta\mathbf{t})$, with $\delta\omega \in \mathbb{R}^3$ indicating an axis-angle rotation and $\delta\mathbf{t} \in \mathbb{R}^3$ a translation:

$$\mathbf{j} = (\mathbf{p}_{\text{world}} \times \mathbf{n}_q, \mathbf{n}_q), \quad e = (\mathbf{p}_{\text{world}} - \mathbf{p}_{\text{proj}}) \cdot \mathbf{n}_q, \quad (4.8)$$

where e is the signed point-to-plane residual and $\mathbf{j}$ is the standard small-angle Jacobian of e with respect to the increment. The increment is assembled into a rigid update δT and applied as $T \leftarrow \delta T \cdot T$, the mean squared residual is recomputed, and iteration continues until the relative change drops below 10^{-4} or a hard cap of 30 iterations is reached. Convergence in practice happens in five to fifteen iterations.

Selecting the best candidate. Both yaw hypotheses are run through the same ICP configuration, and the candidate with the lowest final mean squared residual is kept. In a non-square room the wrong hypothesis grows worse during refinement because its correspondences are forced through walls on the opposite side of the room, the residual gap between the two is typically at least an order of magnitude. The surviving T is returned.

Output transform. Every world-space point the backend emits, the UE position and each ray’s origin and endpoint, is mapped through T before it is placed

on the wire, so the consumer receives coordinates already in the headset frame. Scalar quantities such as attenuation and propagation distance are invariant under rigid motion and pass through unchanged. The transform is applied in the UDP sender (Chapter 6).

Ray Computation

The preprocessing stage of Chapter 4 produces two inputs in a common frame: a measurement model holding, for every timestamp, the UE position together with the eight sensors and their estimated MPCs, and a watertight environment mesh. This chapter describes how those two inputs are combined into the per-frame stream of classified rays that the UE5 application consumes. For each timestamp the backend produces a list of classified rays, one per MPC. Every ray carries an origin, an endpoint, the measured channel attenuation and propagation distance, and one of four labels: *line-of-sight*, *single bounce*, *multi bounce*, or *out of bounds*. The label is derived purely from the measurement geometry and the reconstructed mesh: SAGE provides the MPCs, the mesh provides the room, and the backend tests each path against three canonical propagation mechanisms, with a residual label for paths whose geometry the room cannot accommodate.

5.1 Per-Ray Pipeline

Each MPC produced by SAGE arrives at one antenna panel with an azimuth-of-arrival ϕ , an elevation-of-arrival θ (zenith convention, $\theta = 0$ is straight up), a propagation distance d , and a channel attenuation. These four scalars, together with the antenna panel's world-space position $\mathbf{p}_s$ and rotation matrix $\mathbf{R}_s$ from Chapter 4, are the per-MPC inputs to the classifier. The classifier additionally reads the current UE position $\mathbf{u}$ and the reconstructed mesh, both shared across all rays in a frame. Figure 5.1 summarizes the geometry and the four-way classification developed in the rest of this section.

From spherical AoA to a world-space direction. The angle pair is first converted to a unit direction in the antenna panel's local frame using the standard spherical-to-Cartesian conversion in the zenith convention,

$$\mathbf{d}_{\text{local}} = (\sin \theta \cos \phi, \sin \theta \sin \phi, \cos \theta), \quad (5.1)$$

which is unit length by construction, since $\sin^2 \theta (\cos^2 \phi + \sin^2 \phi) + \cos^2 \theta = 1$. This direction is rotated into the world frame by the antenna panel's orientation matrix

$\mathbf{R}_s$,

$$\mathbf{d}_{\text{world}} = \mathbf{R}_s \mathbf{d}_{\text{local}}. \quad (5.2)$$

The straight-line endpoint that the path would reach if it traveled d meters without any interaction is then

$$\mathbf{e} = \mathbf{p}_s + d \cdot \mathbf{d}_{\text{world}}. \quad (5.3)$$

$\mathbf{e}$ has no physical meaning on its own: SAGE reports a propagation distance, not a destination, and the actual path may have bounced or grazed any number of surfaces. Equation (5.3) is therefore best read as a geometric probe whose role is to feed the classification tests below.

LoS test. With $\mathbf{u}$ the UE position at the current timestamp, define

$$d_{\text{direct}} = \|\mathbf{u} - \mathbf{p}_s\|, \quad \delta_{\text{end}} = \|\mathbf{e} - \mathbf{u}\|. \quad (5.4)$$

A ray is labeled LoS when both the measured propagation distance matches the direct UE to antenna panel distance and the probe endpoint lands close to the UE,

$$|d - d_{\text{direct}}| \leq \tau_{\text{los}}^d \quad \text{and} \quad \delta_{\text{end}} \leq \tau_{\text{los}}^e. \quad (5.5)$$

We use default tolerances $\tau_{\text{los}}^d = \tau_{\text{los}}^e = 0.5$ m, chosen by inspection in the validation visualizer of Section 5.2 and operator-tunable there.

Bounded ray cast. If the LoS fails, the path is cast against the environment mesh to identify its first surface interaction. To avoid self-intersections with the antenna panel or geometry immediately around it, the cast does not start at the panel itself: instead, its origin $\mathbf{o}$ is taken at a minimum intersection distance $\delta_{\text{min}} = 1$ m along the world-space direction $\mathbf{d}_{\text{world}}$, so the QBVH traversal never visits geometry within δ_{min} of the panel. A path of total propagation distance d cannot have traveled further than d meters in any straight-line segment, so the cast is also bounded at a maximum traversal range d_{max} : surfaces beyond that range cannot be the one this MPC interacted with and are excluded from the search. The bounded cast is therefore defined by

$$\mathbf{o} = \mathbf{p}_s + \delta_{\text{min}} \mathbf{d}_{\text{world}}, \quad d_{\text{max}} = \max(d - \delta_{\text{min}}, 0). \quad (5.6)$$

Like the LoS tolerances, δ_{min} is exposed as an operator-tunable parameter in the validation visualizer of Section 5.2. The ray is traversed through the QBVH acceleration structure that `parry3d` [18] builds over the triangle mesh at load time. The traversal yields either a first hit, returning the 3D contact point and the index of the intersected triangle, or nothing. Each mesh triangle has two sides distinguished by its vertex winding, a *front* face and a *back* face, and a ray can strike either. `Parry3d` signals a back-face strike by returning the triangle index shifted by the total triangle count, which the backend strips with a modulo so the same per-triangle normal applies whichever side the ray arrived from. The single-bounce angle test below then orients that normal toward the incoming ray, eliminating any remaining front/back asymmetry before the test is applied.

Out of bounds. A miss means the cast reached $d_{\max}$ without intersecting any mesh triangle. The path has therefore left the reconstructed room, either because the mesh is incomplete along the AoA direction or because SAGE produced an AoA or propagation distance that no path inside the room can satisfy. The endpoint is clipped to the room’s axis-aligned bounding box and the ray is labeled out of bounds. The original measured distance d is preserved on the ray so that downstream stages keep access to the real path length even when the displayed segment has been truncated to the room volume.

Why a single mesh hit is not enough. When the cast does hit a triangle, that hit alone does *not* mean the path was a single specular reflection. The cast reports only the first surface the ray crosses, not what happens beyond it. The real propagation in the room can be one of several things:

- A single specular reflection at that triangle.
- A multi-bounce path whose first physical contact is at that triangle but continues to other surfaces before reaching the UE.
- A diffraction or diffuse-scattering path whose total length matches d without corresponding to a specular bounce at that triangle.

A hit is treated as a single bounce only when its geometry is consistent with one specular reflection at that triangle. We apply two independent checks, total path length and the law of specular reflection, and a path that fails either is labeled multi-bounce.

The same single-bounce / multi-bounce distinction has been addressed for this measurement campaign by Xu et al. [9], using a Kalman-filter framework that combines map information with the MPC parameters and further classifies each interaction as reflection, scattering/diffraction, or mixed. Our classifier is deliberately coarser: the visualization needs a four-way label per ray, not a per-path mechanism characterization, so geometric consistency with the mesh is enough, and the resulting tolerances can be inspected and tuned directly in the validation visualizer of Section 5.2.

Single-bounce distance test. The hit point $\mathbf{x}_{\text{hit}}$ defines an incident leg of length $d_{\text{inc}} = \|\mathbf{x}_{\text{hit}} - \mathbf{p}_s\|$ and an outgoing leg of length $d_{\text{ue}} = \|\mathbf{u} - \mathbf{x}_{\text{hit}}\|$. A genuine single bounce travels exactly $d_{\text{inc}} + d_{\text{ue}}$ from the antenna panel to the UE, so the first condition is

$$|(d_{\text{inc}} + d_{\text{ue}}) - d| \leq \tau_{\text{sb}}^d, \quad (5.7)$$

with default $\tau_{\text{sb}}^d = 1$ m. If this fails, the measured propagation distance is longer than one reflection off this triangle could produce, so the path must have visited other surfaces along the way. The ray is labeled multi-bounce and the angle test below is skipped.

Single-bounce angle test. The distance test alone is not sufficient: many hit points along the ray could produce the same total length without corresponding to a genuine specular reflection. The second test enforces the law of reflection, which requires the angles of incidence and reflection, measured from the surface normal, to be equal.

Let $\hat{\mathbf{i}} = (\mathbf{p}_s - \mathbf{x}_{\text{hit}})/d_{\text{inc}}$ and $\hat{\mathbf{r}} = (\mathbf{u} - \mathbf{x}_{\text{hit}})/d_{\text{ue}}$ be unit vectors at the hit point, pointing back toward the antenna panel and forward toward the UE (Figure 5.1). The triangle normal $\mathbf{n}$ from the mesh is not guaranteed to point into the room: Poisson surface reconstruction (Section 4.2) produces normals whose orientation is locally consistent but globally arbitrary. We therefore flip the normal to face the incoming ray, $\mathbf{n} \leftarrow -\mathbf{n}$ whenever $\mathbf{n} \cdot \hat{\mathbf{i}} < 0$, so that both dot products are measured from the same side of the surface. The two angles are then $\theta_{\text{inc}} = \arccos(\hat{\mathbf{i}} \cdot \mathbf{n})$ and $\theta_{\text{ref}} = \arccos(\hat{\mathbf{r}} \cdot \mathbf{n})$, and the single-bounce condition is

$$|\theta_{\text{inc}} - \theta_{\text{ref}}| \leq \tau_{\text{sb}}^\theta, \quad \tau_{\text{sb}}^\theta = 25^\circ. \quad (5.8)$$

A ray that satisfies both (5.7) and (5.8) is labeled single bounce, a hit that fails either is labeled multi-bounce.

The 25° tolerance is generous on purpose. The angles of arrival inherit SAGE’s estimation noise. The bounce point inherits the mesh reconstruction error, especially from the planar smoothing of Section 4.2, and the triangle normal inherits both. A tighter tolerance would reject genuine single bounces whose geometry has been bent a few degrees by any one of those sources, a looser one would absorb multi-bounce paths that happen to land near a specular angle. The chosen value balances the two and can be adjusted in the validation visualizer.

Parallel Execution Across the Frame. All rays in a frame are independent. Each one reads only immutable shared data, the sensor table, the measurement entries, the scene, and the classification configuration, and writes only its own output record. The MPCs per antenna panel, across all panels, are flattened into a single list of sensor-path pairs, so the unit of work is the individual ray rather than the per-panel cell. The list is traversed by the Rayon [38] data-parallel runtime, which distributes the work units across all available cores without any explicit synchronization. Because the most expensive step per ray is the bounding-volume traversal, this is also where the Rust implementation gains most of its advantage over the original Python prototype: traversal and classification run on native, lock-free worker threads instead of competing for the Python GIL. The collected list of classified rays is then handed to the streaming layer of Chapter 6 as the body of a single per-timestamp frame.

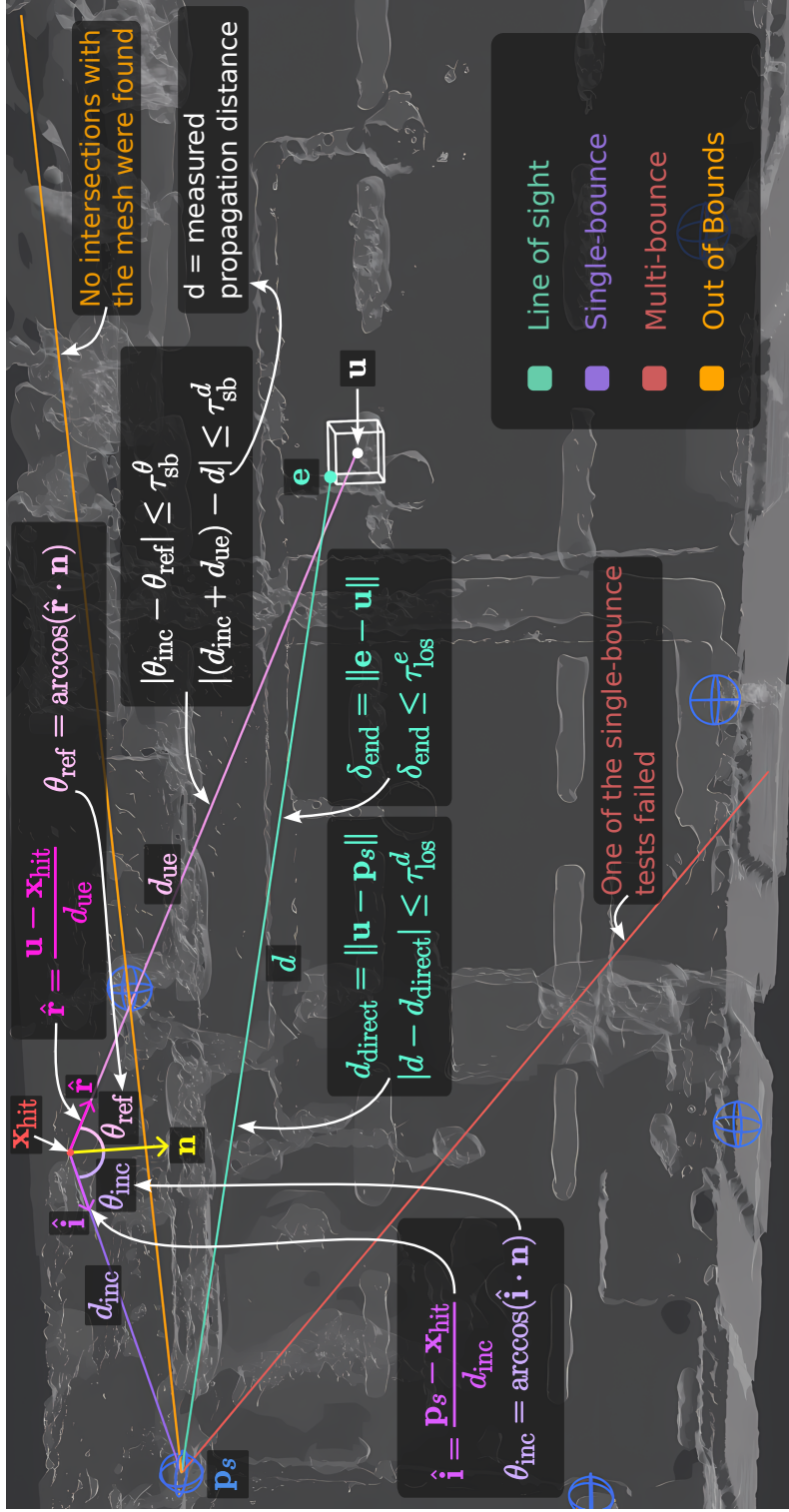


Figure 5.1: Visualization of the math behind the ray classification.

5.2 Validation Visualizer

The classifier of Section 5.1 is silent by design: it labels every MPC as one of four types and forwards the result to the streaming layer without any human-readable output. Verifying that the labels correspond to physically plausible paths, that the tolerances are set sensibly, and that the alignment of Section 4.3 has not introduced a hidden offset all require seeing the rays in the room rather than reading numbers in a terminal. A dedicated desktop visualizer was written for that purpose. It is not part of the live data path to the UE5 application and does not run on the headset, but it was the most useful debugging tool during development: most classification bugs, mesh defects, and alignment errors were caught in the visualizer first, well before they would have shown up in the UE5 renderer where the cause is much harder to diagnose.

The visualizer is built on the Bevy game engine [39], with the egui [40] immediate-mode toolkit providing the side-panel controls. It loads a measurement file and a mesh through the same routines the production backend uses, runs the classifier over every timestamp in the file, and presents the result as an interactive playback in a 3D viewport. Figure 5.2 shows the interface.

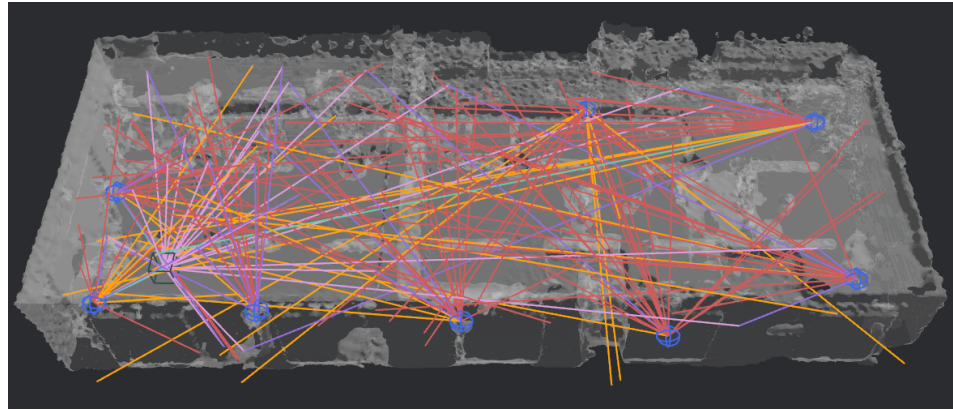


Figure 5.2: Desktop visualizer used to validate the ray classification. The mesh is rendered semi-transparent, so rays inside the room remain visible. Rays are colored by type (line-of-sight, single bounce, multi bounce, out of bounds), the antennas and the UE are drawn as wireframe markers. The side panel exposes playback, filters, the five classification tolerances, and an optional ground plane.

Replaying a measurement file. The 3D viewport draws the room mesh semi-transparent, so that rays inside the room remain visible from any angle. For each timestamp the antenna panels are drawn as spheres at their world-space positions, the UE is drawn as a small cube at its tracked position, and every classified ray is drawn as a colored line. Single-bounce rays are drawn with a second leg, a separate slightly lighter segment from the bounce point to the UE.

The four ray types use a consistent color scheme (green for line-of-sight, purple for single bounce, red for multi bounce, orange for out of bounds).

A playback bar at the top of the side panel walks the timestamp axis. Frames can be scrubbed manually with a slider, stepped one at a time with the arrow keys, or played back continuously at a selectable rate up to 60 frames per second. Together with the per-type toggles (which independently hide or show line-of-sight, single-bounce, multi-bounce, and out-of-bounds rays) and a per-antenna filter (which masks individual panels), this is the fastest way to answer questions of the form “where are the single-bounce rays going at this point in the trajectory” or “why does antenna 3 suddenly produce only out-of-bounds rays after timestamp 600”.

Live classification tuning. The five tolerances of Section 5.1 (τ_{los}^d , τ_{los}^e , δ_{min} , τ_{sb}^d , and τ_{sb}^e) are exposed as sliders in the side panel. When any slider is moved, the current frame is re-classified on the next render tick and the viewport updates immediately. The underlying precomputed frames are not modified, so returning the slider to its default restores the production result without a reload. A reset button in the side panel restores all five defaults at once.

Per-ray inspector. Clicking on any ray in the viewport opens a floating inspector window for that ray. The inspector shows the ray’s type, the antenna it originated from, the origin and endpoint in world coordinates, and the per-leg distances (one leg for line-of-sight, multi-bounce, out-of-bounds rays and two legs for single bounce). It also reports the measured propagation distance d alongside the geometric path length of the displayed segments, so the residual between the two can be read directly and compared against the tolerances of Section 5.1. The inspector also reports the channel attenuation in dB.

Ground-plane reflection probe. An optional ground plane can be toggled on, when enabled, any ray that crosses the plane is split at the crossing into a primary segment and a specularly reflected continuation. The reflection is computed by the standard formula $\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$ at the intersection point. This is not part of the propagation model: nothing in the classifier knows that the floor exists separately from the rest of the mesh. The toggle is intended to make floor-bounce geometry legible by eye when the mesh does not cleanly capture the floor. The plane’s center can be nudged along all three axes interactively. This makes it easy to see which multi-bounce rays in a frame would have hit the floor first if the floor were part of the mesh.

Alignment overlay. The visualizer can also accept a rigid transform (produced by the alignment of Section 4.3) and a separate display mesh. When both are supplied, ray casting still runs against the LiDAR mesh in its original frame, so the classification is identical to the production result. The transform is then applied to every computed ray and antenna position before they are drawn, and the display mesh is rendered as the background geometry in its own frame. In practice the display mesh is the headset mesh and the transform is the source-to-target one from Section 4.3, so the headset’s geometry and the transported ray

endpoints overlay each other in the same 3D viewport. Any residual rotation or translation mismatch in the alignment is then visible directly as a gap between the rays and the headset walls, without waiting for the data to reach the UE5 renderer.

The visualizer is therefore not a contribution to the wave-propagation engine itself, but it is the tool that made every other part of the backend debuggable. It is mentioned here, alongside the classifier whose output it exists to inspect, because the trustworthiness of the labels in Section 5.1 depends in practice on the ability to look at them.

Communication

Chapter 5 ends with a list of classified rays in memory. This chapter describes how that list leaves the backend and reaches the headset. The wire format is intentionally minimal: each classified frame is packed into a contiguous byte buffer, sent as a single UDP datagram. The consumer (UE5 application of Part II) listens on a fixed UDP port, validates each datagram, reinterprets the bytes as records, and consumes them. The datagrams themselves carry no handshake, no acknowledgment, and no retransmission, their only loss-recovery strategy is the next frame.

Around this stateless data path sits a small Hypertext Transfer Protocol (HTTP) control plane that handles per-headset calibration, session tracking, and per-headset routing of the shared frame stream. Sections 6.1 and 6.2 cover the wire format first, Section 6.3 then describes the server that hosts both the HTTP endpoints and the UDP streams, and explains how a per-headset transform is applied to the same shared frames before they go on the wire.

6.1 Transport and Encoding

The interface between the backend and the headset are a one-way, frame-rate-paced stream of small messages whose only loss-recovery strategy is the next frame. TCP's in-order, retransmission-based delivery would actively work against that: a single dropped packet would stall the rendered scene until the retransmission arrives, by which point the data has already been superseded by a newer frame. UDP simply drops the late packet and the renderer continues on the next one, which is the correct behavior for a visualization that has no use for stale data.

The encoding is a packed binary layout rather than a self-describing format such as JSON or Protocol Buffers. With roughly 160 rays per frame (for the first dataset) the per-ray cost dominates: the same payload is substantially larger in JSON (because the floats are American Standard Code for Information Interchange (ASCII)-encoded and every field is named), it requires a parsing pass before any value can be consumed, and it forces an allocation per ray on the receiving side. The binary layout below makes the receiver's parsing reduce to a length check and a pointer cast: the bytes that arrived on the wire are reinterpreted in place as

the same C-compatible records the sender wrote, with no copy and no per-field deserialization.

6.2 Wire Format

A datagram is one 28-byte header followed by one 44-byte record per classified ray. Both records are declared with a stable C-compatible layout, are required to contain only plain-old-data fields, and are serialized by reinterpreting their bytes directly through the `bytemuck` crate [41]. The same declarations are linked into the sender and the receiver, so the wire format is the in-memory format on both sides and any change to one side is a compile error on the other.

Header. The header carries the small set of fields the renderer needs before it can use any of the rays in the payload: a magic constant for sanity-checking, a sequence number that lets the consumer detect dropped or reordered packets, the index of the timestamp inside the original measurement file, the number of ray records that follow, and the UE position at that timestamp. Table 6.1 lists the byte layout.

Table 6.1: Wire header structure. Transmitted once per datagram.

Offset	Field	Type	Bytes	Meaning
0	magic	u32	4	Fixed constant identifying the frame format.
4	sequence	u32	4	Monotonic counter, wrapping at $2^{32} - 1$.
8	timestamp_idx	u32	4	Index corresponding to the measurement dataset.
12	num_rays	u32	4	Number of ray records appended to this header.
16	ue_pos	f32[3]	12	UE world position (x, y, z) .
Total Header Size			28	

The magic value is a 32-bit constant that spells out the four ASCII characters **W**, **A**, **V**, **E**. The receiver checks it before reading any other field and silently drops the packet otherwise, which catches not only protocol mismatches but also stray traffic that happens to land on the listening port. The sequence number is local to one sender and is purely informational: nothing in the protocol acts on it, but its presence lets the receiver log gaps as packet loss without inferring the loss from

timing. The timestamp index is the same index that addresses the measurement file, so a downstream tool can recover the original recording position of a frame even after it has been transported, rotated, and rendered. The ray count tells the parser how many 44-byte records follow the header, which is the only piece of information the receiver needs in order to slice the rest of the buffer.

Ray entry. Each ray is encoded as a 44-byte record whose layout matches the per-ray information used by the classifier, with the addition of a per-frame index and a one-byte type tag. Table 6.2 lists the byte layout.

Table 6.2: Wire ray entry structure. Appended `num_rays` times following the header.

Offset	Field	Type	Bytes	Meaning
0	<code>ray_id</code>	<code>u32</code>	4	Per-frame ray index, 0-based.
4	<code>sensor_id</code>	<code>u32</code>	4	Sensor (antenna) index.
8	<code>ray_type</code>	<code>u8</code>	1	Encoding: 0=line-of-sight, 1=single bounce, 2=multi bounce, 3=out of bounds.
9	<code>_pad</code>	<code>u8[3]</code>	3	Memory alignment padding, strictly zero on the wire.
12	<code>origin</code>	<code>f32[3]</code>	12	Ray start position in world space.
24	<code>endpoint</code>	<code>f32[3]</code>	12	Ray end position in world space.
36	<code>attenuation</code>	<code>f32</code>	4	Channel attenuation as a linear amplitude.
40	<code>propagation_d</code>	<code>f32</code>	4	Measured total propagation distance in meters.
Total Ray Size			44	

Three details of this record deserve a sentence each. First, the three padding bytes at offset 9 exist so that the four-byte `origin` that follows starts on a four-byte boundary, which the C-compatible layout requires; without explicit padding the compiler would have inserted it silently, but writing it as a field keeps the on-wire size deterministic across compilers. Second, the `ray_type` encoding matches the

four-value enumeration produced by the classifier, so the receiver can branch on the raw byte without lookups. Third, the attenuation is sent in linear amplitude (not dB), the consumer takes the logarithm only at the point where it actually wants decibels, which does not happen on the AR application, but does happen in the validation visualizer.

Total payload size. With N rays the datagram is exactly $28+44N$ bytes. The recorded frames carry around 160 rays each, giving a typical payload of 7,068 bytes, comfortably below the 65,507-byte payload limit of a single UDP datagram [42]. UDP fragmentation is transparent to the application.

6.3 Backend Server

The pipeline of Chapter 5 runs as a long-lived backend server that exposes a small HTTP API for control traffic and emits per-headset streams of UDP datagrams for the data traffic. A single binary hosts both planes: the HTTP service is built on the Axum web framework [43] over the Tokio asynchronous runtime [44], while the UDP side uses Tokio's sockets, so a single task scheduler drives both planes. At start-up the server loads the measurement data and the LiDAR mesh, classifies every frame once into memory, and spawns one frame producer that loops these frames at the configured rate and publishes each over an in-process broadcast channel. The HTTP routes are registered once at start-up, per-headset resources are created lazily on connection: a `POST /calibrate` runs mesh registration, off-loaded to Tokio's blocking thread pool and cached by mesh hash, and then spawns a dedicated UDP task that subscribes to the broadcast channel and streams transformed frames to that headset. A background sweeper reaps idle headsets and tears down their UDP tasks.

Calibration handshake. When a headset starts up it scans its own room mesh, exports the vertices and indices as JSON, and POSTs the result to `/calibrate` on the server. The server runs the alignment routine of Section 4.3 against the LiDAR mesh, caches the result, and returns a JSON body containing the headset's source IP (echoed back, so the headset can confirm it), the ICP residual, and a 4×4 row-major matrix encoding the rigid transform. This transform is currently unused in the pipeline, but is retained as a fallback in case the headset needs to perform local geometry transformation.

Registration caching. Alignment is the slowest step of the handshake, since it runs RANSAC and ICP on tens of thousands of vertices (Section 4.3), so the server hashes the incoming vertex list and caches the resulting transform keyed by that hash. A returning headset whose scan has not changed gets the same transform back without re-running registration. A headset that has rescanned its room produces a different hash and triggers a fresh registration. The cache lives in memory and is discarded on restart. This is acceptable because registration runs only once per headset per unique scan.

Scene layout. Once a headset is calibrated it issues a GET to `/layout`, which returns the sensor table (antenna panels positions) transformed into the headset's world frame. Each sensor entry contains an identifier, the name of the UE5 blueprint that should be spawned for it, the world-space position, and a yaw angle around the vertical axis. The yaw is derived from the sensor's local x-axis after applying the same calibration transform, projected onto the horizontal plane: this lets the UE5 application spawn antenna actors with the correct facing direction without having to know the full 3×3 rotation matrix. A request for `/layout` from an uncalibrated IP returns 409 Conflict, the headset is expected to retry the handshake first.

Frame production and broadcast. After the measurement file is loaded and classified at start-up, the resulting array of frames is computed once and never modified again. A single producer task wakes at the configured frame rate, takes the next frame in order (wrapping around at the end), and publishes it on a Tokio broadcast channel as a reference-counted handle (`Arc`). The broadcast layer therefore fans one frame out to all connected headsets with only a reference-count increment per subscriber, rather than copying the frame into each receiver's queue. Each per-headset task then clones its received frame once, applies that headset's calibration transform to the copy in place, and encodes the result for UDP, the shared frame array itself is never mutated.

Per-headset UDP stream. A successful calibration spawns one independent task per headset. Each task subscribes to the broadcast channel, owns its own UDP socket bound to an ephemeral port, and is given the per-headset calibration transform at creation time. For every frame received from the broadcast, the task makes a local copy of the frame, applies the per-headset transform to the copy in place, encodes it through the same per-frame encoder that Section 6.2 documents, and sends one UDP datagram to the headset's IP on the configured UDP port. The sequence counter is local to each headset task so different headsets see contiguous numbering on their own stream regardless of how many other clients are attached. The counter is a 32-bit value with explicit wrapping arithmetic, which means it loops back to zero after 2^{32} increments.

Session lifecycle. Each calibrated headset has a server-side session record holding its transform, the timestamp of its last contact, and the task handle for its UDP stream. The headset issues a GET to `/ping` every few seconds, the handler refreshes the last-contact timestamp and replies with 200 OK or, if no session is present, 409 Conflict. A graceful shutdown sends DELETE `/session`, which aborts the UDP task and removes the record immediately. A background sweeper runs on a fixed interval and removes any session whose last contact is older than the configured timeout, aborting its UDP stream in the process. The same lifecycle applies to repeated POSTs from the same IP: a fresh calibration swaps in the new session and then aborts the previous UDP task, so a headset that reconnects without first issuing DELETE `/session` converges to a single stream rather than accumulating one per reconnect (the old and new tasks overlap only for the brief instant between the swap and the abort).

Operational scope for pre-recorded data. For pre-recorded datasets the server is only required during the initial calibration handshake and the streaming window needed to fill the headset’s frame buffer. Because the headset buffer keys on the timestamp index and silently deduplicates any frame it has already stored, subsequent loops of the broadcast are harmlessly ignored. The server connection can therefore be dropped at any point after the complete dataset has been received, and the researcher is free to analyse the full dataset on the headset independently.

6.4 Diagnostic Sender and Receiver

Two small command-line programs sit alongside the server and exist solely for development. They were the first end-to-end clients of the wire format and remained useful afterward as a way to exercise either side of the protocol in isolation from the production server and the headset. Their relationship to the rest of the system is the same as the validation visualizer of Section 5.2 to the classifier: not part of the production path, but the easiest way to confirm that one stage of the pipeline is doing what it claims to do.

Standalone sender. The standalone sender predates the HTTP server and remains the simplest way to drive a UE5 application without bringing the full calibration service online. It loads the measurement file and the mesh, computes every frame once, and if a persisted alignment transform is supplied (Section 6.5), applies that transform to every frame in place, without one, the frames are sent in the raw measurement coordinate frame. It then loops indefinitely over the resulting buffer at a configurable frame rate. Each iteration encodes one frame through the same encoder the server uses, sends a single UDP datagram to a fixed target host and port, and sleeps for whatever remains of the per-frame period. Because that remaining time is computed from the elapsed time measured at the start of the iteration, using a monotonic clock rather than a fixed delay, a slow encoding step consumes part of the sleep budget instead of being added on top of the period, keeping the average rate close to the target. There is no handshake on the sender’s side: only the target address and frame rate come from the command line and packets begin flowing immediately.

Reference receiver. The reference receiver listens on the configured UDP port and prints a short summary of every datagram it sees. Its purpose is not consumption but inspection: it is the smallest possible client of the wire format, intended to be run alongside either the server or the standalone sender to confirm that packets are arriving, that they parse correctly, that the type counts are sensible, that each antenna is represented, and its sample rays look plausible before the same bytes are pointed at the UE5 application.

Decoding pipeline. For every received datagram the receiver checks that the length is at least 28 bytes (the header size) and that the magic value matches. Failure of either drops the packet with a warning and continues. With those two checks in place, the header is reinterpreted directly out of the first 28 bytes through

the same bit-cast the sender used, and the tail of the buffer is reinterpreted as a slice of ray records. A consistency check compares the declared `num_rays` field with the number of 44-byte records that actually fit in the remaining bytes, a mismatch is reported, and the packet is dropped rather than partially consumed. The decoded view is then handed to the application as a typed slice with zero copies and zero allocations.

Per-frame summary output. For every accepted datagram the receiver prints one summary line containing the sequence number, the timestamp index, the total ray count, the breakdown by ray type, the datagram's byte count, and the source address. It also prints the first two rays per antenna, so the world-space origin/endpoint coordinates and the propagation distances can be spot-checked for sanity. The summary line is deliberately compact, so the per-frame output stays readable while the stream is running.

Relationship to the headset client. The UE5 application of Part II implements its own decoder against the same wire format, with the same length and magic checks and the same bit-cast pattern, so the two consumers see identical bytes and interpret them identically. The reference receiver described here is not a part of the headset application, it is the desktop diagnostic that was used to develop and debug the wire protocol independently of UE5, in the same way that the validation visualizer was used to develop the classifier independently of the renderer.

6.5 Persisted Alignment Transform

The alignment routine of Section 4.3 has two execution paths in the final system. The first is the HTTP handshake of Section 6.3, which runs registration in response to every fresh headset connection and never writes the result to disk. The second is a saved-file path used during development and for stream replay: the alignment binary is run once against an example headset mesh, writes its output to a small JSON file, and the standalone sender of Section 6.4 loads that file at start-up and applies the same transform to every frame before encoding. Both paths produce coordinates in the headset frame at the point where bytes go on the wire, so downstream consumers cannot tell the difference.

The on-disk form is a small JSON object that contains the translation as a length-three array of meters, the rotation as a unit quaternion with components (w, i, j, k) , and (purely for human inspection) the corresponding 4×4 homogeneous matrix. Only the translation and quaternion are read on load, the matrix field is ignored entirely so the file can be hand-edited or pasted into other tools without any risk of an inconsistent matrix being used.

Carrying the transform in a small static file rather than over the wire keeps the per-frame protocol stateless on the data plane and makes a stream replayable from a packet capture without requiring the alignment step to be repeated. The same property holds for the HTTP path, where the transform travels only once per

headset session in the calibration response and is never re-sent inside the UDP stream itself.

Part II

AR Development

Game Engine Selection

Two engines dominate standalone XR development on the Meta Quest platform: Unity and Unreal Engine. Although both are general-purpose game engines capable of shipping a Quest 3 application, they approach the problem from rather different directions. Unity grew out of a desire to lower the barrier to entry for game development and centers on a managed C# runtime, a component-based object model, and a large ecosystem of overlapping plugins. Unreal Engine, by contrast, evolved alongside Epic's own AAA titles and exposes a C++-first architecture with deterministic memory control, a unified rendering pipeline, and a Blueprint visual scripting layer sitting on top. These foundational differences propagate into every part of the developer experience, and they matter unevenly depending on what the application is doing.

The comparison that follows is shaped by the specific workload this project imposes: a real-time visualization system running on Meta Quest 3, where performance is constrained by a strict frame-time budget and the work consists primarily of frequent per-tick updates across many simulated elements rather than complex scene rendering. Four dimensions are examined in turn: rendering performance and frame-time stability under the Quest 3's 11.1 ms budget at 90 Hz [45], development workflow including iteration speed and learning curve, integration with the Meta Horizon OS platform, and the level of control each engine provides over per-frame simulation and update logic. The Meta Quest 3 was the only headset available for this project, so its selection is taken as a given here. A retrospective discussion of how other headsets compare and what they might have offered is provided in Chapter 12.

7.1 Architecture and Memory Management

The two engines differ most consequentially in their object models and memory management, both of which directly affect XR frame-time stability.

Unity uses a component-based model in which every entity is a generic `GameObject` whose behavior is defined by attached components, scripted in C# on a managed runtime (Mono or Intermediate Language to C++ (IL2CPP)) [46]. Memory is

reclaimed by an automatic Garbage Collector (GC) using the Boehm–Demers–Weiser collector [47], which introduces rhythmic CPU spikes as it sweeps. A particularly undesirable behavior in XR, where a single dropped frame is visible as judder [48]. Unity 6 mitigates this with an incremental GC [48] and the Data-Oriented Technology Stack (DOTS) for parallel CPU work [49], but the engine still runs on Mono/IL2CPP with the Boehm collector [47]. A planned CoreCLR migration in Unity 6.8 [50] replaces, rather than removes, the managed collector: a generational GC shortens collection pauses without eliminating their non-determinism, so collection timing remains outside the developer’s control.

Unreal Engine is built around **Actors** extended by components, with core logic in C++ and higher-level scripting in Blueprints [51], at the cost of a steeper learning curve and a heavier baseline footprint than Unity [52]. C++ allows manual, deterministic memory management [53]: an internal GC exists for the reflected `UObject` system, but performance-critical paths remain under the developer’s control. For a workload in which numerous objects are updated every tick, keeping those paths off the garbage collector provides a more predictable baseline for sustained XR rendering [53].

This footprint difference is worth weighing against the platform’s hard limits. The Meta Quest 3 reserves part of physical RAM for tracking and system services, leaving applications a maximum proportional set size of 5.75 GiB before risking abrupt, OS-enforced termination [54]. Unreal’s heavier starting point consumes a slightly larger fraction of this ceiling than Unity, but the difference is small, and the deterministic allocation behavior was judged more valuable for this project than the modest memory saving Unity would have offered.

The rendering pipelines reflect a similar split. Unity exposes the Universal Render Pipeline (URP) for standalone XR and the High Definition Render Pipeline (HDRP) for high-end platforms via its Scriptable Render Pipeline framework [55]. Unreal instead uses a unified architecture that scales by target platform: for standalone mobile XR it defaults to a Mobile Forward Renderer [56], [57], with deferred shading available only as an explicit opt-in [56], further optimized here through the Meta fork of the engine (Section 7.4.1).

7.2 Performance

The Quest 3’s Snapdragon XR2 Gen 2 offers a substantial GPU uplift over its predecessor but a comparatively modest CPU improvement, making CPU overhead (logic, draw calls, engine bookkeeping) the more common bottleneck on standalone hardware. Workloads that perform substantial per-tick computation on the CPU therefore benefit disproportionately from engine architectures that minimize managed-runtime overhead.

Unity’s URP is well-optimized for the Quest profile out of the box. The GPU Resident Drawer offloads draw-call management from CPU to GPU, and Spatial Temporal Post Processing (STP) provides lightweight upscaling without significant overhead [58]. Unreal requires more deliberate configuration but, when paired with

the Meta fork of the engine, gains mobile-specific optimizations such as Uniform Based Dynamic Local Lights and Emulated Uniform Buffers, reported to yield up to a 15% improvement in dynamically lit scenes [59].

For XR, frame-time consistency is a more important metric than peak throughput: a single dropped frame causes perceptible judder and contributes to motion sickness, while headroom above the display refresh rate is wasted by the compositor. Unreal’s deterministic allocation model, described in the previous section, directly reduces the risk of the non-deterministic frame spikes that managed-runtime engines cannot fully eliminate. Both engines additionally support Late Latching to reduce motion-to-photon latency by up to one frame [60].

7.3 Development Workflow

Unity’s primary workflow advantage is C#: a managed language with extensive documentation, gentle syntax, and a low entry barrier [52]. Unreal’s C++ requirement, with header/source separation, manual memory management, and reflection macros (UCLASS, UPROPERTY), has a noticeably steeper learning curve. Blueprints lower the barrier for higher-level logic but do not eliminate the C++ requirement: low-level engine extensions and performance-critical paths still require native code, and Blueprint graphs are stored as binary uassets that version-control systems cannot diff or merge cleanly. The canonical recommendation is a layered architecture with C++ for core systems and Blueprints for content and rapid iteration.

Iteration time differs in character rather than magnitude. Unity recompiles modified C# assemblies and performs a domain reload on entering Play Mode. This scales with serialized state and routinely costs 10-20s on larger projects [61], though Unity 6’s opt-in disabling of domain reload reduces it to near-instant at the cost of manual static-state management [62]. Unreal compiles Blueprints incrementally and uses Live Coding to hot-patch C++ changes confined to function bodies, with cycle times around 10-15s; structural changes (headers, constructors, default properties) still require a full rebuild [63]. Neither engine is uniformly faster, and in practice both reach acceptable iteration speeds with appropriate configuration.

Both engines provide mature profiling tooling for the Quest. Unity offers the Unity Profiler, Memory Profiler, and Project Auditor [64], while Unreal exposes Unreal Insights, supported by Meta for on-device profiling over USB [65]. For this project, the relevant question is not whether Unity is generically easier, but whether Unreal’s added complexity is acceptable given prior C++ experience.

7.4 XR and Meta Quest Integration

Both engines now support the full Quest 3 feature set, but their XR ecosystems differ in shape.

Unity exposes XR functionality through a layered package architecture. The Unity OpenXR Plugin, which Meta now recommends over the deprecated Oculus XR

Plugin [66], sits beneath the XR Interaction Toolkit (XRIT) and the XR Hands package. Meta-specific functionality is delivered through a parallel set of Meta XR SDKs (Core, Interaction, and Platform) together with the Meta Mixed Reality Utility Kit (MRUK) and the Building Blocks editor extension for drag-and-drop feature setup [67], [68]. Unity’s stack is therefore broad but partially overlapping in scope.

Unreal’s XR support is more integrated. Epic ships a native VR Template with a configured pawn, motion controllers, teleport locomotion, and a generic grab component [69]. Quest-specific features such as passthrough, scene understanding, spatial anchors, the Depth API, and fixed foveated rendering are exposed through the single Meta XR plugin layered on top of Epic’s native OpenXR backend [70]. A Meta XR Interaction SDK for Unreal Engine 5.5+ provides controller-driven hands and gesture detection analogous to its Unity counterpart [71].

Feature parity for the capabilities relevant to this project (passthrough, hand tracking, scene understanding via the Scene, Mesh, and Depth APIs) is broadly maintained across both engines [72]. Unity tends to receive new feature releases slightly earlier, in part because Meta’s own showcase applications are predominantly Unity-based, but the gap on stable features is small. Both engines are also supported by the Meta XR Simulator for development without a headset [73]. The integration dimension of the choice therefore turns less on what is available than on how it is exposed: Unity provides multiple overlapping interaction frameworks, while Unreal offers a single integration surface aligned with OpenXR.

7.4.1 Unreal Versions and OpenXR Backend

Unreal development for the Quest 3 has two distinct paths. The first is stock Unreal Engine from the Epic Games Launcher with the Meta XR plugin added separately. The second is a custom build of Meta’s Oculus-VR fork of the engine, which adds Quest-specific rendering optimizations (uniform-buffer LightGrid packing, Emulated Uniform Buffers, an Adreno-specific occlusion query path) and exposes experimental features ahead of the standalone plugin [74], [75]. The fork’s cost is significant: cloning, generating project files, and compiling the engine takes hours, and its release cadence does not align with Epic’s hotfixes.

Initial development for this project used stock Unreal Engine 5.6 with version 117 of the Meta XR plugin, but was migrated to the fork to access soft occlusion through the Depth API, which requires engine-level patches not present in the standalone plugin [75], [76]. Over the project’s lifetime the Meta XR plugin matured to version 201 with Unreal Engine 5.7.4, and this update resolved a number of intermittent failures in MRUK scene loading and anchor alignment that had made the mesh-based coordinate mapping approach impractical on the earlier plugin (see Chapter 8). Soft occlusion, however, still requires the fork: the Depth API patches needed to composite virtual rays correctly behind physical geometry remain fork-exclusive at the time of writing. The fork therefore continues to be a hard dependency for this project, even though the broader pattern across the version history is that fork-exclusive features migrate progressively into the standalone plugin and eventually into Epic’s mainline. Application Space Warp, for

example, is now reachable from Unreal Engine 5.7 onwards through the native `xr.OpenXRFrameSynthesis` CVar without requiring the fork [75].

A related choice is the XR API backend exposed by the Meta XR plugin: Epic’s Native OpenXR plugin (with Meta features layered as extensions) or the legacy Oculus OVRPlugin. Meta designates the Native OpenXR plugin as the recommended backend, with OVRPlugin retained for compatibility [77]. Feature coverage between the two is broadly equivalent for this project’s needs. The Native OpenXR backend was selected for forward compatibility, since improvements made to Epic’s OpenXR plugin propagate without Meta-specific migration.

7.5 Conclusion

Unreal Engine was selected for this project. The decisive factor was frame-time consistency under XR rendering load: Unity reaches higher peak frame rates on lightweight scenes, but Unreal sustains tighter variance under sudden load, which matters more than peak performance at a fixed display refresh rate. Unreal’s deterministic memory model gives the developer direct control over allocation and lifetime for the large number of objects updated each tick, avoiding the non-deterministic GC pauses inherent to a managed runtime. Together with the tight integration of Quest-specific rendering optimizations through the Meta XR plugin and the alignment with the OpenXR direction, these factors made Unreal the more defensible choice.

The trade-off is a higher initial cost in language complexity and engine configuration than Unity would have imposed. This cost is one-time, and was judged acceptable in exchange for the runtime characteristics required for sustained XR comfort. The maturation of the Meta XR plugin observed during the project’s lifetime, in which features once exclusive to a custom-built engine fork have moved into the standard distribution, further reduces the long-term maintenance cost of this choice.

With the engine selected, the remaining chapters of this part describe how the application is built on top of it. The next concern is how the rendering architecture handles the large per-tick ray workload that motivated the deterministic-memory argument above. This is taken up in Chapter 10, where the trade-offs between Unreal’s available rendering paths are examined against the project’s specific constraints. The networking layer that feeds those rays into the engine is covered separately in Chapter 9.

Coordinate Mapping

For the application to correctly visualize signal data in physical space, the headset and the server must share a common spatial reference frame. The server holds a pre-existing mesh annotated with signal positions, the headset must be able to express every point in that plan as a location in the physical room it is currently rendering. Establishing this shared frame, and keeping it accurate enough for the visualized data to align with the real environment, is the central technical challenge addressed in this chapter.

Two successive approaches were developed. The first relied on the user manually placing spatial anchors to define a local coordinate system, which the server would then map onto its own mesh. The second replaced that manual step with the scene mesh provided by the MRUK, achieving a substantially lower positioning error. The remainder of this chapter describes each approach in turn, explains why the first was insufficient, and concludes with a summary comparison.

8.1 Background: Spatial Data on Meta Quest 3

The Meta Quest 3 exposes three complementary types of spatial data to developers: scene data, mesh data, and depth data [78]. Scene data is a simplified semantic model of the room (walls, floor, ceiling, and labeled furniture) that provides structural awareness of the environment. **Mesh data**, accessed through the Mesh API, goes further: it reconstructs the physical room as a single triangulated mesh that captures the geometry of all scanned surfaces, including irregular shapes and furniture contours [79]. Lastly, depth data provides a per-frame depth map derived from the headset’s dedicated depth sensor, which is hardware that the Quest 3 introduced in the Quest line and enables features such as real-time dynamic occlusion [80].

The scene mesh is generated by Meta’s Space Setup process, during which the user scans their room by looking around. This provides “a far more granular representation of a physical environment” than was possible on earlier hardware [79]. The mesh is stored on the device and can be queried by applications at runtime. Importantly for the project, the mesh and the headset’s tracking system share the

same internal coordinate frame: positions extracted from the mesh are expressed in the same space as the headset's world origin, which UE5 inherits through the MetaXR plugin. This property is the foundation of the second mapping approach.

8.2 Approach 1: Manual Spatial Anchors

8.2.1 Overview

Meta's Spatial Anchor API allows applications to create world-locked reference points that persist across the tracking session [81]. A spatial anchor records a position and orientation in the headset's tracking space, virtual content attached to an anchor will remain at the correct physical location even as the user moves around or the tracking system makes small corrections.

The first mapping approach used this API to let the user manually define a local coordinate system at application startup. Before any data could be visualized, the user was prompted to place four anchor points on the physical floor. The first anchor defined the origin of the local frame and was assigned coordinates $(0, 0, 0)$. The second anchor was positioned in the intended forward direction from the origin, thereby defining the positive X -axis, while the third anchor defined the positive Y -axis through placement along the intended rightward direction. A fourth anchor completed the rectangular mapped area and provided an additional spatial reference for the server-side alignment process.

Figure 8.1 illustrates the spatial anchor placement strategy used in the first approach. The anchors define a local horizontal coordinate frame aligned with the physical room, which is subsequently mapped onto the server-side floor plan representation.



Figure 8.1: Manual spatial anchor placement in the room.

All four anchors shared $Z = 0$, as the headset is able to determine ground level independently, reducing the coordinate mapping problem to two horizontal axes. Once placed, the application recentered the Unreal pawn (the virtual actor representing the user's position and headset tracking) so that this anchor-defined frame

coincided with the Unreal world axes. From that point on, any world-space coordinate in Unreal was directly comparable to a position in the server's coordinate system, with only a unit scale conversion required.

Rather than exporting a mesh from the headset, this approach sent the four anchor positions directly to the server. The server located the center of its own floor plan mesh and mapped the four anchor positions onto that mesh, exploiting the fact that all four anchors were intended to correspond to corners of the room. This gave the server enough information to derive a rough spatial registration without requiring a full mesh export from the headset.

8.2.2 Ordering Constraint

The server imposed an important constraint: the first, second, and third anchors always had to correspond to the same physical corners of the room. If the user placed them in a different rotational order, the resulting axis frame would be rotated by 90° or 180° relative to the expected orientation, causing visualized signals to appear on incorrect walls. This meant the application had to enforce a strict anchor placement sequence and clearly communicate that sequence to the user, adding friction to the startup process.

8.2.3 Accuracy Limitations

This approach compounded two independent sources of error. The first source of error came from how precisely the user could physically place each anchor at the intended corner. When using a controller, the user must align the controller's center as accurately as possible with the corner point, which is difficult to achieve, especially in a cluttered or constrained environment. The second source was the server-side mapping: projecting the four anchor positions onto the center of the floor plan mesh is an approximation, and any deviation between the true room geometry and the server's stored mesh contributes additional misalignment. In practice, positional offsets of up to 10 cm between an anchor and its intended physical location were observed, and the server-side mapping step could add further error on top of that.

A third, more subtle problem arose from the interaction with MRUK world-locking. To allow the pawn recentering described above to function correctly, MRUK world-locking had to be disabled for this approach. World-locking is the mechanism by which MRUK continuously corrects small drifts in the headset's tracking estimate, keeping scene anchors aligned with the physical environment over time [82]. With it disabled, the Unreal world origin could drift relative to the physical room during a session, causing the registered coordinate frame to shift even after a successful initial calibration.

8.3 Approach 2: Scene Mesh Alignment

8.3.1 Overview

The second approach eliminated manual anchor placement entirely by using the room mesh exposed by the Meta MRUK. MRUK is a higher-level library built on top of the Scene API that provides ready-made utilities for working with scene understanding data in UE5 [82]. It exposes the scanned room geometry through its `AMRUKRoom` and `AMRUKAnchor` class hierarchy, with the complete triangulated scene mesh accessible via the anchor tagged with the `GLOBAL_MESH` label. Scene data is loaded at runtime by calling `LoadSceneFromDeviceAsync` on the `UMRUKSubsystem`, after which the mesh and all labeled surface anchors are available as Unreal components [82].

MRUK's availability in a stable, production-ready state was a practical prerequisite for this approach. From version 65 onward, MRUK was folded directly into the MetaXR plugin rather than being distributed as a separate package [83], and it was the transition to version 201 of the Unreal integration [84] that made MRUK reliably usable for this project. Earlier plugin versions (v115-v117) used during development did not provide reliable mesh retrieval, and the functionality could not be made to work at all prior to the update to version 201 of the Unreal integration.

One practical constraint introduced by this approach is that the application must run as a standalone build deployed directly to the headset. MRUK's `LoadSceneFromDeviceAsync` could not be made to work over an Oculus Link connection (the wired or wireless mode that streams PC-rendered content to the headset) in the plugin versions used for this project (MetaXR v201), despite enabling the Spatial Data over Meta Horizon Link option in the Link settings. Meta's documentation indicates that a previously captured scene model can in principle be accessed over Link with this setting enabled [78], but this could not be reproduced in practice. This means the MRUK mesh approach could not be iterated on through the editor over Link in the same way as purely logic-based features.

8.3.2 Mesh Extraction and Coordinate Transformations

Because MRUK loads the scene mesh into the same Unreal world space as the headset's tracking origin, the mesh vertices and the default Unreal world axes are guaranteed to be expressed in a consistent spatial frame. The application extracts the vertex and index data from the `UProceduralMeshComponent` associated with the `GLOBAL_MESH` anchor and serializes them together with the Unreal world-axis orientation for transmission to the server. Before serialization, three transformations are applied to convert the data from Unreal's native representation to the coordinate conventions expected by the server.

Figure 8.2 illustrates the data exported from the headset to the server, including the reconstructed scene mesh and the associated axis frame used for coordinate transformation.

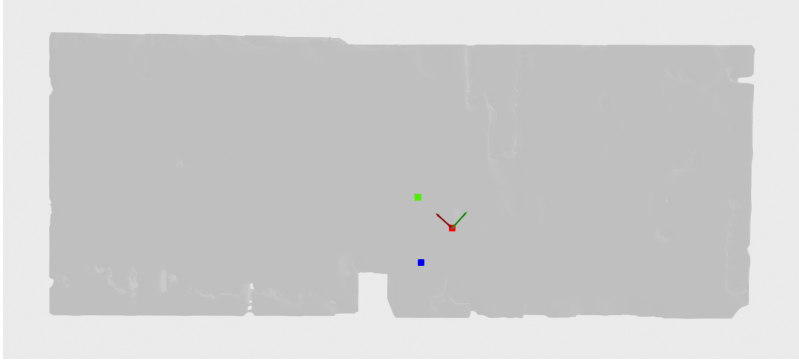


Figure 8.2: Scene mesh and axis frame export.

Scale: Centimeters to Meters UE5 stores world positions in centimeters, whereas the server operates in SI units (meters). Every vertex position and anchor translation is therefore multiplied by 0.01 before export:

$$\mathbf{v}_m = \mathbf{v}_{cm} \times 0.01 \quad (8.1)$$

Handedness: Left-Handed to Right-Handed UE5 uses a left-handed coordinate system in which $+X$ points forward, $+Y$ points right, and $+Z$ points up. Standard mathematical libraries and the server use a right-handed convention. To convert between the two, the sign of the Y component is negated while X and Z are preserved:

$$\begin{pmatrix} X_{\text{export}} \\ Y_{\text{export}} \\ Z_{\text{export}} \end{pmatrix} = \begin{pmatrix} X_{\text{UE}} \\ -Y_{\text{UE}} \\ Z_{\text{UE}} \end{pmatrix} \quad (8.2)$$

Winding Order: Clockwise to Counter-Clockwise Reflecting a single axis (the Y negation above) inverts the orientation of every triangle face, because the cross product used to compute surface normals reverses sign under an odd-axis reflection. In Unreal’s left-handed space, triangle vertices are wound clockwise when viewed from outside the surface, in right-handed space, the convention is counter-clockwise. If left uncorrected, the server would receive a mesh in which all normals point outward into empty space rather than inward into the room. The fix is to swap the second and third index of every triangle:

$$\text{Original: } [i_0, i_1, i_2] \quad (8.3)$$

$$\text{Exported: } [i_0, i_2, i_1] \quad (8.4)$$

Floor Normalization and Axis Sign Correction Two additional normalization steps ensure clean alignment between the exported mesh and the server’s stored floor plan. First, the minimum Z value across all mesh vertices is subtracted from every vertex, placing the floor at $Z = 0$. Second, three Unreal

world-space points at the fixed coordinates $(0, 0, 0)$, $(1, 0, 0)$, and $(0, 1, 0)$ are exported alongside the mesh as a validation reference, as shown in Figure 8.2: they encode the Unreal world origin, forward direction, and right direction respectively, allowing the server to reconstruct the headset axis orientation relative to the mesh. These points have their Y component negated, consistent with the handedness correction applied to the mesh vertices above.

Inverse Transformation on Inbound Data When the server returns layout object positions to the headset, the transformations described above are inverted in sequence: coordinates are scaled back up by a factor of 100, the Y component is re-negated to restore left-handed space, and the yaw (rotation around Z) is negated so that orientation aligns correctly with the physical room.

8.3.3 Server-Side Transformations

On the server side, receiving the mesh and axis frame enables two successive transformations to be computed. The first is a mesh-to-mesh alignment between the received headset mesh and the server’s pre-existing floor plan, this process aligns the two geometric representations of the same physical room and is described in detail in Section 4.3. The second transformation maps from the server’s original internal axis system to the Unreal world axis system embedded in the exported mesh. Any signal coordinate expressed in the server’s original space can then be mapped into Unreal world space by composing these two transformations, and the result is sent back to the headset for rendering at the physically correct location.

This design means the headset side has no knowledge of the server’s coordinate conventions and requires no per-session calibration. The server owns the alignment computation entirely, and the headset simply provides the geometric data needed to make it possible.

8.3.4 Accuracy

The residual positioning error of this approach was measured to be at approximately 1 cm to 5 cm. Unlike the manual anchor approach, this error does not depend on user actions, it arises from two remaining sources. The first is the geometric fidelity of the Quest 3 scene mesh itself: the mesh is captured during Space Setup and represents a triangulated approximation of the real room surfaces, so any deviation between the mesh and the true wall geometry contributes directly to misalignment. The second source is the quality of the server-side mesh-to-mesh alignment, which is bounded by how closely the two meshes correspond to one another in practice.

It is also worth noting that the mesh can exhibit a slight rotational drift relative to the physical walls, particularly immediately after the headset is put on. This drift tends to decrease as the headset accumulates more tracking data through walking around, since the inside-out tracking system refines its spatial map over time. Research into Quest 3 tracking precision has demonstrated that the headset’s inside-out tracking achieves sub-millimeter positional repeatability under con-

trolled conditions [85], confirming that the dominant error source in this approach is mesh reconstruction quality rather than the tracking system itself. With MRUK world-locking enabled, small tracking corrections are applied continuously throughout the session, so any initial drift reduces as the user moves through the space.

8.3.5 Limitations of the Quest Spatial System

While the second approach removes user interaction from the calibration process and improves robustness, it remains constrained by limitations of the underlying Quest spatial mapping system. A first limitation arises from the stability of room recognition across similar environments. When the headset is initialized in a space that closely resembles a previously scanned room, or when it is moved between rooms with only minor geometric differences, the spatial tracking system can exhibit reduced reliability in maintaining a consistent world origin. In practice, this may result in temporary ambiguity in position estimation until sufficient environmental features are re-observed or the headset is restarted.

A related limitation is observed in the MRUK scene selection process during Spatial Setup. When multiple scanned rooms share similar structural characteristics, such as comparable wall layouts or proportions, the system can struggle to reliably distinguish and select the correct scene mesh, and in some cases may fail to select a mesh altogether. This can lead to inconsistent loading of scene data, particularly in environments where multiple spaces are scanned with only subtle differences.

A final practical constraint is imposed by the available application memory budget on the device, which is 5.75 GiB [54]. Large or geometrically complex environments place significant pressure on this limit during scene capture and processing. In the test setup, a room of approximately 6 m by 14 m required a relatively fast scanning procedure to prevent memory exhaustion during mesh generation. This indicates that large-scale or highly detailed environments may require careful scanning behavior to remain within the memory constraints of the platform.

8.4 Summary

Table 8.1 contrasts the two approaches across the dimensions most relevant to the project.

The transition from Approach 1 to Approach 2 removed the user-facing calibration step entirely, reduced the worst-case positioning error by an order of magnitude (from 10 cm or more to 1 cm to 5 cm), and eliminated both the rotational ambiguity from anchor ordering and the session drift caused by disabled world-locking. The remaining error is bounded by the quality of the Quest 3 scene mesh and the server-side alignment, both of which are fixed characteristics of the hardware and algorithm rather than variable user-dependent factors, making Approach 2 a substantially more robust foundation for the rest of the system.

Table 8.1: Comparison of coordinate mapping approaches.

Criterion	Approach 1 (Anchors)	Approach 2 (MRUK Mesh)
User setup required	Yes, 4 points per session	Initial scan of room
Positional accuracy	10 cm or more	1 cm to 5 cm
Rotation ambiguity	Possible (corner order matters)	None
Error sources	Placement precision + server mesh mapping	Scene mesh fidelity + mesh-to-mesh alignment
World-locking	Disabled (risk of session drift)	Enabled (continuous correction)
Build requirement	Link or standalone	Standalone only
Plugin dependency	Spatial Anchor API	MRUK

Signal Data Reception and Buffering

For visualization to be useful in practice, the headset must not only render the correct signals in the correct positions. It must also allow the user to navigate through time. A single live view is insufficient when the goal is to analyze how signal propagation changes as a user moves through a space. This chapter describes how measurement frames are received from the server, stored in a timestamped buffer, and made available for both live display and retrospective playback.

9.1 Wire Protocol

Data arrives from the server as a continuous stream of UDP datagrams. Each datagram encodes a single measurement snapshot and consists of a fixed-length header followed by a variable number of ray records. The header, referred to as **FWireHeader**, carries a magic sentinel value used for validation, a monotonically increasing sequence number, a timestamp index that uniquely identifies the measurement instant, and the current position of the user equipment expressed in the server's coordinate frame. Each ray record, **FWireRay**, stores the ray's origin and endpoint, its type (e.g. line-of-sight or single-bounce), the antenna panel ID, a propagation distance, and an attenuation value.

The timestamp index rather than the sequence number serves as the primary key for the buffer. The sequence number increments with every datagram sent, including retransmissions, whereas the timestamp index identifies the physical measurement instant. Keying on the timestamp index therefore provides natural deduplication: if the same snapshot is retransmitted and arrives twice, only the first occurrence is stored.

9.2 UDP Reception

One consequence of draining the socket during the engine's per-frame update callback (**Tick**) is that the effective receive rate is coupled to the application's frame rate. On the Quest 3 rendering at 90Hz, at most 90 batches of datagrams are processed per second, regardless of the transmission rate of the server. In practice,

this does not impose a limitation on the system. Under high transmission rates during testing on a desktop client, no packet backlog was observed, and received updates remained aligned with the headset’s rendering frequency.

9.3 Frame Buffer and Memory Limitations

Received frames are stored in a parallel data structure: a hash map keyed on the timestamp index that holds the full frame data, and a sorted array of timestamp indices that maintains insertion order for efficient sequential access.

In the current implementation, this buffer is designed for finite measurement sessions. Rather than evicting data, the buffer dynamically expands its maximum capacity when a new, higher timestamp index is received. While this ensures that the entire dataset remains available for retrospective playback, it introduces a strict memory limitation governed by the hardware of the Meta Quest 3.

The headset features 8 GB of unified memory, of which 5.75 GiB [54] is available to the application, shared between the Unreal Engine runtime and the application’s own buffers. The wire protocol is highly structured: each `FWireHeader` requires 28 bytes, and each `FWireRay` requires 44 bytes. Consequently, a single dense measurement frame containing 1,000 rays consumes approximately 44 KB of raw data. While the per-frame footprint is relatively small, allowing a 10-minute session at 100 frames per second would still accumulate approximately 2.64 GB of raw data. Furthermore, the overhead associated with Unreal Engine’s `TMap` [86] and `TArray` [87] data structures substantially increases memory consumption. As the session continues, the indefinite expansion of these structures will eventually lead to memory exhaustion and an out-of-memory (OOM) crash.

Maintaining a stable playback index across this growing buffer requires careful bookkeeping. When a new frame is inserted before the current playback position in the sorted order, the index is incremented to keep it pointing at the same timestamp. This ensures that the frame currently on screen does not change silently as late-arriving data is inserted.

9.4 Playback Modes

The buffer supports two operating modes that can be toggled at runtime.

In live mode, the playback index is always updated to point at the most recently received frame. Display updates are driven by a frame-rate accumulator: rather than rendering every frame as it arrives, the accumulator advances at a configurable rate and triggers a display call when it crosses the threshold. This decouples the visual update rate from the datagram arrival rate, preventing the display from flickering when bursts of datagrams arrive close together.

In playback mode, live tracking is suspended, the user can navigate the buffer freely. Automatic playback advances the index at the same accumulator-driven rate as live mode but steps through stored frames sequentially. Step and jump functions

allow frame-by-frame navigation or direct seeking to a specific timestamp. In both cases the accumulator is reset on any manual navigation action, preventing a stale accumulated delta from causing an immediate skip on the next tick.

When neither live mode nor automatic playback is active, no accumulator runs and the display is frozen on the current frame until the user explicitly steps or jumps. This frozen state is the default entry point for post-session analysis.

Ray Rendering Architecture

With measurement frames available in the buffer, each frame must be translated into geometry that the engine can draw. This chapter describes the rendering subsystem responsible for that translation: the approaches considered, the design chosen, and the implementation details that make it perform within the strict frame budget of a standalone VR headset.

The rendering challenge is not simply one of drawing static lines. The dataset is highly dynamic. Frames arrive continuously, and the set of MPCs estimated by the SAGE algorithm changes with every timestamp. Individual rays must be selectable and hoverable with precise, low-latency hit detection. Furthermore, color must accurately encode signal properties such as attenuation or antenna panel ID without causing visual clutter. All of this data processing and geometry generation must execute within the 11.1 ms frame budget imposed by the Meta Quest 3's 90Hz refresh rate, with no synchronous stalls or unpredictable GC pauses that would cause perceptible XR judder. Because the rendering pipeline is implemented in C++ without dynamic `UObject` churn, it remains outside the scope of Unreal's GC, which only collects unreferenced `UObject`-derived instances and does not sweep raw C++ allocations.

10.1 Evaluation of Rendering Approaches

10.1.1 Overview and Hardware Constraints

Within UE5, several rendering paradigms exist for visualizing dynamic lines: component-based splines, GPU-based particle systems, instanced meshes, procedural geometry, and custom primitive rendering. Debug line helpers (`LineBatchComponent`, `DrawDebugLine`) were excluded from the outset: they rebuild the entire batch from scratch every frame, carry no per-line metadata for hit detection, and are explicitly not intended for production rendering.

However, the target platform imposes a strict hardware reality. While the Quest 3's Snapdragon XR2 Gen 2 SoC provides capable GPU performance, it remains severely bottlenecked by draw calls and thermal limits compared to a tethered

desktop system. A successful rendering architecture must therefore maintain a deterministic draw call count, minimize CPU-GPU memory transfers across the system bus, and avoid synchronous data structure rebuilds.

10.1.2 Component-Based Splines

UE5's native approach to drawing thick, curve-based lines is the `USplineComponent` paired with a `USplineMeshComponent` [88]. While visually robust for static levels, this approach fundamentally fails at scale for dynamic data due to component overhead.

In Unreal's object model, each `USceneComponent` carries costs in registration (which creates a render proxy and physics state), transform propagation through the scene hierarchy, and render-state updates on every transform change. Scene components do not tick by default, but these structural costs alone are prohibitive at scale: representing thousands of dynamic rays would require spawning and managing thousands of individual components, consuming the game thread frame budget in bookkeeping before any rendering even occurs. This approach was immediately ruled out.

10.1.3 GPU-Based Particle Rendering (Niagara)

The project went straight to Unreal's Niagara particle framework to test the early synthetic CSV dataset. Niagara supports both CPU and GPU simulation paths; with the GPU path enabled, simulation and rendering are offloaded entirely to the GPU, making it highly efficient for massive element counts. Each ray was successfully represented as a stretched ribbon or beam particle using the GPU simulation path.

However, the system failed when moving from passive visualization to active interaction. Precise hit detection, identifying exactly which ray the user's controller is pointing at, requires reading per-particle identification data back from the GPU to the CPU. In Unreal, this readback is asynchronous [89] and introduces at least one frame of delay; the render thread runs one frame behind the game thread [90], and the RHI thread adds a further translation stage between rendered commands and the graphics API, compounding the total end-to-end latency. In XR, a multi-frame delay between physical hand movement and visual selection feedback causes a disconnect that violates basic user experience guidelines. Furthermore, outputting unique IDs per particle would require an additional full-screen rendering pass to write per-particle identifiers, a custom buffer that would consume significant memory bandwidth on the Quest 3's mobile tile-based renderer.

This interaction failure points to a deeper architectural mismatch. Niagara is a stateful simulation system in which the GPU owns and evolves particle positions frame by frame. The rays in this system require no such simulation: they arrive pre-computed from the backend and are drawn as-is each frame. The full simulation pipeline (double-buffered GPU particle state, per-module shader permutations, and spawn-and-death bookkeeping) executes every frame for geometry that is simply a fixed set of line segments, and it is precisely this GPU ownership

of state that makes per-ray hit detection expensive. The hit detection problem is therefore not an isolated shortcoming of Niagara but a symptom of applying a simulation abstraction to stateless, pre-computed data.

10.1.4 Instanced Mesh Rendering

Instanced rendering batches multiple copies of the same mesh (e.g., a simple cylinder) into a single draw call, with per-instance transforms stored in a GPU buffer. This reduces CPU overhead substantially compared with individual actors. Unreal's `UHierarchicalInstancedStaticMeshComponent` (HISM) [91] further optimizes this by building a Bounding Volume Hierarchy (BVH) for aggressive frustum and occlusion culling.

However, the approach has two drawbacks for this use case. First, updating instance transforms on every timestamp triggers per-instance render-state dirtying and data re-upload. The cluster-tree rebuild is performed asynchronously via a background task (`BuildTreeAnyThread`), but the tree is marked stale on every update and culling accuracy degrades until the rebuild completes. Second, and more fundamentally, accurate hit detection on thin, line-like primitives in stereo space is not achievable through the instanced mesh interface: the API exposes only an instance index with an approximated bounding-box hit, not a precise geometric correspondence. This argument applies equally to plain `UInstancedStaticMeshComponent`, which forgoes the cluster tree entirely but is otherwise identical for interaction purposes. Hit detection alone is therefore sufficient to rule out the entire instanced mesh family.

10.1.5 Procedural Mesh Components

Procedural mesh components [92] explicitly construct geometry on the CPU by writing vertex and index buffers that are then uploaded to the GPU. This provides complete control over the resulting geometry but introduces a severe per-update cost.

Any change to the ray set requires rebuilding the affected vertex data and transferring it across the CPU-GPU boundary. For a dataset that changes on every displayed frame, this continuous, unamortized data transfer is highly inefficient. Furthermore, dynamically updating procedural geometry frequently triggers synchronous collision cooking and generates substantial transient data. As a result, this constant reallocation introduces unpredictable frame spikes from synchronous collision cooking and repeated CPU-GPU transfers.

10.1.6 Custom Primitive Rendering

The custom primitive approach implements a `UPrimitiveComponent` [93] subclass paired with a `FPrimitiveSceneProxy` [94]. The scene proxy is the object the render thread holds a reference to; the component is the game-thread handle through which the application submits new data. This separation is the key architectural property of the approach: game-thread state and render-thread state are never

accessed concurrently, eliminating the need for synchronization primitives during rendering.

Ray geometry is aggregated on the game thread into a flat array of ray records, each carrying its start and end point, color, and thickness. When the array changes, it is packaged into a render command and enqueued for delivery to the proxy on the render thread. The proxy's `GetDynamicMeshElements` callback then constructs the actual draw commands from this data, generating a camera-facing quad (a four-vertex rectangle composed of two triangles) for each ray. These quads are submitted as a single draw call per view, producing one draw call for the entire ray set regardless of how many rays are visible.

Hit detection is handled entirely on the game thread by computing the closest distance between each ray segment and the cursor's world-space ray. This is an $O(N)$ linear scan over the ray array that requires no GPU readback and no intermediate render pass, making it both exact and deterministic.

10.1.7 Comparative Summary

The custom primitive approach was selected on the basis of its deterministic draw call count, its support for exact hit detection, and its clean separation between game-thread data and render-thread execution. The implementation complexity is higher than the alternatives, but this cost is paid once at authoring time and does not recur at runtime.

10.2 Implementation

10.2.1 Game Thread and Render Thread Separation

The `URayLineComponent` class owns the canonical ray array on the game thread. In UE5, the render thread typically executes one to two frames behind the game thread. Passing direct memory pointers between these threads to update dynamic geometry would inevitably cause race conditions, as the game thread might overwrite ray data for the current frame while the render thread is still drawing the previous one.

To resolve this, when new ray data is submitted via `SetRays`, a copy of the array is packaged into an `ENQUEUE_RENDER_COMMAND` macro [90] and forwarded to the proxy. This acts as a thread-safe boundary, enforcing a strict double-buffering paradigm. The proxy stores its own isolated copy that the render thread reads during `GetDynamicMeshElements`. Neither copy is ever read by the other thread, so no locks are required during rendering; the only synchronization is the render command queue itself, which the engine manages natively.

This architecture also facilitates payload minimization. Color state, which ray is selected and which is hovered, is maintained separately from structural geometry. When selection or hover state changes, pushing a completely new vertex buffer across the CPU-GPU bus would unnecessarily starve memory bandwidth. Instead, only the minimal color array is rebuilt and forwarded to the proxy via a second

Criterion	Niagara (GPU)	Instanced mesh	Procedural mesh	Custom primitive
Draw calls	Low	Low (batched)	One per update	One per frame
Dynamic updates	Highly efficient	Costly at scale	Full buffer rebuild	Render command only
Hit detection	High latency	Approximated	CPU segment test	CPU segment test
Thread safety	GPU managed	Engine-managed	Game thread only	Explicit double-buffer
XR GPU suitability	Poor (proxy cost)	Good	Moderate	Good

Table 10.1: Comparison of rendering approaches for ray visualization.

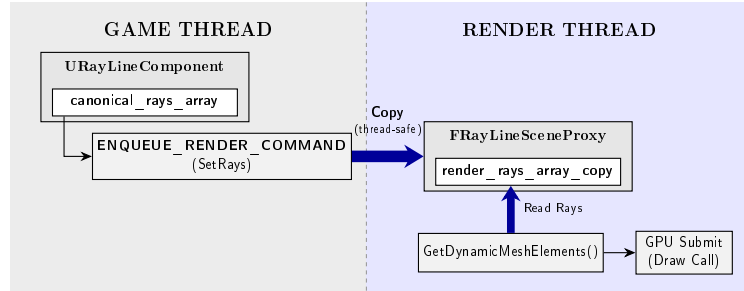


Figure 10.1: Thread-safe double-buffering architecture showing data transfer across the render command boundary.

render command, avoiding a full geometry rebuild for what is effectively a single-integer state change.

10.2.2 Geometry Generation and Billboarding

Each ray is rendered as a camera-facing quad, a technique known in computer graphics as billboarding: a flat polygon is continuously rotated so that it always faces the viewer, giving the visual appearance of a volumetric object regardless of viewing angle. The decision to use billboarding rather than true 3D meshes is driven strictly by the vertex budget. If each ray were rendered as an 8-sided open cylinder (a tube without end caps), a single line segment would cost 16 triangles, a closed cylinder with end caps costs 32. For a dense dataset of 10,000 rays, this translates to 160,000–320,000 triangles per draw submission. By utilizing camera-facing quads, each ray costs exactly 2 triangles, reducing the geometric payload by at least 87.5% (open cylinder) and up to 93.75% (closed cylinder), keeping the thermal load well within the Quest 3’s operating limits.

The quad’s width is taken from the per-ray thickness field, which allows selected and hovered rays to be drawn thicker than the default without requiring separate geometry passes. Camera-facing geometry generation is handled directly within the C++ render thread via the custom scene proxy [94]. The facing direction is computed per-ray on the CPU by taking the cross product of the ray direction and the vector from the ray’s midpoint to the active view location; the resulting camera-oriented vertices are then submitted to the rendering pipeline as dynamic mesh elements. This ensures the quads always present their full face to the viewer regardless of the viewing angle, preserving the visibility of thin rays that would otherwise disappear when viewed nearly edge-on.

10.2.3 Ray Builder and Filters

Before a frame’s wire data reaches the renderer, it passes through a builder stage that applies active filters and resolves per-ray colors. Three independent filters can be active simultaneously: an antenna filter that restricts display to a specific subset of sensors, a ray type filter that shows only the selected types of rays,

and a maximum distance filter that hides rays whose total path length exceeds a threshold.

Single-bounce rays require special handling because their path is not fully described by the origin and endpoint fields alone. The second segment runs from the recorded bounce point to the current position of the user equipment, which is known only at display time. The builder therefore reads the latest equipment position and constructs the second segment dynamically when it encounters a single-bounce ray record.

For visual clarity within the headset, maintaining a neutral, highly readable color scheme is paramount. Relying heavily on bright reds and greens in XR can cause severe visual fatigue and color-clashing against the physical room passthrough. Therefore, color is resolved using a neutral, blue-and-black dominant palette. Attenuation and path distance map to a dark-blue-to-cyan gradient, ensuring the visualization remains stark and professional.

When assigning distinct colors to specific ray identities or antennas, the HSV color space is utilized instead of standard RGB. Randomizing IDs using RGB risks generating "dead" colors (such as muddy browns) that vanish against the real-world background. By hashing the identity strictly into the Hue (H) channel while carefully clamping the Saturation (S) and Value (V), the system mathematically guarantees that every ray is legible without resorting to aggressive or fatiguing neon tones.

10.2.4 Selection and Metadata

When the user points their controller at the scene, the cursor's world-space position and direction are tested against every visible ray segment using a closed-form segment-to-ray distance formula. The ray whose closest point of approach falls within a configurable tolerance and is nearest overall becomes the hovered ray. A controller trigger confirms the selection.

In a traditional physics engine, collision detection relies on spatial partitioning structures like a BVH or an octree. However, implementing an $O(N)$ linear scan for hit detection was chosen deliberately over hierarchical trees due to the highly dynamic nature of the dataset. Spatial trees are only performant if the underlying data remains mostly static, allowing the construction overhead to be amortized over multiple frames. Because the set of measurement rays in this project changes entirely on every single frame, the CPU would spend significantly more time synchronously rebuilding the BVH structure than it would brute-forcing the geometric calculations. For datasets of this scale, sequential linear scanning of contiguous memory on modern mobile processors is exceptionally fast, lightweight, and entirely deterministic.

Selected and hovered rays are drawn with a distinct color and increased thickness. Because thickness is embedded in the per-ray data submitted to the renderer, changing which ray is selected triggers a full `SetRays` call rather than just a color update. This is a deliberate trade-off: the geometry rebuild adds a small cost

on selection events, but it keeps the renderer’s data model uniform: every visual property of a ray lives in the same flat record.

Each visible ray also has an associated metadata record that stores the antenna panel ID, ray type, attenuation, measured propagation distance, and the sequence and timestamp indices of the frame it came from. When a ray is selected, this record is broadcast via a delegate to any User Interface (UI) component that has registered interest, allowing a separate widget to display the signal properties without coupling the renderer to any specific UI implementation.

10.3 Summary

The rendering system successfully translates buffered measurement frames into visible geometry through a strictly unidirectional, double-buffered pipeline. By migrating away from native components and GPU-bound particle systems, the custom primitive architecture navigates the severe hardware constraints of the Meta Quest 3. The ray builder filters and colors the wire data efficiently, and the custom scene proxy submits it to the render thread as a single batched draw call, achieving an $O(1)$ draw call footprint regardless of the dataset’s density.

Crucially, separating game-thread ownership from render-thread consumption eliminates synchronization overhead during rendering. Meanwhile, performing hit detection entirely on the game thread via cache-coherent linear scanning ensures that ray selection remains exact, bypasses the latency of GPU readbacks, and avoids the CPU penalty of synchronous BVH generation.

The result is a highly robust rendering architecture whose per-frame memory and processing costs are entirely predictable. It successfully isolates the visualization pipeline from the volatility of incoming UDP datagrams, ensuring that even under heavy measurement loads, the user experiences smooth, judder-free spatial interaction.

User Interface and Interaction

The AR application exposes a head-up display and a set of runtime controls that serve two distinct purposes: communicating system state to the user and allowing the user to configure what is visualized. This chapter describes the structure of the Head-up Display (HUD), the controls it exposes, how playback is operated from within it, and the communication pattern used to connect the UI to the rest of the application.

11.1 HUD Architecture

The interface is implemented as a set of world-space widgets attached directly to the VRPawn camera rig (the Unreal actor class representing the user's head position and motion controllers). Because all widgets are parented to the camera, the widgets float, like a HUD, at a fixed position in the user's field of view regardless of head movement. To keep a good overview only one widget is visible at a time, a cycling system on the pawn steps through them, keeping the implementation simple without requiring a dynamic widget loading mechanism. Navigation between panels is performed with the same laser pointer used for ray selection, described further in Section 11.4.

This single-widget-at-a-time approach has a useful performance property. Widget construction and layout rebuilding are comparatively expensive operations, and keeping all widgets resident but hidden means those costs are paid once at startup rather than on every panel switch. State updates to the visible widget, slider value changes and text readouts, are lightweight and can run every tick without meaningful frame budget impact.

11.2 Configuration Controls

Figure 11.1 shows the settings panel as it appears in the headset. The panels expose the following runtime configuration:

Connection. The server IP address and port are set before the application starts

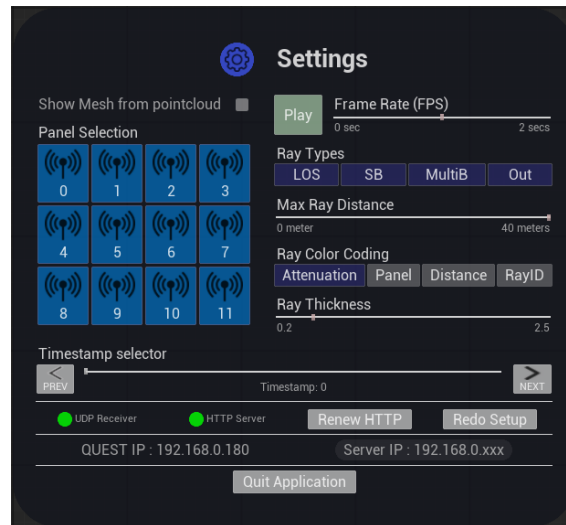


Figure 11.1: The in-headset settings panel.

but can be updated from this panel while the application is running, triggering a socket reconnect without requiring a restart.

Antenna panel selection. Each antenna panel known to the system is represented by a toggle. The panel is constructed dynamically from the panel identifiers received over UDP, so it expands automatically if a session grows to include more panels later on. Toggling an antenna calls the ray builder’s antenna filter, described in Chapter 10, which takes effect on the next displayed frame.

Ray type and distance. Several toggles to restrict the types of rays that get shown. A slider sets the maximum total path length, hiding rays that exceed the threshold. Both controls feed the same builder filter stage as the antenna toggles.

Color mode. A selector switches between the four color mappings available in the ray builder: attenuation gradient, antenna panel ID, path distance gradient, and ray identity.

Scene mesh visibility. A toggle shows or hides the room mesh reconstructed from the MRUK scene data.

Playback. Described separately below.

11.3 Playback Controls

The playback panel surfaces the buffer navigation described in Chapter 9. It exposes three controls: a play/pause toggle, a frame-rate slider that sets the accumulator target rate, and a seek slider that maps the full range of buffered timestamps onto a single scrub bar, allowing the user to jump directly to any stored frame.

Manual interaction with either slider resets the accumulator to prevent an unintended skip on the next frame.

11.4 Laser Pointer and Interaction

The controller laser pointer serves a dual role: it selects rays in the scene and drives the HUD. A boolean flag on the pawn tracks whether the HUD is currently active. When the flag is set, hit-test priority is given to the widget and ray selection logic described in Chapter 10 is suppressed, preventing accidental ray hovers while navigating menus. When the HUD is dismissed the flag clears and the pointer resumes its scene interaction role.

11.5 Communication Pattern

Unreal's conventional approach to widget-to-actor communication is direct casting: obtain a typed reference to the target actor and call its functions directly. This is straightforward but creates a compile-time dependency between the widget and the concrete actor class it talks to, meaning any change to the actor interface requires the widget to be updated in tandem.

Blueprint Interfaces decouple this relationship [95]. Rather than calling functions on a specific class, a widget calls an interface function on whatever object it has been given a reference to. The object on the other end implements the interface and handles the call; neither side needs to know the concrete type of the other. In this system interfaces carry traffic in both directions: control inputs from the UI to the VRPawn/UDP Receiver and simulation state updates from the VRPawn/UDP Receiver back to the UI. The ray selection metadata delegate introduced in Chapter 10 follows the same principle: the renderer broadcasts a selection event without any knowledge of which widget, if any, is listening.

The practical consequence is that each UI panel can be developed and modified independently of the systems it controls, and the renderer remains decoupled from any specific UI layout.

Hardware Platform Retrospective

The Meta Quest 3 was selected as the development platform for this project based on availability rather than a systematic comparison of alternatives. This chapter examines how three contemporary standalone headsets (Apple Vision Pro, HTC VIVE Focus, Pico 4 Ultra) would interact with the key technical decisions made in this thesis: the automated coordinate mapping pipeline, the UDP data reception layer, and the Depth API dependency for soft occlusion.

12.1 Apple Vision Pro

The Apple Vision Pro is built around a dual-chip M2 and R1 architecture [96]. Its micro-OLED displays (3660×3200 pixels per eye) offer substantially higher pixel density than the Quest 3 (2064×2208 pixels per eye), which would reduce the spatial aliasing visible on thin MPC ray segments at distance.

Since Unreal Engine 5.4, visionOS has been a supported target platform [97], making ARKit scene reconstruction [98] accessible from within the running Unreal application. Provided the reconstruction is performed within the same running session, the returned mesh anchors are expressed in that session's world coordinate frame, which is the same frame the headset pose is tracked in. This is directly analogous to how MRUK loads mesh vertices into the live tracking space on the Quest 3, meaning the automated coordinate mapping pipeline (RANSAC wall extraction followed by ICP registration) could be applied without regression to manual anchor placement. If the mesh were instead captured in a separate prior session and imported as a static asset, this frame relationship would be lost and manual alignment would become necessary, reintroducing the positioning errors that motivated the transition from Approach 1 to Approach 2 (Section 8.2).

The real constraint for a visionOS port is therefore soft occlusion. Apple Vision Pro is the only one of the candidate platforms that does not implement OpenXR, and its rendering runs through the RealityKit compositor on visionOS. While RealityKit provides depth compositing natively, no equivalent to the Meta Depth API is exposed through Unreal Engine's visionOS pipeline (Section 7.4.1). Without it, virtual rays would be rasterized over foreground physical objects, breaking

spatial plausibility. In this respect the Apple Vision Pro faces the same constraint as the Pico 4 Ultra.

12.2 HTC VIVE Focus

The HTC VIVE Focus line targets enterprise deployment and exposes spatial functionality through the VIVE OpenXR SDK [99]. The underlying hardware uses a comparable Qualcomm Snapdragon SoC, placing it in a similar thermal and memory envelope to the Quest 3.

The VIVE OpenXR SDK exposes environment meshes through standard OpenXR scene understanding extensions, so raw mesh data is accessible in principle. It lacks a scene management layer equivalent to MRUK, however. On the Quest 3, MRUK handles asynchronous scene loading and delivers the room geometry as ready-to-use UE5 components with consistent coordinate conventions. On VIVE, the coordinate transformations described in Section 8.3.2 (centimeter-to-meter scaling, left-to-right handedness inversion, and winding order correction) would need to be applied manually against raw OpenXR extension structs. This increases implementation complexity but does not change the underlying algorithm, porting is feasible.

12.3 Pico 4 Ultra

The Pico 4 Ultra is close to the Quest 3 in hardware profile, with a similar Snapdragon SoC and full OpenXR compliance [100]. It exposes spatial scene meshes, so the coordinate mapping pipeline would work on this platform. The `URayLineComponent` rendering architecture and UDP reception layer would also run unmodified, as neither depends on Meta-specific APIs.

The blocking constraint is soft occlusion. Rendering virtual rays correctly behind physical geometry requires the Depth API, which on the Quest 3 is provided through engine-level patches in the Meta Oculus-VR fork of Unreal Engine (Section 7.4.1). The Pico XR SDK targets stock Unreal Engine builds and has no equivalent depth-compositing path [101]. Without soft occlusion, virtual rays are rasterized over foreground physical objects, breaking spatial plausibility. All other parts of the system would function correctly.

12.4 Summary

Table 12.1 summarizes the compatibility assessment. The Quest 3 is not an interchangeable hardware choice: the automated coordinate mapping depends on MRUK's raw mesh access, and soft occlusion depends on the Meta engine fork. Both dependencies are Meta-specific at the time of writing. As noted in Chapter 7, fork-exclusive features have historically migrated into the standard plugin over time, so the occlusion dependency may become portable as the Pico XR and mainline Unreal ecosystems mature.

Table 12.1: Cross-platform compatibility assessment.

Platform	SDK	Raw mesh access	Soft occlusion	Compatibility
Meta Quest 3	OpenXR/Meta MRUK	Yes (GLOBAL_MESH)	Meta Depth API	Fully operational
Apple Vision Pro	visionOS/ARKit	Yes (ARKit, UE 5.4+)	RealityKit (native only)	Feasible in-session, no occlusion
HTC VIVE Focus	OpenXR/VIVE OpenXR SDK	Yes (OpenXR extensions)	Standard bounding volumes	Feasible (manual transforms)
Pico 4 Ultra	OpenXR/Pico XR SDK	Yes (scene meshes)	Not available	Feasible (no occlusion)

Part III

Evaluation and Conclusion

Results and Discussion

To elevate this evaluation beyond a mere reporting of metrics, this chapter critically examines the system’s performance, spatial accuracy, and visual plausibility. The results are discussed in the context of the methodological choices made during development, explicitly reflecting on the limitations of the pipeline and determining whether the overarching aims of the thesis were fulfilled.

13.1 Streaming and Classification Throughput

13.1.1 Performance Benchmarks and Architectural Gains

The backend classification pipeline was evaluated against the complete 780 times-tamp dataset (comprising 122,262 total rays cast against a 696,228-triangle LiDAR mesh). Running on an 8-core host, the Rust-based pipeline achieved a mean per-frame processing time of 75 μ s (58.7 ms total across all 780 frames), equating to a throughput of approximately 13,000 frames per second (fps). This represents an approximate 30,000 $\times$ speedup over the geometrically equivalent Python (PyVista) baseline, which operated at 0.39 fps (see Appendix A).

Discussion: While this massive performance delta confirms the success of the rewrite, it is critical to correctly attribute the gains. As explored in Chapter 3, the drastic speed-up is not solely a product of Rust’s raw execution speed. Instead, it stems from correcting the inefficient resource management of the Python prototype, which synchronously rebuilt the spatial index from scratch for every single ray during the `find_closest_cell` call. By utilizing the `parry3d` crate (Chapter 5), the final backend builds the QBVH once at load time and reuses it across all frames. Furthermore, eliminating Python’s GIL allowed the `rayon` runtime to seamlessly parallelize the workload across all available cores.

At the deployment target of 10 fps, classification consumes less than 0.1% of the inter-frame budget, confirming that heavy geometric processing can be safely decoupled from the headset.

13.1.2 Transport Stability and Bounded Latency

The transmission of classified frames over the custom UDP binary protocol proved highly stable. During prolonged local-network testing sessions, the system maintained a steady 10 fps pacing with no observed frame-level stalls.

Discussion: The results validate the decision to utilize UDP rather than TCP, a trade-off detailed in Chapter 6. In an augmented reality context, delayed data is obsolete data. Because the custom binary payload requires only 7,068 bytes per frame for dataset [1] (well under the 65,507-byte UDP ceiling [42]) fragmentation overhead was avoided entirely on standard local networks. If a packet is dropped, the renderer simply advances to the next temporal snapshot without triggering the cascading retransmission stalls (head-of-line blocking) that would otherwise severely degrade the XR experience.

13.2 On-Device Rendering and Spatial Alignment

13.2.1 Hardware Constraints and Rendering Headroom

To maintain a comfortable user experience on the Meta Quest 3, the rendering pipeline must strictly adhere to an 11.1 ms frame budget to sustain 90 Hz. Qualitative testing across sessions lasting up to 15 minutes revealed no visible frame drops, stutter, or judder.

Discussion: The success of the visualizer is deeply rooted in the $O(1)$ draw call architecture implemented in Chapter 10. By migrating away from Unreal Engine’s Niagara system and aggregating all rays into a single dynamically updated mesh element using billboarding, the CPU-to-GPU memory transfer was drastically minimized.

A notable limitation of this evaluation is the absence of formal GPU frame-time profiling. While qualitative stability was achieved, the exact rendering margin within the 11.1 ms budget and the absolute ceiling for concurrent ray visualization remain unquantified.

13.2.2 Spatial Registration Accuracy

The automated coordinate mapping pipeline, utilizing the MRUK scene mesh combined with server-side RANSAC and point-to-plane ICP, reduced the residual positioning error to between 1 cm and 5 cm.

Discussion: As discussed in Chapter 8, this automated alignment provides an order-of-magnitude improvement over the initial manual spatial anchor approach, which suffered from user-induced placement errors of 10 or more cm. Furthermore, because this approach integrates directly with MRUK’s world-locking mechanism, the headset’s inside-out tracking continuously corrects for spatial drift over time. Critically, the remaining 1-5 cm discrepancy is no longer a backend algorithmic

failure; it is largely bounded by the fundamental fidelity of the Quest 3’s initial spatial scan and the inherent noise of the hardware sensors.

13.3 Visual Plausibility and Geometric Constraints

13.3.1 Path Distribution and Environmental Anomalies

The geometric classifier categorized the 122,262 captured rays into a distribution that closely mirrors expectations for a furnished indoor environment: 62.6% multi-bounce, 15.1% single-bounce, 5.7% line-of-sight, and 16.5% out-of-bounds.

Discussion: While line-of-sight and single-bounce rays consistently anchored to the correct physical surfaces, critically reviewing the 16.5% out-of-bounds rate highlights a major limitation in the mesh reconstruction pipeline. This anomaly is primarily driven by the absence of the floor surface in the reconstructed LiDAR mesh (Chapter 4); consequently, downward-propagating rays fail to intersect geometry and are prematurely clipped. A second source of geometric error is reflective surfaces such as windows: the LiDAR scanner captures mirror reflections as points apparently located behind the glass, projecting ghost geometry into the interior of the room. Even after the filtering stage, residual ghost points can survive into the Poisson reconstruction and introduce false surfaces that incorrectly intercept rays. Additionally, small objects may not be captured with sufficient point density for the reconstruction to represent them faithfully; where the scan is sparse the smoothing stage may reduce or lose the feature, and rays that physically bounced off such objects may find no intersecting geometry in the mesh. Finally, SAGE estimation noise on weaker multipath components occasionally produces AoAs that cannot be geometrically reconciled with the room.

13.3.2 Synthesizing Spatial Intuition

Discussion: Despite the geometric constraints of the static mesh, the system successfully translates abstract channel data into an intuitive spatial diagram, as illustrated in Figure 13.1. By implementing real-time filtering (antenna isolation, ray-type toggles, and distance clipping) with color encoding, the high density of multi-bounce paths is managed effectively without cluttering the AR application.

Together, the real-time filtering, color encoding, and spatial registration allow the dense overlay of multi-bounce paths to be explored interactively, replacing abstract numerical data with room-scale spatial context.

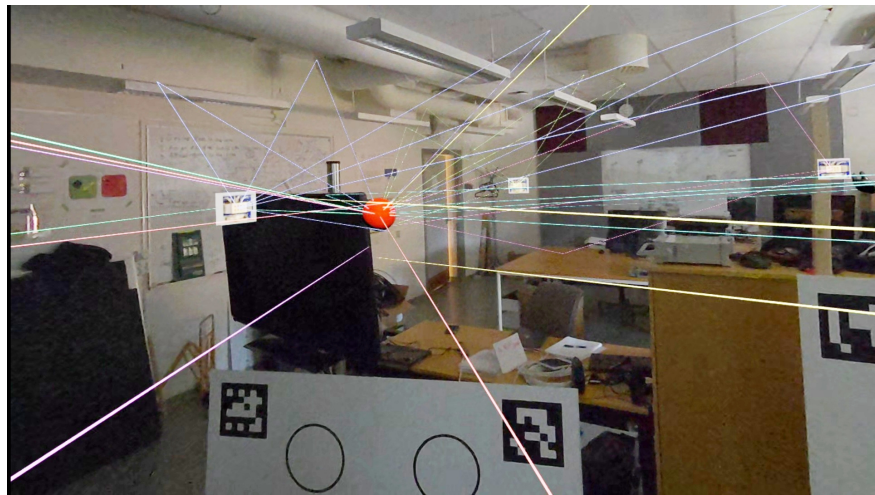


Figure 13.1: Propagation rays overlaid on the measurement room as seen through the Meta Quest 3.

Future Work

14.1 Live Data Integration

The most impactful near-term extension is the integration of a live real-time measurement source, though this requires a real-time MPC estimator. Once that prerequisite is met, the server side requires changes only to the data-ingestion stage and outer driver loop, since the classification, streaming, and rendering layers are already designed to be source-agnostic. On the headset side, two architectural changes would be needed to support unbounded live sessions: replacing the current indefinitely expanding frame buffer with a fixed-size circular buffer that evicts the oldest frames when the memory ceiling is reached, and introducing a pre-allocated pool of frame containers to eliminate repeated allocation and deallocation on the ingestion hot path. Long-running installations would additionally require explicit handling of 32-bit timestamp rollovers, though the current range is sufficient for several days of continuous 100 Hz streaming.

A further requirement for live sessions is dynamic environment tracking: the room mesh is currently captured once before a session and remains static throughout, so physical changes such as moved furniture during a measurement campaign cause the geometry used for bounce-point classification to diverge from the actual environment.

14.2 Network Protocol

The UDP wire protocol currently encodes each timestamp as a single datagram, which limits the maximum frame size to the 65,507-byte UDP payload ceiling. For denser datasets or extended ray sets, this becomes a hard constraint; a multi-datagram framing scheme, where a single timestamp is split across several sequenced datagrams and reassembled on the receiver, would remove this limit entirely.

14.3 Engine and Hardware Expansion

Formal GPU frame-time profiling, via Snapdragon Profiler or Unreal Insights on an on-device build, would quantify the remaining rendering margin within the 11.1 ms budget and establish how many additional rays the pipeline can sustain before the constraint is reached.

On the engine side, the system currently depends on the Meta Oculus-VR fork of Unreal Engine for soft occlusion via the Depth API. Once these features migrate to the mainline plugin release, the engine dependency could be dropped, simplifying the build pipeline.

Future work can focus on transitioning the project fully toward standard OpenXR implementations, vastly improving its portability to other OpenXR-compliant headsets.

14.4 Environment and Geometric Processing

A complementary improvement to the static mesh capture is refinement of the mesh smoothing algorithm. The quality of the reconstructed mesh is directly limited by the density and surface-normal information available in the input point cloud; providing a denser point cloud with per-point normals would give the Poisson reconstruction more accurate surface orientation data, resulting in better-preserved geometry for small objects and tighter bounce-point classification.

Where AoD estimates are available from SAGE, they could additionally be used to constrain the geometric classification: combining the arrival direction at the receiver with the departure direction from the UE provides a tighter bound on the expected bounce point, reducing misclassification in ambiguous multi-bounce scenarios.

14.5 Interaction and Parameter Tuning

The interaction system could be extended to expose adjustable parameter ranges for all configurable values (classification thresholds of Section 5.1, filter bounds, and attenuation scale) directly within the headset UI. Currently these are compile-time constants, so researchers must recompile to explore different settings. Exposing them as runtime controls would allow parameter tuning during a live measurement campaign without interrupting the session.

This thesis addressed the gap between the raw, numerical output of indoor D-MIMO channel measurements and the spatial intuition required to interpret them. Propagation delays and AoAs only become actionable when placed in the physical environment that produced them. To bridge this gap, this project developed an end-to-end pipeline that translates SAGE-estimated MPCs into spatially registered, interactive AR rays rendered on a standalone Meta Quest 3 headset without tethered workstations or external tracking infrastructure.

15.1 Summary of Findings

The outcomes of this work directly address the three research questions established in Chapter 1:

RQ1: Classification and Streaming. The Rust-based QBVH backend classifies all measurement frames against the environment mesh at a mean rate of approximately 13,000 frames/s on an 8-core host. This represents a $\approx 30,000\times$ improvement over the geometrically equivalent Python (PyVista) baseline, a gain attributable primarily to building the spatial index once at load time rather than per call, and secondarily to eliminating the GIL to parallelize per-ray classification across all cores. At the deployment delivery rate of 10 fps, classification consumes less than 0.1% of the available inter-frame budget. The compact binary wire format reliably delivered frames over UDP with no observed frame-level stalls or packet-reassembly failures. Thus, MPC classification and stable real-time streaming to a standalone AR application are feasible.

RQ2: On-Device Rendering and Alignment. By submitting the entire ray dataset as a single batched draw call with an $O(1)$ GPU command count, the custom scene proxy architecture sustained smooth rendering across extended test sessions without visible judder. Spatial registration, powered by the automated MRUK, RANSAC, and ICP pipeline, achieved a residual positioning error of 1 cm to 5 cm, an order-of-magnitude improvement over the manual approach. Consequently, the rays can be rendered efficiently within the device’s hardware

constraints, and the spatial registration remains highly accurate for analytical applications.

RQ3: Visual Plausibility. Classification across all 780 timestamps yielded a non-LoS dominated distribution (62.6% multi-bounce, 15.1% single-bounce, 5.7% line-of-sight, and 16.5% out-of-bounds), reflecting the true physical complexities of the furnished indoor environment. Qualitative assessments confirmed that line-of-sight and single-bounce rays consistently anchored to the correct physical walls, floors, and ceilings. The pipeline successfully generates visually plausible propagation paths that maintain strict consistency with the reconstructed room geometry.

15.2 Broader Implications

The results of this thesis present several broader implications for the field of spatial channel analysis:

Democratization of Spatial Analysis. The entirety of the system operates on a consumer-grade Meta Quest 3 paired with a standard laptop. The elimination of expensive rendering workstations and external tracking systems significantly lowers the barrier to entry for spatial analysis. Measurement campaigns that already possess basic computing resources can deploy this interactive tool at negligible additional cost, shifting the primary bottleneck of spatial visualization from hardware availability to software pipeline execution.

Validation of the Decoupled Architecture. The architectural decision to pre-classify all frames on the backend and stream them statelessly over UDP proved highly effective. This decoupled approach entirely hides the computational cost of BVH traversal from the headset, ensuring the mobile processor is strictly dedicated to rendering.

Platform Dependencies as a Portability Constraint. As analyzed in Chapter 12, the system’s reliance on raw scene mesh access (via MRUK for automated coordinate mapping) and the Depth API (for soft occlusion) currently restricts its portability. Competing platforms, such as the Apple Vision Pro, restrict raw mesh access, which would necessitate a regression to error-prone manual anchor placement. Consequently, spatial correctness in this domain is not yet portable by default; evaluating native API capabilities must be the primary criterion when selecting future target platforms.

15.3 Limitations

Three constraints bound the scope of this work. The visualization is qualitative: it supports spatial inspection but not formal verification against ray-tracing predictions. The room mesh is static, captured once and not updated during a session. Finally, the GPU frame-time profiling was not instrumented, so the rendering headroom within the 11.1 ms budget remains uncharacterized.

Bibliography

- [1] Y. Xu, X. Cai, M. Sandra, S. Willhammar, and F. Tufvesson, *3D cooperative user tracking for distributed integrated sensing and communication*, 2025. arXiv: [2511.03612 \[eess.SP\]](https://arxiv.org/abs/2511.03612). [Online]. Available: <https://arxiv.org/abs/2511.03612>.
- [2] Y. Xu, M. Sandra, X. Cai, S. Willhammar, and F. Tufvesson, "Interacting object-enabled clustering and characterization of distributed MIMO channels," *IEEE Transactions on Wireless Communications*, vol. 25, pp. 12 376–12 390, 2026. DOI: [10.1109/TWC.2026.3665431](https://doi.org/10.1109/TWC.2026.3665431).
- [3] X. Yin, B. Fleury, P. Jourdan, and A. Stucki, "Polarization estimation of individual propagation paths using the SAGE algorithm," in *14th IEEE Proceedings on Personal, Indoor and Mobile Radio Communications, 2003. PIMRC 2003.*, vol. 2, 2003, 1795–1799 vol.2. DOI: [10.1109/PIMRC.2003.1260424](https://doi.org/10.1109/PIMRC.2003.1260424).
- [4] A. Bounceur, L. Clavier, P. Combeau, *et al.*, "CupCarbon: A new platform for the design, simulation and 2D/3D visualization of radio propagation and interferences in IoT networks," in *2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, Jan. 2018, pp. 1–4. DOI: [10.1109/CCNC.2018.8319179](https://doi.org/10.1109/CCNC.2018.8319179). [Online]. Available: <https://ieeexplore.ieee.org/document/8319179/> (visited on 05/13/2026).
- [5] P. Tang, J. Zhang, Y. Zhang, *et al.*, "Demo abstract: "See the radio waves" via VR and its application in wireless communications," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, May 2021, pp. 1–2. DOI: [10.1109/INFOCOMWKSHPS51825.2021.9484582](https://doi.org/10.1109/INFOCOMWKSHPS51825.2021.9484582). [Online]. Available: <https://ieeexplore.ieee.org/document/9484582> (visited on 02/10/2026).

- [6] G. Koutitas, V. Kumar Siddaraju, and V. Metsis, “In situ wireless channel visualization using augmented reality and ray tracing,” *Sensors*, vol. 20, no. 3, p. 690, Jan. 2020, ISSN: 1424-8220. DOI: [10.3390/s20030690](https://doi.org/10.3390/s20030690). [Online]. Available: <https://www.mdpi.com/1424-8220/20/3/690> (visited on 05/13/2026).
- [7] M. Wang, H. Wang, H. Dong, and A. E. Saddik, “3D Wi-Fi signal measurement in realistic digital twin testbed environments using ray tracing,” *IEEE Transactions on Instrumentation and Measurement*, vol. 75, pp. 1–13, 2026, ISSN: 0018-9456, 1557-9662. DOI: [10.1109/TIM.2026.3660422](https://doi.org/10.1109/TIM.2026.3660422). arXiv: [2602.13340](https://arxiv.org/abs/2602.13340) [cs.NI]. [Online]. Available: <http://arxiv.org/abs/2602.13340> (visited on 05/13/2026).
- [8] *This ESP32 Antenna Array Can See WiFi*, Feb. 14, 2025. [Online]. Available: <https://www.youtube.com/watch?v=sXwDrcd1t-E> (visited on 05/13/2026).
- [9] Y. Xu, X. Cai, S. Willhammar, and F. Tufvesson, “Experimental analysis of multipath characteristics in indoor distributed massive MIMO channels,” *IEEE Antennas and Wireless Propagation Letters*, vol. 24, no. 11, pp. 3866–3870, Nov. 2025, ISSN: 1536-1225, 1548-5757. DOI: [10.1109/LAWP.2025.3598631](https://doi.org/10.1109/LAWP.2025.3598631). arXiv: [2504.12258](https://arxiv.org/abs/2504.12258) [eess]. [Online]. Available: <http://arxiv.org/abs/2504.12258> (visited on 05/05/2026).
- [10] “Meta Quest 3: Next-gen virtual reality headset,” Meta. (n.d.), [Online]. Available: <https://www.meta.com/se/en/quest/quest-3/?srsltid=AfmB0orVyq3j1ZSHkfflTWvpdTKdyE1VunjcoS3E0EbGeC0dq70hX8sq> (visited on 05/21/2026).
- [11] J. D. Hunter, “Matplotlib: A 2D graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: [10.1109/MCSE.2007.55](https://doi.org/10.1109/MCSE.2007.55).
- [12] “Overview of Niagara effects for Unreal Engine | Unreal Engine 5.6 documentation | Epic developer community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/overview-of-niagara-effects-for-unreal-engine> (visited on 05/28/2026).
- [13] Localtonet. “TCP vs UDP: The complete guide to network protocols,” Localtonet. (Feb. 27, 2026), [Online]. Available: <https://localtonet.com/blog/tcp-vs-udp> (visited on 05/28/2026).
- [14] “Protocol Buffers,” Protocol Buffers Documentation. (n.d.), [Online]. Available: <https://protobuf.dev/> (visited on 05/28/2026).
- [15] B. Sullivan and A. Kaszynski, “PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK),” *Journal of Open Source Software*, vol. 4, no. 37, p. 1450,

- May 2019. DOI: [10.21105/joss.01450](https://doi.org/10.21105/joss.01450). [Online]. Available: <https://doi.org/10.21105/joss.01450>.
- [16] “Trimesh 4.12.2 documentation,” Trimesh. (n.d.), [Online]. Available: <https://trimesh.org/> (visited on 05/28/2026).
- [17] “Faster Python: Unlocking the Python Global Interpreter Lock | The PyCharm Blog,” The JetBrains blog. (Jul. 29, 2025), [Online]. Available: <https://blog.jetbrains.com/pycharm/2025/07/faster-python-unlocking-the-python-global-interpreter-lock/> (visited on 05/25/2026).
- [18] “Parry3d - Rust,” Docs.rs. (n.d.), [Online]. Available: <https://docs.rs/parry3d/0.17.6/parry3d/> (visited on 05/28/2026).
- [19] “Qbvh in parry3d:partitioning - Rust,” Docs.rs. (n.d.), [Online]. Available: <https://docs.rs/parry3d/0.17.6/parry3d/partitioning/struct.Qbvh.html> (visited on 05/28/2026).
- [20] “Nalgebra - Rust,” Docs.rs. (n.d.), [Online]. Available: <https://docs.rs/nalgebra/0.33.3/nalgebra/> (visited on 05/28/2026).
- [21] “Hdf5 0.8.1 - Docs.rs,” Docs.rs. (n.d.), [Online]. Available: <https://docs.rs/crate/hdf5/0.8.1> (visited on 05/28/2026).
- [22] “Point cloud - Open3D 0.19.0 documentation,” Open3D 0.19.0 Documentation. (n.d.), [Online]. Available: <https://www.open3d.org/docs/release/tutorial/geometry/pointcloud.html> (visited on 05/31/2026).
- [23] “Numpy.histogram — NumPy v2.4 Manual,” Numpy.org. (n.d.), [Online]. Available: <https://numpy.org/doc/2.4/reference/generated/numpy.histogram.html#numpy.histogram> (visited on 05/30/2026).
- [24] M. A. Fischler and R. C. Bolles, “Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography,” *Commun. ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1, 1981, ISSN: 0001-0782. DOI: [10.1145/358669.358692](https://doi.org/10.1145/358669.358692). [Online]. Available: <https://dl.acm.org/doi/10.1145/358669.358692> (visited on 06/02/2026).
- [25] J. M. Martínez-Otzeta, I. Rodríguez-Moreno, I. Mendiáldua, and B. Sierra, “RANSAC for robotic applications: A survey,” *Sensors*, vol. 23, no. 1, p. 327, Jan. 2023, ISSN: 1424-8220. DOI: [10.3390/s23010327](https://doi.org/10.3390/s23010327). [Online]. Available: <https://www.mdpi.com/1424-8220/23/1/327> (visited on 06/02/2026).
- [26] “Open3d.geometry.radius_outlier_removal — Open3D 0.6.0 documentation,” Open3D 0.6.0 Documentation. (n.d.), [Online]. Available: https://www.open3d.org/docs/0.6.0/python_api/open3d.geometry.radius_outlier_removal.html (visited on 05/30/2026).

- [27] “Surface reconstruction - Open3D 0.19.0 documentation,” Open3D 0.19.0 Documentation. (n.d.), [Online]. Available: https://www.open3d.org/docs/release/tutorial/geometry/surface_reconstruction.html (visited on 05/31/2026).
- [28] M. Kazhdan, M. Bolitho, and H. Hoppe, “Poisson surface reconstruction,” in *Symposium on Geometry Processing (SGP '06)*, The Eurographics Association, 2006, pp. 61–70. DOI: [10.2312/SGP/SGP06/061-070](https://doi.org/10.2312/SGP/SGP06/061-070). [Online]. Available: <https://hhoppe.com/poissonrecon.pdf>.
- [29] “Open3d.geometry.TriangleMesh — Open3D 0.7.0 documentation,” Open3D 0.7.0 Documentation. (n.d.), [Online]. Available: https://www.open3d.org/docs/0.7.0/python_api/open3d.geometry.TriangleMesh.html (visited on 05/31/2026).
- [30] G. Taubin, “Curve and surface smoothing without shrinkage,” in *IEEE International Conference on Computer Vision*, Jul. 1995, 852–857, ISBN: 0-8186-7042-8. DOI: [10.1109/ICCV.1995.466848](https://doi.org/10.1109/ICCV.1995.466848).
- [31] M. Desbrun, M. Meyer, P. Schröder, and A. H. Barr, “Implicit fairing of irregular meshes using diffusion and curvature flow,” in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '99, USA: ACM Press/Addison-Wesley Publishing Co., Jul. 1, 1999, pp. 317–324, ISBN: 978-0-201-48560-8. DOI: [10.1145/311535.311576](https://doi.org/10.1145/311535.311576). [Online]. Available: <https://dl.acm.org/doi/10.1145/311535.311576> (visited on 06/24/2026).
- [32] T. Tasdizen, R. Whitaker, P. Burchard, and S. Osher, “Geometric surface processing via normal maps,” *ACM Transactions on Graphics*, vol. 22, no. 4, pp. 1012–1033, Oct. 2003, ISSN: 0730-0301, 1557-7368. DOI: [10.1145/944020.944024](https://doi.org/10.1145/944020.944024). [Online]. Available: <https://dl.acm.org/doi/10.1145/944020.944024> (visited on 06/01/2026).
- [33] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, “Surface reconstruction from unorganized points,” in *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '92, New York, NY, USA: Association for Computing Machinery, Jul. 1, 1992, pp. 71–78, ISBN: 978-0-89791-479-6. DOI: [10.1145/133994.134011](https://doi.org/10.1145/133994.134011). [Online]. Available: <https://dl.acm.org/doi/10.1145/133994.134011> (visited on 06/24/2026).
- [34] T. Rabbani, F. A. van den Heuvel, and G. Vosselman, “Segmentation of point clouds using smoothness constraint,” in *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences (IAPRS)*, vol. 36, Dresden, Germany: ISPRS, Sep. 2006, pp. 248–253. [Online]. Available: https://www.isprs.org/proceedings/xxxvi/part5/paper/RABB_639.pdf.
- [35] L. Li, J. Yao, J. Tu, X. Liu, Y. Li, and L. Guo, “Roof plane segmentation from airborne LiDAR data using hierarchical clustering and

- boundary relabeling,” *Remote Sensing*, vol. 12, no. 9, p. 1363, Jan. 2020, ISSN: 2072-4292. DOI: [10.3390/rs12091363](https://doi.org/10.3390/rs12091363). [Online]. Available: <https://www.mdpi.com/2072-4292/12/9/1363> (visited on 06/01/2026).
- [36] Y. Chen and G. Medioni, “Object modelling by registration of multiple range images,” *Image and Vision Computing*, vol. 10, no. 3, pp. 145–155, 1992, ISSN: 0262-8856. DOI: [10.1016/0262-8856\(92\)90066-C](https://doi.org/10.1016/0262-8856(92)90066-C). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/026288569290066C>.
- [37] P. J. Besl and N. D. McKay, “Method for registration of 3-D shapes,” in *Sensor Fusion IV: Control Paradigms and Data Structures*, vol. 1611, Spie, 1992, pp. 586–606.
- [38] “Rayon - Rust,” Docs.rs. (n.d.), [Online]. Available: <https://docs.rs/rayon/latest/rayon/> (visited on 06/03/2026).
- [39] “Bevy Engine,” Bevy. (n.d.), [Online]. Available: <https://bevy.org> (visited on 06/24/2026).
- [40] “Bevy_egui - Rust,” Docs.rs. (n.d.), [Online]. Available: https://docs.rs/bevy_egui/latest/bevy_egui/ (visited on 06/03/2026).
- [41] “Bytemuck - Rust,” Docs.rs. (n.d.), [Online]. Available: <https://docs.rs/bytemuck/latest/bytemuck/> (visited on 06/03/2026).
- [42] R. Molenaar. “User Datagram Protocol (UDP) Packet Header,” NetworkLessons.com. (Jan. 13, 2026), [Online]. Available: <https://networklessons.com/ip-routing/user-datagram-protocol-udp-packet-header> (visited on 06/05/2026).
- [43] “Axum - Rust,” Docs.rs. (n.d.), [Online]. Available: <https://docs.rs/axum/latest/axum/> (visited on 06/03/2026).
- [44] “Tokio - Rust,” Docs.rs. (n.d.), [Online]. Available: <https://docs.rs/tokio/latest/tokio/> (visited on 06/03/2026).
- [45] “Missed frames and frame recovery,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unity/os-missed-frames/> (visited on 06/03/2026).
- [46] U. Technologies. “Unity - Manual: Introduction to scripting back ends,” Unity Documentation. (n.d.), [Online]. Available: <https://docs.unity3d.com/6000.2/Documentation/Manual/scripting-backends-intro.html> (visited on 06/14/2026).
- [47] U. Technologies. “Unity - Manual: Mono scripting back end,” Unity Documentation. (n.d.), [Online]. Available: <https://docs.unity3d.com/6000.3/Documentation/Manual/scripting-backends-mono.html> (visited on 06/14/2026).
- [48] U. Technologies. “Unity - Manual: Garbage collection modes,” Unity Documentation. (n.d.), [Online]. Available: <https://docs.unity3d.com/6000.3/Documentation/Manual/garbage-collection-modes.html> (visited on 06/14/2026).

- .com/6000.1/Documentation/Manual/performance-incremental-garbage-collection.html (visited on 06/14/2026).
- [49] “Entities overview | Entities | 1.0.16,” Unity. (n.d.), [Online]. Available: <https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html> (visited on 06/14/2026).
- [50] “Path to CoreCLR, 2026: Upgrade Guide - Technical Articles,” Unity Discussions. (Mar. 26, 2026), [Online]. Available: <https://discussions.unity.com/t/path-to-coreclr-2026-upgrade-guide/1714279> (visited on 06/14/2026).
- [51] “Actors in Unreal Engine | Unreal Engine 5.7 Documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/actors-in-unreal-engine> (visited on 06/14/2026).
- [52] “Unity vs Unreal Engine: Which is right for your game?” Hackr.io. (n.d.), [Online]. Available: <https://hackr.io/blog/unity-vs-unreal-engine> (visited on 06/03/2026).
- [53] “Common memory and CPU performance considerations in Unreal Engine | Unreal Engine 5.7 Documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/common-memory-and-cpu-performance-considerations-in-unreal-engine> (visited on 06/14/2026).
- [54] “Memory / RAM,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/po-memory-ram/> (visited on 06/03/2026).
- [55] U. Technologies. “Unity - Manual: Scriptable render pipeline fundamentals,” Unity Documentation. (n.d.), [Online]. Available: <https://docs.unity3d.com/6000.4/Documentation/Manual/scriptable-render-pipeline-introduction.html> (visited on 06/14/2026).
- [56] “Mobile rendering and shading modes for Unreal Engine | Unreal Engine 5.7 Documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/mobile-rendering-and-shading-modes-for-unreal-engine> (visited on 06/14/2026).
- [57] “Forward shading renderer,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/unreal-forward-renderer/> (visited on 06/14/2026).
- [58] A. Tereshchuk. “Unity vs Unreal in 2026: Which to choose?” N-iX Games. (Feb. 19, 2026), [Online]. Available: <https://gamestudio.n-ix.com/unity-vs-unreal-which-to-choose/> (visited on 06/03/2026).

- [59] “Oculus-VR Fork Game Engine performance enhancement,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/po-fork-engine-perf/> (visited on 06/03/2026).
- [60] “Enable late latching,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/unreal-late-latching/> (visited on 06/03/2026).
- [61] U. Technologies. “Unity - Manual: Enter Play mode with domain reload disabled,” Unity Documentation. (n.d.), [Online]. Available: <https://docs.unity3d.com/6000.3/Documentation/Manual/domain-reloading.html> (visited on 06/03/2026).
- [62] “Unity Iteration Speed best practices,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unity/po-unity-iteration/> (visited on 06/03/2026).
- [63] “Live Coding Primer | Knowledge base,” Epic Games Developer. (Dec. 8, 2021), [Online]. Available: <https://dev.epicgames.com/community/learning/knowledge-base/GDdl/unreal-engine-live-coding-primer> (visited on 06/03/2026).
- [64] “Profiling and debugging with Unity and native platform tools,” Unity. (n.d.), [Online]. Available: <https://unity.com/how-to/profiling-and-debugging-tools> (visited on 06/03/2026).
- [65] “Testing and performance analysis,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/unreal-debug-android/> (visited on 06/03/2026).
- [66] “Unity and OpenXR compatibility,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unity/unity-and-openxr-compatibility/> (visited on 06/03/2026).
- [67] “Develop Unity apps for Meta Quest VR headsets,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unity/unity-development-overview/> (visited on 06/03/2026).
- [68] “Interaction SDK,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unity/unity-isdk-interaction-sdk-overview/> (visited on 06/03/2026).
- [69] “VR Template in Unreal Engine | Unreal Engine 5.7 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/vr-template-in-unreal-engine> (visited on 06/03/2026).
- [70] “Meta XR Plugin settings,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/unreal-plugin-settings/> (visited on 06/03/2026).

- [71] “Meta Quest 3 hand and controller tracking setup with interaction SDK in Unreal Engine 5.5 | Community tutorial,” Epic Games Developer. (Sep. 16, 2025), [Online]. Available: <https://dev.epicgames.com/community/learning/tutorials/qp0V/meta-quest-3-hand-and-controller-tracking-setup-with-interaction-sdk-in-unreal-engine-5-5> (visited on 06/03/2026).
- [72] “Building for Mixed Reality on Meta Quest 3,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/blog/building-mixed-reality-MR-meta-quest-3-connect-developers-presence-platform/> (visited on 06/03/2026).
- [73] “Meta XR Simulator Overview,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/xrsim-intro/> (visited on 06/03/2026).
- [74] “Build Oculus-VR Fork Unreal Engine from Source,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/unreal-building-ue4-from-source/> (visited on 06/03/2026).
- [75] “Choosing Unreal Engine installation by feature compatibility,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/unreal-compatibility-matrix/> (visited on 06/03/2026).
- [76] “Depth API overview,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/unreal-depthapi-overview/> (visited on 06/03/2026).
- [77] “Choosing an OpenXR Backend,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/unreal-openxr/> (visited on 06/03/2026).
- [78] “Manage spatial data sharing on Meta Quest | Quest help (en-gb),” Meta. (n.d.), [Online]. Available: <https://www.meta.com/en-gb/help/quest/625635239532590/> (visited on 06/03/2026).
- [79] “Build believable Mixed Reality experiences with mesh API and depth API,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/blog/mesh-depth-api-meta-quest-3-developers-mixed-reality/> (visited on 06/03/2026).
- [80] T. Bezmalinovic. “Meta Quest 3 has a much better understanding of your room,” MIXED Reality News. (Sep. 18, 2023), [Online]. Available: <https://mixed-news.com/en/meta-quest-3-spatial-data/> (visited on 06/03/2026).
- [81] “Spatial Anchors overview,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/unreal-spatial-anchors/> (visited on 06/03/2026).

- [82] “Mixed Reality Utility Kit features,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/documentation/unreal/unreal-mr-utility-kit-features/> (visited on 06/03/2026).
- [83] “Meta XR MR Utility Kit (Unreal),” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/downloads/package/meta-xr-mr-utility-kit-unreal/> (visited on 06/03/2026).
- [84] “Unreal Engine: Meta XR Plugin v201,” Meta Horizon. (n.d.), [Online]. Available: <https://developers.meta.com/horizon/reference/unreal/v201/> (visited on 06/03/2026).
- [85] J. Lopez-Xarbau, M. A. Rodriguez-Fernandez, M. Faundez-Zanuy, J. Calvo-Sanz, and J. J. Garcia-Tirado, “Robot-driven calibration and accuracy assessment of Meta Quest 3 inside-out tracking using a TECHMAN TM5-900 Collaborative Robot,” *Sensors*, vol. 26, no. 8, p. 2285, Jan. 2026, ISSN: 1424-8220. DOI: [10.3390/s26082285](https://doi.org/10.3390/s26082285). [Online]. Available: <https://www.mdpi.com/1424-8220/26/8/2285> (visited on 06/03/2026).
- [86] “Map containers in Unreal Engine | Unreal Engine 5.7 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/map-containers-in-unreal-engine> (visited on 06/05/2026).
- [87] “Array containers in Unreal Engine | Unreal Engine 5.7 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/array-containers-in-unreal-engine> (visited on 06/05/2026).
- [88] “Blueprint Spline components overview in Unreal Engine | Unreal Engine 5.7 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/blueprint-spline-components-overview-in-unreal-engine> (visited on 06/05/2026).
- [89] “GPU raytracing collisions in Niagara for Unreal Engine | Unreal Engine 5.7 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/gpu-raytracing-collisions-in-niagara-for-unreal-engine> (visited on 06/05/2026).
- [90] “Threaded rendering in Unreal Engine | Unreal Engine 5.7 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/threaded-rendering-in-unreal-engine> (visited on 06/05/2026).

- [91] “Instanced static mesh component in Unreal Engine | Unreal Engine 5.7 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/instanced-static-mesh-component-in-unreal-engine> (visited on 06/05/2026).
- [92] “UProceduralMeshComponent | Unreal Engine 5.6 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/API/Plugins/ProceduralMeshComponent/UProceduralMeshComponent> (visited on 06/05/2026).
- [93] “Components in Unreal Engine | Unreal Engine 5.7 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/components-in-unreal-engine> (visited on 06/05/2026).
- [94] “Graphics programming overview for Unreal Engine | Unreal Engine 5.7 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/graphics-programming-overview-for-unreal-engine> (visited on 06/05/2026).
- [95] “Blueprint communication usage in Unreal Engine | Unreal Engine 5.7 documentation | Epic Developer Community,” Epic Games Developer. (n.d.), [Online]. Available: <https://dev.epicgames.com/documentation/unreal-engine/blueprint-communication-usage-in-unreal-engine> (visited on 06/05/2026).
- [96] “Apple Vision Pro - Tech specs,” Apple Support. (n.d.), [Online]. Available: <https://support.apple.com/en-us/117810> (visited on 06/03/2026).
- [97] “Can UE on macOS achieve feature parity with Windows? Find out!” Unreal Engine. (n.d.), [Online]. Available: <https://www.unrealengine.com/tech-blog/bringing-unreal-engine-on-macos-up-to-feature-parity-with-windowsprogress-report> (visited on 06/14/2026).
- [98] “ARKit in visionOS,” Apple Developer Documentation. (n.d.), [Online]. Available: <https://developer.apple.com/documentation/arkit/arkit-in-visionos> (visited on 06/03/2026).
- [99] “Unreal Engine: Scene understanding | VIVE OpenXR - Developer resources.” (n.d.), [Online]. Available: <https://developer.vive.com/resources/openxr/unreal/unreal-tutorials/interactive-visual-computing/sceneunderstanding/> (visited on 06/03/2026).
- [100] “PICO4 Ultra-VR-MR-headset | PICO Global.” (n.d.), [Online]. Available: <https://www.picoxr.com/global/products/pico4-ultra/specs> (visited on 06/03/2026).

-
- [101] “Spatial Mesh Pico,” PICO Developer. (n.d.), [Online]. Available: <https://developer.picoxr.com/document/unreal/spatial-mesh/> (visited on 06/03/2026).

Classification Throughput Benchmark Data

This appendix documents the raw measurements from the backend classification throughput benchmarks referenced in Section 13.1. All runs used the same input dataset: 780 measurement timestamps, 8 antenna arrays, and approximately 160 MPCs per frame, against a 696,228-triangle LiDAR environment mesh.

Test environment

- **Host machine:** Lenovo ThinkPad E15 Gen 2, 11th Gen Intel Core i7-1165G7 @ 4.700 GHz (8 logical cores)
- **Rust toolchain:** rustc 1.90.0 (1159e78c4 2025-09-14), release build (opt-level = 3, lto = true)
- **Python version:** 3.12
- **Dataset:** Mea_DMIM0.mat (780 timestamps, 8 sensors, variable MPCs per cell)

Python SnapshotBuilder (no mesh)

The Python `SnapshotBuilder` was timed across all 780 timestamps over 20 trials. Classification uses a distance-threshold heuristic only (no ray-mesh intersection).

```
Data load   : 1022.0 ms
Rays/frame : 156 (timestamp 0) | trials = 20
mean = 2588.48 ms   std = 88.07 ms
min  = 2436.60 ms   max = 2690.84 ms
fps  mean = 301.3   range = [289.9, 320.1]
```

Python PyVista mesh classifier

The PyVista-based classifier reproduces the full geometric classification (equivalent to the Rust pipeline): per-ray `pv.ray_trace()` BVH intersection, distance check,

and angle-of-incidence single-bounce validation. Due to the high per-call cost, 30 evenly-spaced timestamps were timed per trial over 20 trials (plus 1 warm-up pass). The 780-frame estimate is extrapolated linearly.

```
Mesh load   : 535.6 ms (696,228 cells)
Timestamps  : 30 per trial (evenly spaced) | trials = 20
Rays/frame  : ~157
Per ray     : 16,251.2 µs (mean over 30 ts)
```

```
mean = 76,429.45 ms   std = 5,817.87 ms (CV = 7.6 %)
min  = 68,564.26 ms   max = 90,948.49 ms
fps (sample, mean) = 0.39   range = [0.33, 0.44]
```

Extrapolated to 780 frames: ~1,987 s (~0.39 fps, ~33 minutes)

Rust benchmark

The Rust `bench` binary times `compute_frame()` across all 780 timestamps. The same full BVH classification (parry3d, line-of-sight / single-bounce / multi-bounce / out-of-bounds) is performed in all variants.

8-thread run (default)

```
LiDAR load   : 126.3 ms (696,228 triangles)
Data load    : 70.5 ms (780 timestamps)
Rays/frame   : 156 (timestamp 0)
```

Sequential (frames in order, intra-frame Rayon active):

```
Total time : 58.7 ms
Per frame  : 75.3 µs
Throughput : 13,286.8 frames/s
Total rays : 122,262
```

Parallel (frames AND rays in parallel):

```
Total time : 38.1 ms
Per frame   : 48.8 µs
Throughput  : 20,494.3 frames/s
Speedup     : 1.54× over sequential
```

1-thread run (RAYON_NUM_THREADS=1)

```
LiDAR load   : 137.6 ms (696,228 triangles)
Data load    : 72.9 ms (780 timestamps)
```

Sequential (true single-thread, Rayon disabled):

```
Total time : 198.5 ms
Per frame   : 254.5 µs
```

Throughput : 3,929.5 frames/s

Speedup summary

Table A.1: Speedup of Rust backend relative to Python baselines for full 780-timestamp dataset.

Comparison			Speedup	Note
Rust	1-thread	vs.	$\approx 13\times$	Different complexity: Rust does full QBVH, Python uses distance threshold only
Python	Snapshot-Builder			
Rust	8-thread	vs.	$\approx 40\times$	Same caveat as above
Python	Snapshot-Builder			
Rust	1-thread	vs.	$\approx 10,000\times$	Equivalent task: PyVista figure extrapolated from 20×30-frame samples
Python	PyVista			
Rust	8-thread	vs.	$\approx 30,000\times$	Equivalent task: same extrapolation
Python	PyVista			

The specific benchmark scripts `backendRust/src/bin/bench.rs` (Rust) and `benchmark_python.py` (Python) used to evaluate the implementations are maintained in a private repository and are available from the authors upon request.