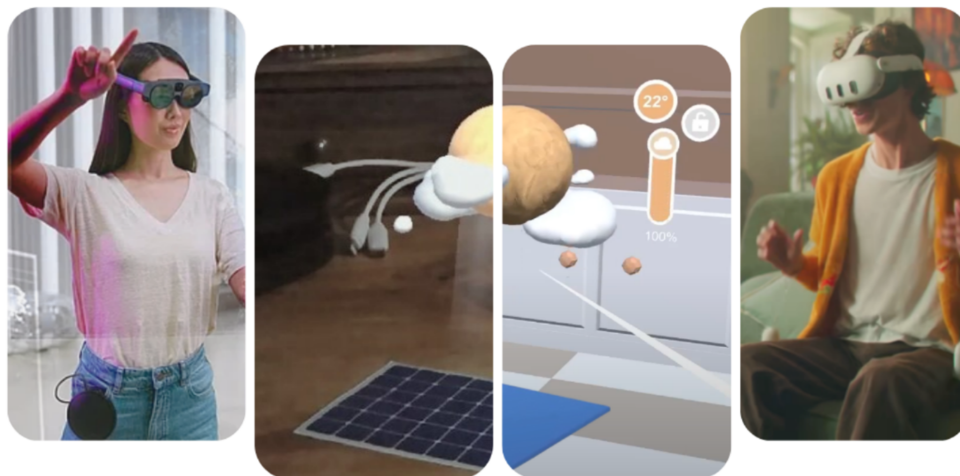
ISEL
INSTITUTO SUPERIOR DE
ENGENHARIA DE LISBOA



# Cross-Device Platform for Collaborative Educational Experiences in Mixed Reality


**LETÍCIA BARBEDO LUCAS**

(Licenciada em Engenharia Informática e Multimedia)


Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e Multimédia

**Orientadores:** Doutora Carla Maria Duarte da Silva e Costa
Doutor Pedro Miguel Torres Mendes Jorge

**Júri:**

**Presidente:** Doutor Pedro Viçoso Fazenda
**Vogais:** Doutor Pedro Emanuel Albuquerque e Baptista dos Santos
Doutor Pedro Miguel Torres Mendes Jorge

**Setembro 2025**

# Cross-Device Platform for Collaborative Educational Experiences in Mixed Reality

**LETÍCIA BARBEDO LUCAS**

(Licenciada em Engenharia Informática e Multimedia)

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e Multimédia

**Orientadores:** Doutora Carla Maria Duarte da Silva e Costa, ISEL
Doutor Pedro Miguel Torres Mendes Jorge, ISEL

**Júri:**

**Presidente:** Doutor Pedro Viçoso Fazenda, ISEL
**Vogais:** Doutor Pedro Emanuel Albuquerque e Baptista dos Santos, IPS
Doutor Pedro Miguel Torres Mendes Jorge, ISEL

**Setembro 2025**

# Acknowledgements

This work would not have been possible without the original vision of my advisers, who entrusted this project to me and helped shape its path from the very beginning. Their support, guidance, and insight have been essential throughout the entire process.

I also wish to thank the Instituto Superior de Engenharia de Lisboa (ISEL), for being my academic home for the past five years. To all its professors and staff, whose rigor, care and availability throughout this time have been decisive in shaping not only my education, but also my personal and professional growth.

I formally acknowledge the support of the ICSE Science Factory project (2023–2026), funded by the EU's Horizon Europe programme (Grant No. 101093387, Call HORIZON-WIDERA-2022-ERA-01). DOI: 10.3030/101093387.

On a personal note, I extend my heartfelt thanks to my friends - both those of long standing and those who I was lucky enough to meet during my years at this institution - for their companionship through shared endeavors.

Finally, I am endlessly grateful to my family: those who to this day remain by my side, supporting me unfailingly; and also to those who, without having ever imagined this path for me, would have undoubtedly been my strongest supporters.

## Statement of Integrity

I declare that this dissertation is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

Letícia Lucas

Lisbon, September 2025

# Abstract

This dissertation presents the design and implementation of a cross-device platform for collaborative educational experiences in Mixed Reality (MR). While these technologies have the potential to transform learning, turning abstract concepts into something tangible, their generalized adoption is hindered by some persistent challenges: device fragmentation and the lack of interoperability between different systems. With different devices providing incompatible capabilities, truly inclusive and scalable classroom experiences are difficult to attain.

To address this, the project proposes a modular client-server architecture enabling seamless collaboration between heterogeneous platforms. On the client side, OpenXR provides the foundational layer for interoperability, enabling unified user interaction across devices. On the server side, a dual approach is adopted: a Unity server coordinates multiplayer sessions, object alignment and scene management, while a Python server handles computer vision tasks. The latter integrates a custom object detection pipeline capable of personalized marker recognition and communicates with clients via sockets for real-time responses.

The developed platform is validated through a representative educational scenario focused on renewable energies, where users employing Magic Leap 2 and Meta Quest 3 devices - examples of Augmented and Virtual Reality, respectively - collaboratively manipulate synchronized virtual objects (a Sun and clouds), anchored to physical markers (miniature photovoltaic panel replicas). Results demonstrate that the proposed approach achieves real-time synchronization, robust object tracking, and extensibility for integrating both new devices and educational experiences in the future.

This work provides an open-source framework that enables educators and developers to create their own custom Mixed Reality experiences, without requiring extensive technical knowledge. Beyond its immediate educational applications, the platform establishes a foundation for future research in cross-platform Mixed Reality systems, especially those that foster collaboration and creativity in learning.

**Keywords:**
Collaborative Mixed Reality, Cross-Device Development, Educational Platform, Immersive Experiences, Object Detection.

# Resumo

Esta dissertação apresenta processo de implementação de uma plataforma multi-dispositivo para experiências educativas colaborativas em Realidade Mista (RM). Embora estas tecnologias tenham o potencial de transformar a aprendizagem, ao tornar conceitos abstratos em algo tangível, uma adoção generalizada é dificultada por alguns desafios: nomeadamente a fragmentação tecnológica e a falta de interoperabilidade entre diferentes sistemas. Com dispositivos distintos a oferecer capacidades incompatíveis, experiências de sala de aula verdadeiramente inclusivas e expansíveis tornam-se difíceis de alcançar.

Para contornar estes obstáculos, o projeto propõe uma arquitetura modular cliente-servidor para permitir colaboração entre plataformas heterogéneas. Do lado do cliente, o OpenXR fornece os alicerces fundamentais para a interoperabilidade, permitindo unificar a interação dos utilizadores independentemente do dispositivo. Do lado do servidor, é adotada uma abordagem dupla: um servidor Unity coordena sessões multijogador, alinhamento de objetos e gestão de módulos, enquanto um servidor Python trata das tarefas de visão computacional. Este último integra um esquema de deteção de objetos personalizado capaz de reconhecer marcas específicos, e comunica com os clientes via *sockets* para respostas em tempo real.

A plataforma é validada através de um cenário educativo representativo focado em energias renováveis, onde utilizadores com dispositivos Magic Leap 2 e Meta Quest 3 - exemplos de Realidade Aumentada e Realidade Virtual, respetivamente - colaboram na manipulação de objetos virtuais sincronizados (um Sol e nuvens), ancorados a marcadores físicos (réplicas em miniatura de painéis fotovoltaicos). Os resultados demonstram que a abordagem proposta atinge sincronização em tempo real, rastreamento de objetos e adaptabilidade para integrar novos dispositivos e experiências educativas no futuro.

Este trabalho fornece uma *framework open-source* que permite que educadores e desenvolvedores criem as suas próprias experiências em Realidade Mista, mesmo sem conhecimento técnico avançado. Para além das suas vantagens educativas, a plataforma estabelece uma base para futuras pesquisas em sistemas de Realidade Mista multi-platforma, especialmente aqueles que promovem colaboração e criatividade na aprendizagem.

**Palavras-chave:**
Colaboração em Realidade Mista, Desenvolvimento Multi-Dispositivo, Deteção de Objetos, Experiências Imersivas, Plataforma de Ensino.

# Contents


**List of Figures** xvii

**List of Tables** xxi

**Listings** xxiii

**Glossary** xxv

**Acronyms** xxvii

**1 Introduction** 1

1.1 Motivation, Goals and Contributions . . . 2

1.1.1 Motivation . . . 2

1.1.2 Goals . . . 3

1.1.3 Contributions . . . 4

1.1.3.1 Scientific and Technical Contributions . . . 4

1.1.3.2 Scientific Publications and Conference Presentations . . . 5

1.2 Document Structure . . . 5

**2 Foundations of Mixed Reality Technologies** 7

2.1 Origins of Immersive Technologies . . . 7

2.2 Current State and Future Expectations . . . 8

2.3 Definitions and Distinctions . . . 9

2.3.1 Virtual Reality: Complete Immersion . . . 9

2.3.2 Augmented Reality: Fusion Between Real and Virtual . . . 10

2.3.3 The Concept of Mixed Reality . . . 11

**3 Related Work** 13

3.1 Overview of Current Existing Devices . . . 13

3.2 Current Mixed Reality Applications . . . 15

3.3 Cross-Platform Frameworks and Standards . . . 17

3.4 Spatial Mapping and Alignment Techniques . . . 18

3.5 Identified Gaps and Research Opportunities . . . 19

**4 Proposed Architecture** 21

4.1 Client Side - Device Management and Interaction . . . 22

4.2 Server Side - Multiplayer and Experience Management . . . 22

4.2.1 Unity Server - Experience Management and Synchronization . . . 22
4.2.2 Python Server - Object Detection . . . . . . . . . . . . . . . . . . . 23
4.3 Final Considerations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**5 System Implementation 25**
5.1 Python Server for Custom Object Detection . . . . . . . . . . . . . . . . . 25
5.1.1 Model Overview and Design . . . . . . . . . . . . . . . . . . . . . . 26
5.1.1.1 Training Process . . . . . . . . . . . . . . . . . . . . . . . 27
5.1.2 Technical Implementation . . . . . . . . . . . . . . . . . . . . . . . 28
5.1.2.1 Detection Process and Response Format . . . . . . . . . . 29
5.1.2.2 Concurrent Connection Management . . . . . . . . . . . . 31
5.1.2.3 Network Configuration Optimization . . . . . . . . . . . . 31
5.1.3 Key Design Decisions and Analysis . . . . . . . . . . . . . . . . . . 32
5.2 Application Design and Scene Management . . . . . . . . . . . . . . . . . 32
5.2.1 Tiered Scene Architecture . . . . . . . . . . . . . . . . . . . . . . . 33
5.2.2 Technical Implementation . . . . . . . . . . . . . . . . . . . . . . . 35
5.2.2.1 `SceneInfo` - Generalizing Scene Information . . . . . . . 36
5.2.2.2 `SceneLoader` - Managing Scene Transition Logic . . . . . 37
5.2.3 Key Design Decisions and Analysis . . . . . . . . . . . . . . . . . . 39
5.3 Multiplayer and Lobby Management . . . . . . . . . . . . . . . . . . . . . 39
5.3.1 Unity Gaming Services and the Challenges of Multiplayer . . . . . 39
5.3.1.1 Networking Solutions Comparison . . . . . . . . . . . . . 40
5.3.2 Technical Implementation . . . . . . . . . . . . . . . . . . . . . . . 42
5.3.2.1 User Authentication and Session Management . . . . . . 43
5.3.2.2 Lobbies and Relay System Overview . . . . . . . . . . . . 44
5.3.2.3 Lobby System for Session Coordination . . . . . . . . . . 44
5.3.2.4 Relay System for Experience Synchronization . . . . . . . 48
5.3.3 Key Design Decisions and Analysis . . . . . . . . . . . . . . . . . . 50
5.4 Object Interaction and Networking . . . . . . . . . . . . . . . . . . . . . . 51
5.4.1 Technical Implementation . . . . . . . . . . . . . . . . . . . . . . . 52
5.4.1.1 Network Object Lifecycle Management . . . . . . . . . . . 54
5.4.1.2 `Spawnable` vs. `Interactable` Distinction . . . . . . . . . 56
5.4.2 Key Design Decisions and Analysis . . . . . . . . . . . . . . . . . . 58
5.5 Device Management . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.5.1 Technical Implementation - General Overview . . . . . . . . . . . . 60
5.5.1.1 Handling Different Devices . . . . . . . . . . . . . . . . . 61
5.5.1.2 Device Lifecycle and Subsystem Management . . . . . . . 62
5.5.2 Marker Tracking in Augmented Reality . . . . . . . . . . . . . . . 64
5.5.2.1 The Challenge of Shared Space for Spatial Alignment . . 65
5.5.2.2 Completing the Marker Detection Workflow . . . . . . . . 67
5.5.2.3 Final Considerations about Frame Capturing and Distortion 69
5.5.3 Key Design Decisions and Analysis . . . . . . . . . . . . . . . . . . 70

**6 Module Development and Evaluation** **73**
6.1 Object Preparation and Interaction . . . . . . . . . . . . . . . . . . . . . . 75
6.1.1 Preparing Device Rigs . . . . . . . . . . . . . . . . . . . . . . . . 75
6.1.2 Preparing and Testing Custom `Interactable` Objects . . . . . . . 76
6.2 Cross-Device Multiplayer Validation . . . . . . . . . . . . . . . . . . . . . 79
6.3 Detection Model Training and Evaluation . . . . . . . . . . . . . . . . . . . 82
6.3.1 Model Training and Performance Analysis . . . . . . . . . . . . . . 82
6.3.2 Module Integration and Tracking Validation . . . . . . . . . . . . . 86

**7 Conclusions and Future Work** **89**
7.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

**Bibliography** **91**

**Appendices**

**A Depth Undistortion Utility Class for Magic Leap 2** **97**

**B *SunInteractable* Class - Handling the Testing Module's Interaction Requirements** **103**

# List of Figures


2.1 Images picturing "The Sword of Damocles". The use of a "a hand-held wand" enabled object interaction, closely resembling modern controller-interface paradigms. 7

2.2 Pictured are several examples of the use of Heads-Up Displays (HUDs) in aviation: on the left, a U.S. Navy jet can be seen aligned in the HUD of another aircraft; the two images on the right show how flight-related data can be displayed to aid the pilot. . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3 Some popular Virtual Reality (VR) device choices are shown: A. Meta Quest 2, B. HTC Vive Pro and C. Valve Index (Original). A simple exemplification of use is also shown: this technology completely immerses users in a virtual world. 10

2.4 Some Augmented Reality (AR) technologies can be seen: A. Holo Lens Pro 2, B. Magic Leap 2 and C. Google Glass Enterprise Edition 2. To the right, a smartphone-based application (Pokémon Go) is also shown: this technology overlays virtual objects atop real environments. . . . . . . . . . . . . . . . . . . 11

2.5 Virtuality-Reality spectrum as proposed by Paul Milgram. The illustrated diagram showcases examples of various technologies, and defines the range of some relevant terms. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.1 From left to right: a user handles a molecular structure in Nanome, an interactable heart model is shown in the zSpace setup, a virtual representation wraps around a real soccer ball detected using Geogebra. . . . . . . . . . . . . . . . 15

3.2 Examples of ArUco markers are illustrated. To the right, a practical use can be seen: upon detection, the markers' position and orientation can be estimated and displayed. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

4.1 Proposed architecture model depicting the main components along with their primary tasks, as well as the technologies and communication protocols employed. 21

5.1 Examples of You Only Look Once (YOLO) detection formats: on the left, a bounding box is constructed from the detected corners; on the right, a pixel-level mask captures all detected points. . . . . . . . . . . . . . . . . . . . . . . . . 26

5.2 Class diagram of the implemented object detection server. . . . . . . . . . . 28

5.3 Main data structures from the first three steps: 1) Request format; 2) Image frame; 3) Detection result format. . . . . . . . . . . . . . . . . . . . . . . . . . 29

5.4 Reference for the mask analysis steps: The segmentation mask (left) identifies every pixel that belongs to the detected object. Step 4a extracts only the outer contour, reducing the pixel amount. In step 4b, the contour is further approximated into a quadrilateral, keeping only the four most relevant corner points. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.5 Mock-up for the main menu interface: the experience selection panel is shown the left, while lobby options are displayed on the right (to be discussed in subsequent sections). . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.6 Visual reference for the proposed scene architecture, with the identified layers and transition conditions represented throughout the application lifecycle. . . 34
5.7 Class diagram focusing on scene-related classes: `SceneInfo` and `SceneLoader`. 35
5.8 Example of the creation of a `SceneInfo` object via the Unity editor interface. 37
5.9 Flowchart depicting the scene transition lifecycle. . . . . . . . . . . . . . . . . 38
5.10 Class diagram depicting the core components of the networking implementation. 42
5.11 Detailing of the right-hand panel of the main menu (previously shown in Figure 5.5), which enables lobby browsing and interaction. . . . . . . . . . . . . . . . 45
5.12 Updated panel after joining a lobby: the illustrated scenario is specific to the host, with coordination privileges being shown. . . . . . . . . . . . . . . . . . 46
5.13 Interaction diagram depicting the message flow between lobby participants and Unity's backend services during relay setup. . . . . . . . . . . . . . . . . . . . 48
5.14 Diagram depicting classes relevant to network object management. . . . . . . 52
5.15 Example of creating an object association ruleset in the Unity inspector: for each mapping, the marker identification/label can be typed out, with the prefab being drag-and-dropped from Unity's asset list. . . . . . . . . . . . . . . . . . 53
5.16 Diagram illustrating the Remote Procedure Call (RPC) calls between server and clients, along with the operational pipeline that manages object creation, updates, and synchronization across the network. . . . . . . . . . . . . . . . . 55
5.17 Example of how object visualization varies across device types: the virtual solar panel is made visible only to VR users, conceptually replacing the physical printed marker. The sun and the clouds are `Interactable` objects, enabling manipulation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.18 Diagram with the device management classes used across all Mixed Reality (MR) hardware. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
5.19 Example of the creation of `DeviceInfo` assets for AR and VR devices - respectively MagicLeap2 and MetaQuest3. . . . . . . . . . . . . . . . . . . . . . . . 62
5.20 Diagram depicting classes relevant to complete the marker detection workflow. 64
5.21 Example illustrating a major challenge in spatial alignment: users places within a common coordinate space (Unity) have different understandings of a same object. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.22 Detection conversion pipeline for spatial alignment. . . . . . . . . . . . . . . . 66
5.23 Marker detection interaction diagram. . . . . . . . . . . . . . . . . . . . . . . 67
5.24 Example of the fisheye effect, comparing an original capture with its undistorted version after correction algorithms are applied [66]. . . . . . . . . . . . . . . . 70

6.1 Conventional schematic representation of the solar panel power output, expressed as a function of direct normal irradiance ($I_{\text{DNI}}$), incidence angle ($\theta$), panel area ($A$), and efficiency ($\eta$), according to models reported in the solar energy literature [67, 68]. . . . . 73
6.2 Finalized setup, visualized in the Unity Inspector: two `DeviceInfo` assets linked to their corresponding device prefabs and interaction types. . . . . 76
6.3 Prefab prepared for the test module, shown in both AR and VR. Key interface elements are highlighted, including the virtual Sun with clouds, value visualizations, and other interactive components. Both the physical marker and its virtual proxy are visible in their respective views. . . . . 76
6.4 Testing atmospheric control: the slider interface lets users adjust cloud cover. 77
6.5 Testing with the virtual Sun, which enables users to adjust the angle of incidence. 78
6.6 Reference points from the textbook graph were selected replicated in the simulated module, allowing a direct comparison between computed outputs and the expected values. . . . . 78
6.7 Finalized setup, visualized in the Unity Inspector: the prepared `SceneInfo` is shown, along with the list of scenes included in the build. . . . . 80
6.8 Placeholder image linking to the demonstration video. . . . . 80
6.9 Annotation process using Label Studio, showing segmentation mask definition for the solar panel replica. . . . . 83
6.10 Examples from four validation images. . . . . 84
6.11 Example of color similarity challenges (left) and proof of multiple marker detection (right). . . . . 85
6.12 When a solar panel replica is detected in the user's physical environment, the virtual interactive elements are automatically spawned and aligned with the marker position. . . . . 86

# List of Tables

3.1 Comparison of Extended Reality (XR) Device Families . . . . . . . . . . . . . . 14
3.2 Educational Extended Reality (XR) Applications Comparison . . . . . . . . . 16

5.1 Comparison of popular networking frameworks and solutions. . . . . . . . . . 41
5.2 Unity Gaming Services (UGS): Advantages and limitations analysis. . . . . . 42

6.1 Summary of performance metrics obtained from three tests. . . . . . . . . . . 81
6.2 Performance metrics of the trained model. . . . . . . . . . . . . . . . . . . . . 84

# Listings

5.1 Minimal model training script. . . . . . . . . . . . . . . . . . . . . . . . . . 27
5.2 Excerpt showing the structure of the server response: a detection entry contains the object's class and ordered corner coordinates. . . . . . . . . . . 30
5.3 Flag-based definition of subsystem requirements. . . . . . . . . . . . . . . . 36
5.4 Method called on start up, allowing authentication into the Unity Gaming Services (UGS). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.5 Abstract `SubsystemManager` implementation. . . . . . . . . . . . . . . . . . 63
6.1 YOLO model training script used for testing. . . . . . . . . . . . . . . . . . 83

# Glossary

**C#** Pronounced “C-Sharp”, it is an object-oriented programming language created by Microsoft, often used for game and simulation development. Is is leveraged by Unity. 24

**firewall** A firewall is a network security device that monitors and controls incoming and outgoing network traffic based on predefined security rules. 23

**NAT** A Network Address Translation (NAT) is a process that allows multiple devices on a private network to share a single public IP address when communicating with the outside world. 23

**prefab** A prefab is a reusable Unity asset that stores a fully configured GameObject, including its components and properties. Once created, it can be instantiated consistently across multiple scenes (or even shared between projects) without the need for reconfiguration. This enables updates from a single source, making it easy to maintain consistency. 36, 53

**RPC** A RPC is a software communication protocol that one program can use to request a service from another program located on a different computer and network, without having to understand that network’s details. In other words, used to call other processes on remote systems as if the process were a local system. 23

**socket** In the context of computing and networking, a socket is an endpoint that allows programs running on a network to “talk” to each other. It acts as a communication channel, enabling data exchange between applications, whether on the same machine or across different networks. 23

# Acronyms

2D Two-dimensional 52, 59, 60, 64, 65, 68

3D Three-dimensional 3, 4, 7, 10, 15, 19, 23, 24, 52, 53, 55, 56, 59, 60, 64, 65, 66, 68, 70, 76

6DOF Six Degrees of Freedom 13

API Application Programming Interface 17

AR Augmented Reality xvii, xviii, xix, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 14, 15, 16, 19, 23, 24, 26, 40, 51, 57, 58, 59, 60, 61, 62, 63, 64, 65, 69, 70, 76, 80, 83, 89

CPU Central Processing Unit 84, 85

FOV Field of View 13

GPU Graphics Processing Unit 84, 85

HMD Head-Mounted Display 7

HUD Heads-Up Display xvii, 8

ICSE International Conference on Software Engineering 5

iOS iPhone Operating System 14, 42

IP Internet Protocol 49

JSON JavaScript Object Notation 30

LiDaR Light Detection and Ranging 14

mAP Mean Average Precision 84

MB Megabyte 85

MR Mixed Reality xviii, 4, 7, 9, 11, 13, 15, 17, 18, 19, 25, 27, 33, 34, 39, 40, 41, 42, 51, 56, 58, 60, 69, 75, 82, 89

NAT Network Address Translation xxv, 31, 41, 48, 50

NGO Netcode for GameObjects 23

PC Personal Computer 14

RGB Red, Green, Blue 66
RPC Remote Procedure Call xviii, xxv, 23, 54, 55, 56

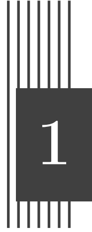
SLAM Simultaneous Localization and Mapping 18
STEM Science, Technology, Engineering, and Mathematics 2, 5, 90

UGS Unity Gaming Services xxi, xxiii, 23, 39, 40, 41, 42, 43, 44, 46, 47, 49, 50, 52
URL Uniform Resource Locator 17, 23

VR Virtual Reality xvii, xviii, xix, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 14, 15, 16, 19, 36, 37, 40, 51, 57, 58, 59, 61, 62, 63, 75, 76, 80, 81, 89

XR Extended Reality xxi, 7, 11, 14, 16, 17, 23, 61

YOLO You Only Look Once xvii, xxiii, 4, 19, 23, 24, 26, 27, 28, 74, 82, 83



# 1 Introduction

***"You are in the story, you speak to the shadows and they reply, and instead of being on a screen, the story is all about you, and you are in it."***

— Stanley G. Weinbaum, in "*Pygmalion's Spectacles*" (1935)

In 1935, in a small science fiction magazine, Stanley G. Weinbaum presented "Pygmalion's Spectacles" [1], where he narrated the story of Professor Haskel and his revolutionary creation: a pair of "magical" virtual reality glasses. When worn, they transported the user to a fully immersive world, where they could "see, hear, smell, taste, and touch" an alternate reality that perfectly mirrored the real-life experience of human sensations. Decades later, Arthur C. Clarke expanded upon this vision in "The City and the Stars" [2], imagining a distant future where entire societies lived exclusively in virtual environments.

In an era where television was only just beginning to reach living rooms, and computers occupied entire rooms, these concepts seemed destined to remain confined to the imagination, inseparably bound to the label of science fiction. No one could have predicted how close some of these visionary speculations were to the reality that would emerge in the 21st century.

Less than a century after these first literary works, the world has entered an era where VR and AR not only exist, but are profoundly redefining the paradigms of interaction between the physical and digital worlds. Weinbaum's "magical glasses" are now materialized in headsets capable of transporting users to entirely new universes, while AR technologies can even simulate the concept of holograms with the projection of digital information directly into the wearer's field of vision, creating an unprecedented fusion between both worlds.

This transformation represents more than a simple technological advancement: it constitutes a revolution in how we perceive and interact. The present landscape is defined by the exponential transformation in hardware, software, and experience design, reshaping sectors from education and medicine to entertainment and industry [3].

## 1.1 Motivation, Goals and Contributions

This dissertation seeks to present the project developed as a Master's thesis, one that emerges within this context: a platform for collaborative mixed reality experiences. While primarily designed to support the creation of immersive learning environments, it stands as a response to some of the most pressing challenges in mixed reality development: device fragmentation and cross-platform interoperability.

This section establishes the foundational context and objectives of this project. Through Subsection 1.1.1, it begins by presenting the educational and technological challenges that guide the work, before outlining the primary goals and specific scientific and technical contributions achieved throughout the research in Subsections 1.1.2 and 1.1.3, respectively.

### 1.1.1 Motivation

To understand why education became a natural focus for this work, it is necessary to consider the inherent limitations faced when trying to explain complex, invisible phenomena. Whether as students or educators, there is a common encounter with the difficulty of conveying concepts with dynamic, interacting components that cannot be directly observed.

Maintaining student attention and focus is one of the main challenges of STEM education, especially when dealing with abstract concepts or systems invisible to the naked eye. While traditional teaching methods have undeniable value, they meet their natural limits when faced with these invisible barriers between knowledge and true understanding [4].

After all, how does one effectively explain the movement of electrons, or the inner workings of an electrical circuit using only analogies or flat diagrams on a board? To truly deepen the link between theory and practice, a critical component is missing: the ability to touch and experience these ideas in three dimensions.

These missing elements are exactly what Augmented and Virtual Reality technologies can bring. With them, students can hold a molecule in their hands, directly visualize current flow in circuit boards, or even safely conduct experiments that would otherwise be too dangerous in conventional classrooms [5] [6] [7]. These methods allow students to become active participants in their learning experience, rather than passive observers.

However, despite clear potential, these technologies remain far from everyday reality in many sectors. One of the main reasons for this is the technological fragmentation. The mixed reality landscape suffers from a fundamental interoperability problem, as existing devices and platforms often operate in isolation, or are "locked" to specific systems [8]. This means that to support different AR and VR technologies developers are often forced to create and maintain separate versions of the same application – a costly and inefficient approach that severely limits scalability.

This is not just a technical inconvenience, as these issues directly shape the user experience and constrain widespread adoption. Because devices differ so widely in capabilities, participants in the same virtual environment often face unequal access: some may lack essential features, while others may be unable to participate at all. The experience itself becomes fragmented, rather than truly shared.

With this in mind, this project aims to address these challenges by developing an open-source, mixed reality platform that allows users with different devices to interact in shared virtual environments. In other words, one that enables true, real-time collaboration, regardless of the users' specific device or actual physical location.

### 1.1.2 Goals

To clearly establish the project's scope, understanding how the collaborative mixed reality experiences function in practice is essential. These experiences can range from theoretical educational simulations (such as exploring molecular structures or visualizing abstract mathematical concepts) to highly interactive gamified environments where users collaboratively solve puzzles or conduct virtual experiments. Users join together to explore, manipulate, and learn from shared content, spanning both real and virtual domains.

As will be further established in Section 2.3, Augmented Reality devices possess cameras that allow users to see the real world while overlaying it with digital objects. To take full advantage of these technologies, many experiences incorporate physical markers. These markers, which can range from simple printed sheets of paper to natural objects and environmental features, function as anchor points in physical space. When a camera-equipped device detects one of these markers, a virtual object spawns at that exact location, becoming visible and interactable to all users in the same virtual session.

For instance, a group may place a marker on a desk and see a 3D molecular structure or an erupting volcano appear above it. A remote user - whether using a Virtual Reality headset or even a laptop - will see these same objects, fully synchronized with the others.

The overarching goal is to create an illusion of shared, real-time presence across different hardware platforms, as if all users are interacting in the same physical room. This requires solving several technical challenges, from cross-platform compatibility to spatial mapping and session synchronization.

The main objectives of this research include:

- **Cross-Platform Interoperability:** Develop a solution that enables different AR and VR devices to interact within the same virtual environment, working towards eliminating existing technological barriers.
- **Real-Time Collaboration:** Allow users with different device types to collaborate effectively, manipulating virtual objects and observing synchronized changes in the shared environment.

- **Modular Architecture:** Create a flexible structure that facilitates future integrations of both new devices and new experiences, without requiring system-wide rework.
- **Educational Enhancement:** Explore how immersive MR experiences can give educators greater freedom to design engaging, interactive, and pedagogically effective learning activities.

### 1.1.3 Contributions

This subsection outlines the key contributions of this research work, encompassing both the scientific and technical advances achieved through the development of the platform, as well as the dissemination of these findings through academic publications and conference presentations.

#### 1.1.3.1 Scientific and Technical Contributions

The developed system includes the following scientific and technical contributions:

- Development of an **architecture leveraging Unity 3D as the development framework** demonstrating seamless interaction between AR and VR device users within shared virtual spaces. Magic Leap 2 and Meta Quest 3 devices serve as exemplars of each device type, respectively.
- Definition of **universal systems that enable users to seamlessly manipulate virtual objects regardless of their device type**, creating the illusion of shared real-time presence across different hardware platforms.
- Implementation of **shared virtual rooms** where users observe real-time scene changes, incorporating marker-based alignment techniques that function as physical anchor points in space.
- Development of a **specialized Python server** utilizing YOLO for optimized marker detection and tracking, enabling complete customization of detection algorithms tailored to specific use cases. This is complemented by the **integration of a WebSocket-based communication protocol**, to ensure seamless real-time data exchange between Unity and the server.
- Creation of a **flexible structure** that both **facilitates seamless integration of new devices** and **enables educators and developers to easily create custom collaborative experiences from scratch**, without requiring extensive technical expertise in mixed reality development.
- **Practical demonstration of the platform's educational applicability**, showcasing how remote users can collaborate in manipulating complex 3D objects across different scientific domains. This is achieved through the detailed implementation of a comprehensive example environment.

Contributing to the open-source community and facilitating future research in collaborative mixed reality, the various components developed during this work are available in public repositories: Unity Project and Python Server.

#### 1.1.3.2 Scientific Publications and Conference Presentations

In addition to the scientific and technical outcomes, this work has also been disseminated through publications and presentations in scientific venues, ensuring broader impact and engagement with the research community:

- **ICSE Science Factory 3rd STEM Education Conference** (Lisbon, June 28, 2025): Abstract accepted and presented, introducing the platform's application to STEM education. The presentation to an audience primarily composed of educators provided direct feedback on the platform's educational potential. Both the abstract and presentation materials are available in an external repository: Presentation Repository.
- **2025 9th International Young Engineers Forum on Electrical and Computer Engineering (YEF-ECE)** (Lisbon, July 4, 2025): Full paper accepted and presented, showcasing the proposed platform and its underlying architecture. The conference, aimed at highlighting innovative research by graduate students in electrical and computer engineering, provided an opportunity to disseminate both the technical foundations and the educational applicability of the system. DOI:10.1109/YEF-ECE66503.2025.11117510.
- **European ICSE Science Factory Conference and 6th International STEM Education Conference** (Istanbul, November 8–9, 2025): Extended abstract accepted for presentation with an approved workshop proposal to be conducted during the conference, providing hands-on experience with the developed platform.

## 1.2 Document Structure

This section provides a comprehensive overview of the document's organization, describing the content and purpose of each subsequent chapter to guide the reader through the structure.

This document is organized into seven chapters, with each one building upon the previous as to provide a comprehensive understanding of the research process, technical solutions, and practical applications.

These chapters are outlined below:

- **Chapter 1 - Introduction**
  This chapter, of which this section forms part, articulates the motivation, objectives, and contributions of the project. The problem of device fragmentation in mixed reality, which the thesis seeks to address, is presented.

- **Chapter 2 - Foundations of Mixed Reality Technologies**
  Establishes the historical and theoretical foundations relevant to the project, including details regarding the evolution of VR and AR technologies, device characterization and relevant definitions.

- **Chapter 3 - Related Work**
  Reviews the current landscape of related research and technological developments for mixed reality, with particular focus on learning contexts. Examines current platforms, frameworks, and approaches, identifying gaps in research to further contextualize the project.

- **Chapter 4 - Proposed Architecture**
  Details the overall design and structure of the developed platform. Presents and justifies the architectural decisions, component relationships, and technical frameworks that underpin the system.

- **Chapter 5 - System Implementation**
  Provides an in-depth description of the implementation process of each system component. Covers and justifies the specific programming solutions, algorithms, and protocols developed to enable real-time collaboration, including marker detection, custom communication flows, and spatial synchronization mechanisms.

- **Chapter 6 - Module Development and Evaluation**
  Demonstrates the platform's practical applicability through the implementation and testing of a collaborative educational experience. This exemplification illustrates the creation workflow, and highlights the potential of the proposed solution in real scenarios.

- **Chapter 7 - Conclusions and Future Work**
  Summarizes the main findings and contributions of the thesis and provides a reflection on both the achievements and limitations of the developed system. Evaluates which of the initial objectives were met and identifies future research and development possibilities.

# 2 Foundations of Mixed Reality Technologies

This chapter begins by providing, throughout Sections 2.1 and 2.2, some background regarding both the historical origins and the current landscape of VR and AR technologies. With this context in mind, Section 2.3 proceeds to make the distinction between these two technologies, allowing a clearer understanding of the unique promises that each offers, and ends with a proper introduction to the concepts of MR and Extended Reality (XR).

## 2.1 Origins of Immersive Technologies

Although the origins of VR technologies are linked to science fiction, their technical development began to take shape in the 1960s.

Morton Heilig, often referred to as the "father of virtual reality", presented the Sensorama in 1962 - a multimodal system that offered a multisensory experience by combining 3D films, stereo sound, vibrations, and even scents [9]. A few years later, in 1968, Ivan Sutherland developed what is considered the first Head-Mounted Display (HMD) system, nicknamed "The Sword of Damocles" [10]. Suspended from the ceiling due to its considerable weight, this system demonstrated, for the first time, the possibility of projecting three-dimensional images that responded to the user's head movements. This is illustrated by Figure 2.1.

Figure 2.1: *Images picturing "The Sword of Damocles". The use of a "a hand-held wand" enabled object interaction, closely resembling modern controller-interface paradigms.*

Despite the technical limitations of the time, these innovations established a crucial precedent: the conviction that technology could create truly immersive experiences that engaged with multiple senses simultaneously, transcending the limitations of traditional two-dimensional interaction.

Parallel to VR development, a complementary yet conceptually distinct concept was emerging: Augmented Reality. Rather than completely "replacing" the perceived reality, this technology sought to enrich it through the overlay of digital information.

In a way, similar concepts had existed decades before: Heads-Up Displays (HUDs) used in military aviation since the 1960s can be seen as a primitive, yet functional, form of AR [11]. These projected critical data, such as altitude or speed, directly onto the windscreen, allowing pilots to access information without having to divert their gaze. In Figure 2.2, some examples of this concept are shown.

Figure 2.2: *Pictured are several examples of the use of HUDs in aviation: on the left, a U.S. Navy jet can be seen aligned in the HUD of another aircraft; the two images on the right show how flight-related data can be displayed to aid the pilot.*

The most significant theoretical development came from Ronald Azuma [12], who in 1997 established a definition of AR based on three fundamental characteristics:

- **Overlaying "virtual" onto "real":** creating a single coherent image;
- **Real-time interaction:** responding immediately to user actions when interacting with virtual objects;
- **Three-dimensional registration:** ensuring that elements remain coherently anchored in physical space.

This definition continues to guide the development of the field to this day.

## 2.2 Current State and Future Expectations

By the end of the 20th century, these technologies remained largely confined to university research laboratories and specialized military applications: this was because, although technically impressive and conceptually revolutionary, they were prohibitively expensive and too complex for widespread use, thus remaining as curiosities with unrealized potential [13].

The true renaissance of VR and AR began in the 2010s, driven by a remarkable convergence of technological factors, notably: exponentially more powerful processors, high-resolution

displays, increasingly precise motion sensors, and crucially, the miniaturization of these components into commercially viable devices [14] [15] [16].

The launch of the Oculus Rift in 2016 [17] stands out as the catalyst for the beginning of a new era for VR, marking the first time this type of technology became genuinely accessible to the common consumer, with prices in the hundreds rather than thousands of euros.

The global phenomenon of Pokémon GO [18] perfectly demonstrated the potential of AR technologies, introducing tens of millions of users to these concepts through a playful and intuitive experience that redefined expectations about digital interaction in a physical space [19] [20]. With the continuous evolution of smartphones, AR is literally at the tips of consumers' fingers, and users, often without realizing it, interact with these technologies daily: social media filters or navigation applications with digital overlays have become a natural part of the everyday experience [21].

Today, these technologies are no longer mere promises: they are gradually being adopted across the most varied sectors, with technology giants like Meta, Apple, and Microsoft investing billions in developing increasingly sophisticated devices [22]. AR, in particular, has found practical applications in navigation, education, and social media, while VR establishes itself firmly in gamification, immersive training, and medical therapies [23] [24].

As these technologies mature and integrate into daily life, it is possible to anticipate even more profound changes: new ways of working, learning, communicating, and experiencing the world around us. What began as imaginary visions and science fiction is now shaping the future of human-machine interaction, creating paradigms that transcend traditional physical limitations.

## 2.3 Definitions and Distinctions

To fully understand the implementation of the solutions presented in this dissertation, it is fundamental to clearly distinguish the characteristics and differences between VR and AR devices, as well as to understand where the emerging concept of Mixed Reality (MR) fits.

### 2.3.1 Virtual Reality: Complete Immersion

Virtual Reality uses headsets that completely block out the real world, replacing it with digitally generated images. When the device is worn, the user's entire field of vision is filled with the virtual scene, creating the sensation of being physically present in another environment.

The characteristics of VR devices include:

- **Visual and auditory isolation:** Providing complete sensory immersion with internal lenses, high-resolution screens, and headphones with spatial sound that muffle external noise.

- **Motion tracking:** Capturing both rotation (*pitch*, *yaw*, *roll*) and displacement ($x$, $y$, $z$) of the head and hands, with the latter task aided by the use of controllers.
- **Low latency:** The system must respond almost instantly to movements (typically $< 20$ ms) to maintain the sense of presence and avoid discomfort [25].

Figure 2.3 illustrates some popular examples, including Meta Quest [26], HTC Vive [27], or Valve Index [28] devices.

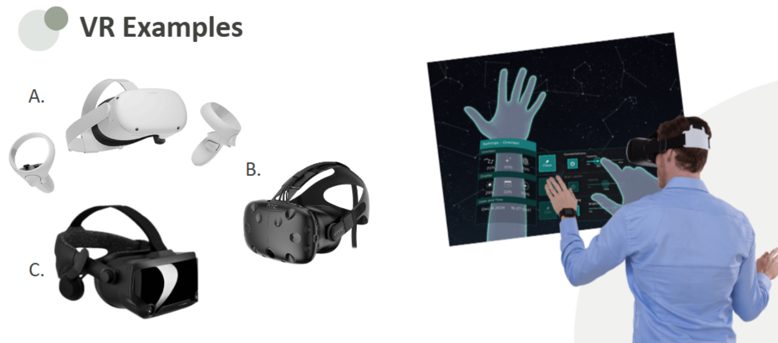



Figure 2.3: *Some popular VR device choices are shown: A. Meta Quest 2, B. HTC Vive Pro and C. Valve Index (Original). A simple exemplification of use is also shown: this technology completely immerses users in a virtual world.*

### 2.3.2 Augmented Reality: Fusion Between Real and Virtual

Augmented Reality, by contrast, preserves the perception of the physical world while enriching it with overlaid digital elements, with Azuma's classic definition still holding relevance.

AR devices introduce the following characteristics:

- **Hybrid visualization:** The user sees the real world and, simultaneously, virtual objects aligned with the environment. This can be achieved in two ways:
  - **"See-through" lenses:** use of transparent lenses, onto which virtual holograms are projected (e.g. Microsoft HoloLens [29], Magic Leap [30]).
  - **Based on video "see-through":** use of cameras to capture the real world, with the image displayed alongside virtual elements on a screen - this setup is particularly common in AR smartphone applications, such as Pokémon GO.
- **Hand tracking:** Cameras and depth sensors allow the precise detection of hand movements, enabling direct interaction with virtual objects.
- **Spatial tracking and registration:** Those same cameras and sensors continuously capture the 3D structure of the environment to accurately register virtual elements within the real world.
- **Realistic visual integration:** In more recent devices, emphasis is placed on using sensors to simulate shadows, occlusion, and reflections to blend naturally with the real scenery [31].

Figure 2.4 showcases some of the mentioned examples.

Figure 2.4: *Some AR technologies can be seen: A. Holo Lens Pro 2, B. Magic Leap 2 and C. Google Glass Enterprise Edition 2. To the right, a smartphone-based application (Pokémon Go) is also shown: this technology overlays virtual objects atop real environments.*

### 2.3.3 The Concept of Mixed Reality

In 1994, Paul Milgram introduced the concept of the "virtuality-reality continuum" [32]. As presented in Figure 2.5, different technologies are positioned on a continuous spectrum between two extremes: physical reality and complete virtuality (pure VR).

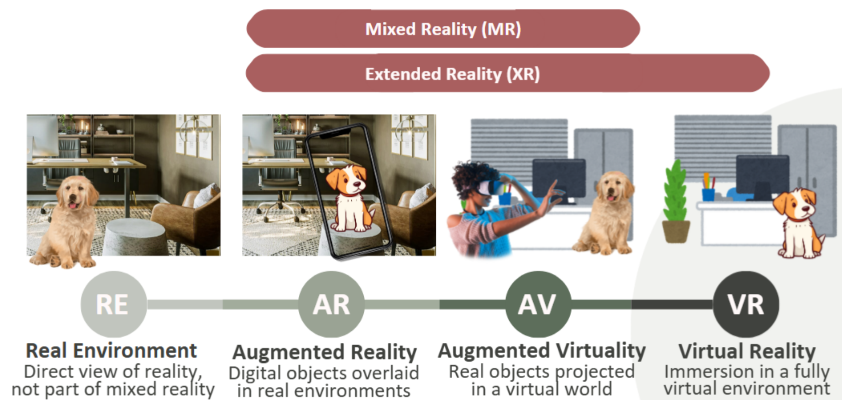



Figure 2.5: *Virtuality-Reality spectrum as proposed by Paul Milgram. The illustrated diagram showcases examples of various technologies, and defines the range of some relevant terms.*

The concept of Mixed Reality (MR) refers to any intermediate point along this spectrum where real and virtual elements coexist and interact.

Within this category, both traditional AR (predominantly real, with virtual additions) and scenarios of so-called "augmented virtuality" (virtual environments enriched with real elements) can be found. In recent years, the term Extended Reality (XR) started being used as an umbrella term that also encompasses pure VR.

This perspective helps understand that the boundaries between technologies are more fluid than rigid categories, paving the way for a wide variety of hybrid experiences.

# 3 Related Work

This chapter provides an in-depth look into the current landscape and relevant approaches in MR development.

Section 3.1 begins by exploring modern hardware solutions, examining their varying capabilities and implications for collaborative experiences while Section 3.2 analyzes existing educational platforms and applications, revealing critical gaps in the market. Section 3.3 examines current frameworks designed to address interoperability challenges, highlighting both their contributions and limitations, and Section 3.4 proceeds to outline popular spatial alignment techniques, with particular emphasis on their pedagogical implications.

Finally, Section 3.5 synthesizes these findings to identify the gaps and opportunities that guided the system design and architectural decisions presented in subsequent chapters.

## 3.1 Overview of Current Existing Devices

The rapid evolution of MR technology has produced a diverse ecosystem of devices, each with fundamentally different capabilities, interaction paradigms, and technical constraints. Understanding these differences is crucial to determine where the challenges faced in cross-platform development take root.

Building on the technological characteristics outlined in Section 2.3, an examination of state-of-the-art devices in each category helps to better understand their strengths and limitations within the current technological landscape:

**A) Virtual Reality Headsets:**
This category is dominated by standalone headsets (Meta Quest 3, Apple Vision Pro [33]) and high-end tethered systems (HTC Vive, Varjo Aero [34]).

- **Strengths:** Complete immersion, precise Six Degrees of Freedom (6DOF) tracking, wide Field of View (FOV) (90-120°).
- **Limitations:** Full physical isolation, heavy form factors, limited battery life (standalone, about 2-3 hours), potential to cause motion sickness.

**B) Augmented Reality Glasses:**
Microsoft HoloLens 2 and Magic Leap 2 lead this category, while lightweight options focused on display overlay (Nreal Air [35], Rokid Max [36]) have recently emerged.

- **Strengths:** Real-world integration, hands-free operation, persistent spatial anchoring, social acceptability in professional environments [37].
- **Limitations:** Narrow field of view (40-50°), limited battery life, higher cost, brightness constraints in outdoor environments.

**C) Mobile AR Devices:**
Movile devices with ARKit (iOS) [38] and ARCore (Android) [39]. Systems like the iPhone 15 Pro with Light Detection and Ranging (LiDaR) offer enhanced depth sensing [40].

- **Strengths:** No additional hardware investment required, high-resolution displays, widespread social acceptance.
- **Limitations:** User fatigue from handheld use, touch-only interaction, battery drain, reduced screen visibility in bright conditions.

**D) PC-Tethered Systems:**
VR systems (Valve Index, Varjo) that require a connection to a high-end Personal Computer (PC).

- **Strengths:** Unlimited computational power, sub-millimeter tracking accuracy, no battery constraints, advanced haptic feedback.
- **Limitations:** Restricted movement, complex setup requirements, high cost, dedicated space requirements, technical expertise needed.

Table 3.1 summarizes these categorical differences across key dimensions:

Table 3.1: Comparison of Extended Reality (XR) Device Families

| Category | Input Methods | Cameras | Display Type |
|---|---|---|---|
| VR Headsets | Controllers, Hand tracking | Limited/None | Full immersion |
| AR Glasses | Hand/Eye tracking, Voice | Multiple sensors | Lens Overlay |
| Mobile AR | Touch, Device motion | Single/Dual camera | Small screen |
| PC-Tethered | Controllers, External tracking | Variable | High resolution |

Examined side by side, the diversity of these devices becomes clear: some prioritize hand tracking, while others rely solely on controllers; camera setups range from non-existent in basic VR to sophisticated multi-sensor arrays in advanced AR; and even display types span from handheld screens to wide-field immersive projections.

This diversity creates a challenging development scenario. Writing custom logic for each device category (let alone individual device) becomes unsustainable, particularly given the exponential rate at which new models enter the market. However, treating all devices

identically undermines their unique strengths and results in lowest-common-denominator experiences that fail to fully leverage their capabilities.

For the practical scope of this dissertation, the Magic Leap 2 and Meta Quest 3 were used as representative exemplars of AR and VR devices, respectively. Focusing on these two platforms enabled an exploration of cross-platform challenges while reflecting the broader trends and constraints observed across the current MR device ecosystem.

## 3.2 Current Mixed Reality Applications

MR technologies have found applications across diverse sectors, from industrial training and medical simulation to entertainment and education. Given the focus of this project, the latter warrants specific examination.

Recent studies demonstrate that immersive pedagogical approaches enhance learner motivation, deepen conceptual understanding, support long-term knowledge retention, and foster critical thinking [41] [42]. As such, several applications have emerged across key educational domains, a few examples of which are presented below:

- **Chemistry and Biology:** Nanome [43] (Figure 3.1a) enables students to collaboratively explore and manipulate molecular structures in VR. Labster [44] offers a suite of interactive science labs, allowing virtual experiments in biology and chemistry without the safety or resource constraints of physical labs.
- **Physics and Engineering:** PhET Simulations [45] recently extended into AR and VR, enabling visualization of complex physical phenomena such as electric fields or wave interference. zSpace [46] (Figure 3.1b) provides MR labs where students can dissect virtual machines, test engineering principles, and interact in real time.
- **Mathematics and Geometry:** GeoGebra AR [47] (Figure 3.1c) allows students to visualize functions, surfaces, and 3D shapes directly in physical space through mobile AR, making abstract concepts more intuitive. MEL Science [48] supplements this with focus on mathematical modeling.
- **History and Social Studies:** Platforms such as VictoryXR Campus [49] and TimeLooper [50] create immersive environments where students can experience historical events from multiple perspectives, or take "virtual field trips" exploring reconstructed historical sites or cultural landmarks.

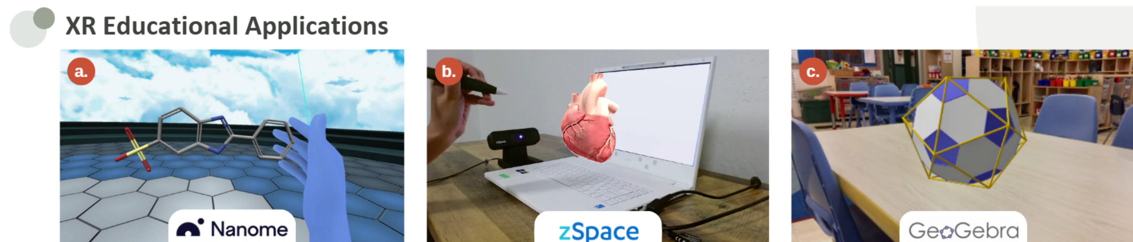



Figure 3.1: *From left to right: a user handles a molecular structure in Nanome, an interactable heart model is shown in the zSpace setup, a virtual representation wraps around a real soccer ball detected using Geogebra.*

However, these applications must be analyzed not only for their educational content but also for their platform availability and collaborative capabilities. Table 3.2 summarizes the results of this analysis and situates the platform resulting from this dissertation within the same context.

Table 3.2: Educational XR Applications Comparison

| Application | Device Family | Collaboration | Price Range |
|---|---|---|---|
| Nanome | VR | Real-time | $500-2000/year (institutional) |
| Labster | Web + VR | Asynchronous | $200-800/year (individual) |
| PhET Simulations | Web + AR/VR | Limited | Free |
| zSpace | Proprietary XR | Real-time | $4000-8000 (per setup) |
| GeoGebra AR | Mobile AR | None | Free |
| MEL Science | Mixed | Limited | $30-50/month (individual) |
| VictoryXR Campus | VR | Real-time | $10-30/month (individual) |
| TimeLooper | Mobile AR | None | $5-15/experience (individual) |
| Proposed Platform | Aproprietary XR | Real-time | Free (open-source) |

This closer analysis of the existing applications reveals a significant limitation: while these may succeed individually, they are tightly coupled to specific hardware ecosystems. Schools rarely maintain perfectly uniform device inventories, and even devices within the same "family" often prove incompatible due to different generations, operating system versions, or hardware specifications.

This reality creates immediate barriers to inclusive classroom experiences. Students using different devices cannot collaborate meaningfully, and educators are forced to commit to specific hardware ecosystems rather than leveraging the diverse technologies that may be present in their classrooms. Institutions become locked into whatever educational content exists for their chosen platform, and forced to work within a limited application pool rather than selecting the best tools for their curriculum. The very technology that promises to democratize education risks reinforcing existing inequalities instead [51].

Finally, to position the proposed work within this context, the project focuses on developing a **real-time, collaborative XR platform that is broadly accessible across different devices, while remaining free and open-source**. By prioritizing cross-platform compatibility and user-centered design, this project aims to enable inclusive, interactive learning experiences without tying institutions to a single hardware ecosystem, directly tackling the limitations identified in current solutions.

## 3.3 Cross-Platform Frameworks and Standards

The findings from the previous section raise a critical question: can meaningful collaboration actually be achieved when device capabilities and constraints vary so dramatically? In this section, focus will be shifted into exploring the efforts that have been put into solving these problems.

To understand this challenge, an analogy may be used: a speaker attempting to communicate with a room full of people, each speaking a different language. A "global translator" - one that accounts for all these linguistic differences and the specific ways each participant can perceive the information. This describes the goal that cross-platform MR frameworks like OpenXR [52] and WebXR [53] have emerged to address. These serve as abstraction layers that provide unified interfaces for:

- **Input Abstraction:** Unified handling of controllers, hand tracking, and gesture recognition across different hardware.
- **Display Management:** Consistent rendering regardless of underlying technology.
- **Tracking Systems:** Common APIs for spatial mapping and user movement tracking.

It is very important to note that, while this abstraction significantly helps, it does not completely solve the stated issues. These frameworks reduce boilerplate code and ease the integration of new devices, but they don't address the user experience disparities that arise when different device types attempt to share the same virtual environment.

In other words, developers no longer need to understand device-specific implementations, they don't need to know how to make information understandable to all devices, but they still need to determine what information should be shared and how collaborative interactions should be designed. Interactions and behaviors specific to the collaborative system - the very illusion of shared space - must be thought, designed and implemented. This should be the focus point of the proposed work.

When comparing the two main standards, WebXR and OpenXR, the distinction mainly lies in their intended scope and environments. WebXR is designed to bring XR experiences to the web, making them accessible through a simple Uniform Resource Locator (URL) without requiring dedicated installations. This makes it particularly suited for lightweight applications, such as sharing browser-based virtual museum tours or a classroom demonstrations. However, this accessibility comes at the cost of performance and limited access to lower-level device features, making it less suitable for resource-intensive collaborative environments.

OpenXR, in contrast, provides a native runtime environment with direct hardware access, exposing detailed device capabilities and offering developers the fine-grained control necessary for implementing more complex collaborative mechanisms. Given this dissertation's focus on addressing the challenges of cross-device collaboration and exploring the

customization of interaction paradigms, OpenXR's extensive documentation, and strong Unity integration make it the more suitable foundation.

## 3.4 Spatial Mapping and Alignment Techniques

This section gives a brief overview of some existing solutions regarding spatial alignment and object tracking.

For shared MR experiences to feel convincing, users must perceive virtual objects as consistently anchored in their physical surroundings: say a user is looking at a certain point, and a digital object appears directly in front of them; upon head movement, the object must "stay in place" in relation to the real world, and not get "stuck" in relation to the user's viewpoint. This requires alignment techniques that synchronize the digital and real worlds across devices. Two broad families of approaches can be highlighted:

- **Markerless Methods (e.g. Simultaneous Localization and Mapping (SLAM)):** These algorithms allow devices to use maps of the environment while tracking their position within it. Systems like Azure Spatial Anchors create persistent global coordinate references that can be shared across multiple users and sessions.

  While this approach can create seamless integrations, it demands stable environment conditions, high computational power, and reliable network connectivity - factors which, in classrooms or resource-constrained settings, may not always be available.

- **Marker-based Methods (e.g. ArUco [54], AprilTags [55]):** This technique uses physical markers to establish known reference points in physical space (Figure 3.2). When cameras detect these markers, they can immediately determine its position and orientation within a relative coordinate system [56].

  Although this solution may struggle with occlusion or certain viewing angles, it represents a low-cost and simple alternative.

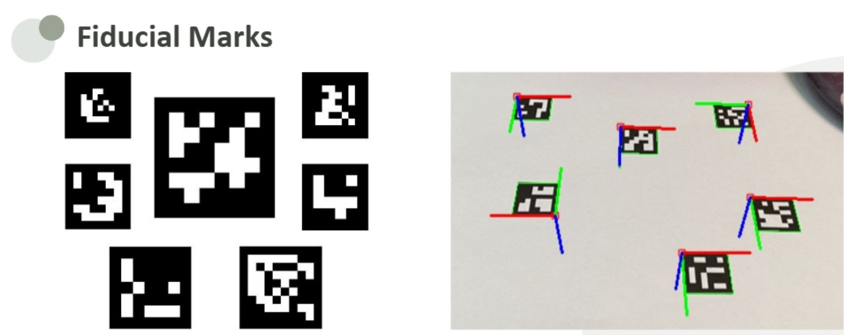



Figure 3.2: *Examples of ArUco markers are illustrated. To the right, a practical use can be seen: upon detection, the markers' position and orientation can be estimated and displayed.*

A particular pedagogical benefit can be extracted from the marker-based solution: it allows full advantage of the combination of real and digital content. There's significant potential to be explored here: students can look at a molecular structure diagram in a textbook and see it become three-dimensional, then manipulate and reconstruct it with their own

hands; historical artifacts could be examined as images in books before materializing as interactive 3D models that students can rotate, dissect, or even "step inside" to explore.

However, it becomes clear that using general-purpose fiducial markers like AprilTags or ArUco would limit such a range of possibilities: as can be seen in Figure 3.2, these are simple geometric patterns rendered in black and white designed for quick detection rather than content integration with real meaning.

With this in mind, this project integrates You Only Look Once (YOLO), a deep-learning object detection framework, for more robust and adaptable object tracking. Detection models can be trained to track any desired object, enabling full customization, and allowing everyday educational materials to double as spatial reference points within the immersive environments.

The technical implementation, training methodology and ease of customization will be detailed in Section 5.1, but this contextual foundation helps explain the reasoning behind architectural decisions.

## 3.5 Identified Gaps and Research Opportunities

To summarize the findings from this section, the following key gaps can be identified:

- **True Cross-Platform Collaboration:** Despite the proven educational value, very few solutions enable seamless, real-time collaboration across both AR and VR devices without degrading functionality. Platforms either remain confined to a single ecosystem or sacrifice device-specific strengths for compatibility, limiting the richness of shared experiences.
- **Spatial Synchronization Protocol:** Most spatial alignment solutions are optimized for single-device scenarios rather than multi-user collaboration. It is imperative to build an architecture and communication protocol that allows a lightweight pipeline of object detection and tracking, taking into consideration varying computational constraints.
- **Support for Custom-Made Experiences:** Most existing MR frameworks and applications emphasize general-purpose interactivity. This forces educators to adopt pre-packaged content and work around rigid interaction models. Focus should be given on providing a pipeline that allows educators to easily tailor experiences to specific learning goals, maximizing pedagogical impact and allowing for more creativity.

Taken together, these gaps reveal a broader architectural challenge: one that calls for new approaches to platform design, device integration, and collaborative interaction.

This dissertation takes that challenge as base focus, with the following chapters detailing the implementation of the proposed system and unpacking the reasoning behind its most important architectural decisions and building blocks.

# 4 Proposed Architecture

Achieving the goals outlined in Section 1.1 requires more than just application design: it is necessary to build an underlying infrastructure capable of not only allowing the educational experiences catalogue to expand over time, but that also enables the easy integration of new devices without having to start from scratch, thus addressing the fragmentation concerns.

Figure 4.1 presents the model proposed in this work: the project adopts a modular client-server architectural model, aiming to optimize the separation of concerns between interaction logic and heavier computational tasks.

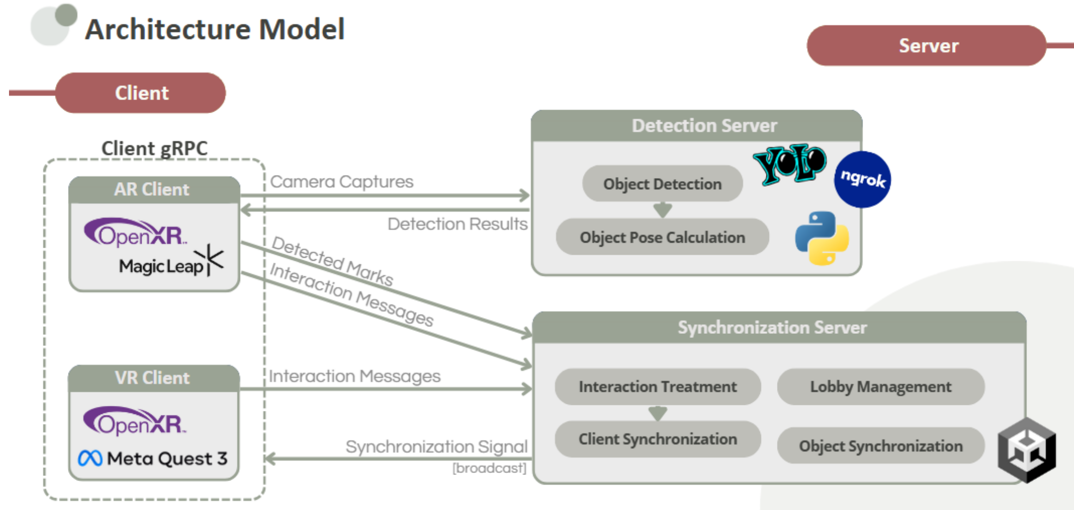



Figure 4.1: *Proposed architecture model depicting the main components along with their primary tasks, as well as the technologies and communication protocols employed.*

The present chapter provides, throughout Sections 4.1 and 4.2, a brief overview of each main component to provide clarity on the division of responsibilities, communication requirements, and leveraged technologies. Section 4.3 then culminates in a general overview and justification of the key design decisions.

As a note, specific implementation and code-related details regarding these components are set aside to be discussed in Chapter 5.

## 4.1 Client Side - Device Management and Interaction

The client side manages direct user-device interaction logic, this includes tasks such as:

- **Data Management:** Storing user data, preferences, and device specifications.
- **Visualization Logic:** Rendering interface components and tracking head movement.
- **Input Handling:** Processing hand movements, specific gestures or button clicks.
- **Subsystem Management:** Controlling movement, tracking, and camera systems based on application needs and permissions.
- **Camera Management:** When camera functionality is available, continuously capturing real-time images and transmitting them to the detection server for analysis.

Strictly speaking, this "side" is physically represented by the devices themselves, whether that's MagicLeap2 glasses, MetaQuest3 headsets, or even a simple tablet. As such, it is supported by each manufacturer's specific technology.

To paraphrase the findings from Chapter 3, each device has vastly different characteristics, such as different cameras and sensors or different interaction paradigms. So, to meet the enumerated tasks, it is necessary to leverage an intermediate technology layer to allow development without the need to understand the specific inner workings of each device. As was previously stated and justified, OpenXR is leveraged for this purpose.

Implementation details for this component are found in Section 5.5.

## 4.2 Server Side - Multiplayer and Experience Management

As a whole, the server side, handles all computationally intensive tasks. It comprises the two right-hand components illustrated in Figure 4.1 - the synchronization server and the detection server -, each addressing a distinct responsibility. This specialized dual-server design stems from the two primary technical responsibilities that emerge when examining the project goals established in Chapter 1:

- **Experience Management:** Which encompasses the project's core functionality - connecting users in shared virtual rooms and sustaining the illusion of shared space through real-time synchronization;
- **Object Detection:** For camera-equipped, the system must instantly recognize and locate physical markers in the captured frames during certain experiences.

This distinction leads to the implementation of a synchronization server leveraging Unity Technologies - which is further described in Subsection 4.2.1, and a Python-based computer vision server for object detection - in turn explored in Subsection 4.2.2.

### 4.2.1 Unity Server - Experience Management and Synchronization

The real-time coordination of the collaborative experience is one of the core responsibilities of the project: when a student moves a virtual object in their interface, everyone else

connected to the session needs to see that change instantly, with precise synchronization, as if standing in the same room. In other words, this first server acts as the central authority, or a maestro in an orchestra: it ensures all participants keep to the same rhythm, regardless of their equipment or location.

Implemented using Unity Technologies [57], a industry standard for XR development, providing robust tools for real-time 3D rendering and networking. Some built-in features, such as Unity Gaming Services (UGS) and Netcode for GameObjects (NGO), are used as intermediate layers, essential for user authentication and study group coordination, and to assert general real-time synchronization [58] [59].

The server continuously receives interaction updates from clients via Remote Procedure Calls (RPCs), learning how they are interacting with the content: for instance, what they detect with their cameras, which objects they grab, or where objects are moved to. These changes are relayed to all connected users in real time, using the same RPC mechanism.

More information regarding this implementation can be found in Section 5.3, where an in-depth explanation of what Unity Gaming Services entail will also be provided.

### 4.2.2 Python Server - Object Detection

To manage the second task, regarding image detection tasks, a Python-based computer vision server was developed. To put things simply, AR devices send captured camera frames to this server at a fixed rate. The server then processes them using a custom-trained YOLO model and returns detection information to the client. It is the equivalent of having a specialist capable of looking at any image and immediately responding: "this image has a marker of *this* kind, in *this* position, with *this* exact orientation".

This means the server may receive near-continuous requests from multiple users simultaneously. To avoid blocking or delay, the server provides asynchronous client-server communication through persistent WebSocket connections, with each client being managed by a separate parallel socket [60]. This architecture allows multiple requests to be processed concurrently, maintaining responsiveness under typical usage conditions.

Furthermore, the server is exposed using Ngrok [61], a tool that provides secure, temporary URLs that make the server accessible from any location without the need of complex network setups, even if behind firewalls or Network Address Translations (NATs).

Finally, while inference speed is inherently dependent on the available resources, the implementation focuses on minimizing computational overhead, making the server efficient even without requiring excessive hardware. The design choices that support this are detailed in Section 5.1, which also provides additional information on model training and usage.

## 4.3 Final Considerations

The division of the server side goes beyond mere convenience: it allows each individual server to utilize optimal technologies and programming languages for their specific tasks.

The synchronization server is implemented using Unity Technologies, which in turn leverages C# as the programming language. Unity itself is a widely adopted platform for real-time 3D rendering and networking, providing a well-established ecosystem for demanding synchronization. In turn, C# contributes with the high performance and memory management capabilities necessary to sustain these real-time operations [62].

On the other hand, the object detection task is accomplished using a Python-based approach. Python offers a vast range of libraries for machine learning, computer vision and image manipulation. In the context of the project, given that the detection and classification processes rely on YOLO models, this language provides the optimal environment with significantly less boilerplate code than other alternatives.

This hybrid approach reduces code complexity by utilizing each language for its strengths: C# for high-performance real-time operations and Python for analysis and machine learning tasks. Beyond the server-side advantages, the overall modular client-server design offers other significant benefits:

- **Resource Optimization:**

  By offloading intensive tasks to servers, devices can remain computationally lightweight, focusing only on user interaction and basic management.

  This preserves battery life and reduces overheating (both inherent concerns of AR devices), and allows cheaper, less powerful systems to still deliver optimized experiences. This democratizes access across different economic segments and varying budget constraints, as high-end processing units are less of a requirement.
- **Fault Isolation:**

  The separation of the system in various blocks ensures that failures in one component do not cascade to affect the entire system.

  For example, if the Python detection server experiences issues or even goes offline, users can still connect to shared virtual rooms and interact with virtual objects through the Unity server, they simply lose object detection functionality temporarily.
- **Scalability and Future Development:**

  Similarly, each part of the system can be scaled and upgraded independently. As machine learning models improve or enhancements to collaborative features emerge, updates can be deployed exclusively on the server-side without requiring any device-specific software updates or hardware replacements.

  Additionally, different teams can work simultaneously on each system without interdependency conflicts. This parallel approach accelerates development and eases expertise application where it is most beneficial.

The following chapter will focus on detailing the complete implementation of the proposed model.

# 5 System Implementation

This chapter provides a comprehensive overview of the implementation process. Rather than following the chronological development timeline, the explanation becomes more coherent when taking a modular approach, beginning with the most standalone components and progressing through those with increasing dependencies.

The implementation description is structured as follows: Section 5.1 details the standalone Python server, examining the employed models and asynchronous behaviors in depth. Sections 5.2, 5.3, and 5.4 explore various aspects of the Unity server, covering scene management, multiplayer functionalities, and object alignment and synchronization, respectively. Lastly, Section 5.5 describes the client architecture component, focusing on device-centered implementations and subsystem manipulation.

Each section emphasizes key design decisions and concludes with a brief analytical summary to highlight the reasoning behind those choices.

As a final note, to further support the discussion of specific classes and their relationships, diagrams depicting the most relevant dynamics are provided throughout the document.

## 5.1 Python Server for Custom Object Detection

The implemented object detection application operates with a straightforward purpose: it receives requests from clients, processes the provided frames through specialized detection models, and returns any relevant findings.

Although the underlying goal within the project's context is to detect objects that serve as anchors for the Mixed Reality experiences, it is important to note that this system maintains relative independence from the rest of the system. Requests are processed without explicit knowledge of their origin or context.

This agnostic design principle ensures that the server remains modular and reusable across various scenarios, and can easily adapt to future use cases or changes to other components

without major rework. However, it also introduces significant challenges, as the system must handle different formats, resolutions, and quality levels inherent to this flexibility. The input could originate from any source: real-time AR streaming, standalone cameras, or even prerecorded videos, for example.

During implementation, three key questions needed to be addressed:

- How can customized detection models be trained, while maintaining high accuracy?
- How should the system handle heterogeneous input formats and provide standardized, agnostic responses?
- How can the server architecture support concurrent users while maintaining accessibility across diverse network configurations?

This section seeks to answer these questions, starting with the introduction of the You Only Look Once (YOLO) model architecture and training approach in Subsection 5.1.1. Subsection 5.1.2 then specifies the Python application design, with the key decisions and benefits being highlighted in Subsection 5.1.3.

### 5.1.1 Model Overview and Design

As has been previously stated, the detection process relies on YOLO models. These are projected to allow near-immediate object detection in both videos and images, standing out from traditional two-stage detection approaches by processing frames with a single pass-through, obtaining both bounding boxes and probabilities simultaneously. Compared to traditional two-stage detectors, this approach offers a significant speed advantage. However, it is worth noting that inference speed depends on the model's complexity: larger, more sophisticated YOLO architectures can achieve high accuracy even in complex or low-resolution scenarios, but at the cost of slower inference.

These models can be adapted to a variety of tasks: from simply identifying which objects are present in an image to locating their specific coordinates. For this project, a segmentation-trained variant was adopted - rather than providing rectangular, axis-aligned bounding boxes, these models generate pixel-level masks that capture exact object shapes and contours. An example of this distinction is shown in Figure 5.1.

Figure 5.1: *Examples of YOLO detection formats: on the left, a bounding box is constructed from the detected corners; on the right, a pixel-level mask captures all detected points.*

This segmentation capability proves essential for supporting the objectives outlined in Chapter 1. Within the context of the Mixed Reality experiences, the system must determine not only the presence of the real-world target objects but also their three-dimensional position and orientation. Therefore, the two-dimensional coordinates provided by the detection server are only an intermediate step in this anchoring process, making such a high level of detail crucial.

#### 5.1.1.1 Training Process

Regarding the training process, it was essential to prioritize ease of customization, allowing specialized models to be prepared and deployed for different educational contexts without requiring extensive machine learning knowledge.

This is another key advantages of using YOLO lies: the relative simplicity of its training and customization pipeline. The process can be summarized as follows:

1. **Dataset Preparation:** The process begins with collecting a training set of images containing the target objects (similar to those shown in Figure 5.1). For improved performance, these images should ideally capture variations in lighting, viewpoints, and backgrounds, and may contain multiple instances of a same object. While larger datasets generally provide better results, as few as 20–30 object instances per class can be sufficient for simple tasks.

   Annotation tools such as Label Studio [63] can then be used to draw the masks that outline the objects, as well as assigning a class/identification to each one. This step is needed to “teach” the model what to look for.

   While the annotation process can be time-consuming, it is the only stage that requires significant manual effort. These programs output a structured directory and a `.yaml` file that can be fed directly into the training pipeline.

2. **Model Training:** A simple training script (with a basic example provided in Listing 5.1) can then be executed to fine-tune a model to detect the desired objects.

```
1  # 1 :: Load a pre-trained model
2  model = YOLO("yolov8n-seg.pt")  # '-seg' for segmentation
3
4  # 2 :: Train on custom dataset
5  model.train(
6      data="dataset.yaml",    # Path to obtained dataset file
7      epochs=100,             # Number of training epochs
8      imgsz=640,              # Image size
9      batch=16,               # Batch size
10     name='example',         # Model name (opt.)
11 )
12
13 # 3 :: By default, the trained weights are automatically saved in:
14 #       'runs/segment/train/weights/best.pt'
```

Listing 5.1: Minimal model training script.

The reason why high accuracy can be achieved even with relatively small, domain-specific datasets lies in the use of *transfer learning*. This technique builds upon pre-trained YOLO weights that were originally trained on large, generic datasets, meaning the model has already learned to capture fundamental visual features and how to distinguish general object patterns.

For efficiency, it is recommended to use "nano" base variants such as `YOLOv8n` or `YOLOv11n`, which are optimized for real-time or resource-constrained environments. These feature simpler architectures and fewer parameters, reducing computational demands and speeding up inference. Although slightly less accurate than larger variants, applying transfer learning on domain-specific datasets allows them to achieve comparable performance while remaining significantly faster and resource-efficient.

Upon training completion, the resulting model weights are saved and ready to be used for the project tasks.

In short, the training pipeline is designed with accessibility in mind. Researchers or educators can create custom models by simply preparing a small set of manually annotated images, obtaining the designated directory structure, and running a single script.

This accessibility aligns with the system's goal of enabling different experiences to leverage different models, as both the base models and the training parameters can be tuned to accommodate specific needs and use cases: for example, scientific applications may benefit from more sophisticated bases with extended training iterations, while the lightweight "nano" variants are ideal for real-time interactive scenarios that prioritize responsiveness.

### 5.1.2 Technical Implementation

Focus may now shift towards exploring the technical implementation of the detection server. Figure 5.2 illustrates the relevant classes.

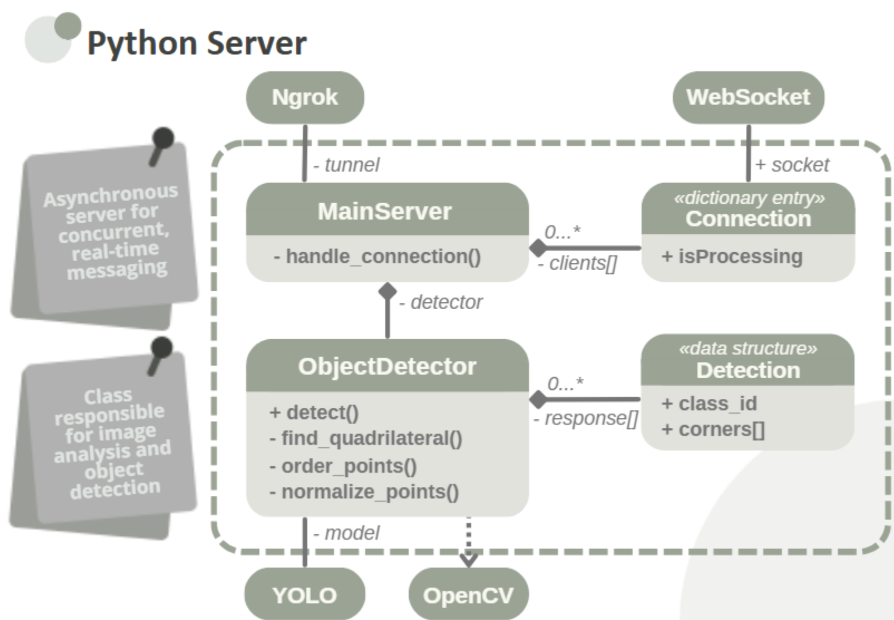



Figure 5.2: *Class diagram of the implemented object detection server.*

The application centers around the `ObjectDetector` class, which encapsulates the core computer vision functionality (detailed in Subsection 5.1.2.1). Meanwhile, the WebSocket server (`MainServer`) manages client communications and coordinates concurrent requests (see Subsections 5.1.2.2 and 5.1.2.3).

Almost all represented external tools have been presented, with OpenCV [54] being the only exception. This open-source library provides essential image processing capabilities including format conversion, resizing and geometric analysis, all fundamental operations to support the detection pipeline.

#### 5.1.2.1 Detection Process and Response Format

The `ObjectDetector` class provides a single public method, `detect()`, which encapsulates the entire request-processing workflow. A description of each step is provided below:

1. **Request Reception:** The client request includes:
   - An image frame which, at this stage, has already been decoded and converted to a supported format (JPEG, PNG, etc.).
   - An identifier that specifies which existing model to use.
2. **Image Preprocessing:** To ensure consistency during inference, images are then resized to match the input dimensions specified by the chosen model. Resizing may sometimes cause loss of detail, making the resulting images look "pixelized" or "stretched" in some way. To mitigate this, OpenCV's bilinear interpolation methods are applied, estimating new pixel values based on surrounding ones while providing an effective balance between computational efficiency and visual quality.
3. **Model Inference:** The specified model is applied to the processed image, generating the following data for each detected target object:
   - Class identifier;
   - Bounding box;
   - Pixel-level segmentation mask;
   - Detection confidence score;

   This inference process runs on dedicated thread pools to prevent blocking during intensive computation. Figure 5.3 serves as a visual support illustrating relevant data structures used up to this point.

Figure 5.3: *Main data structures from the first three steps: 1) Request format; 2) Image frame; 3) Detection result format.*

4. **Mask Analysis and Simplification:** Detections with high confidence (> 0.7 by default) then undergo further geometric analysis:
   a) Contours are extracted from the segmentation mask;
   b) The contour is approximated with the Douglas–Peucker algorithm, obtaining the best-fitting quadrilateral;

   Figure 5.4 illustrates the outcome of each analysis step, highlighting how the representation is progressively simplified and the number of points reduced.

Figure 5.4: *Reference for the mask analysis steps: The segmentation mask (left) identifies every pixel that belongs to the detected object. Step 4a extracts only the outer contour, reducing the pixel amount. In step 4b, the contour is further approximated into a quadrilateral, keeping only the four most relevant corner points.*

   The resulting quadrilateral approximation is, in practice, an enhanced variant of the standard bounding box, as it is *object-oriented* and not constrained to right angles or parallel sides. This approximation captures the minimal geometric information required to unambiguously determine both the center position and rotation angle of *any* target object, regardless of shape, while eliminating the computational overhead associated with processing complete polygonal masks.

5. **Coordinate Normalization:** All corner pixel coordinates are normalized to a $[0, 1]$ range relative to the image width and height, making them independent of original resolution. The origin $(0, 0)$ is defined at the bottom-left corner of the frame, to match common graphics conventions. Object corners are ordered clockwise for consistency.

Normalization plays a crucial role in this pipeline: rather than reporting absolute pixels, coordinates are expressed proportionally (e.g., "a corner lies at 20% of the image width and 10% of its height"), maintaining consistency even after being resized.

After this process, the server returns detection results in a JSON format, as exemplified in Listing 5.2.

```
1 { "detections": [
2     { "class_id": "solar_panel",
3       "corners": [ [0.245, 0.823], [0.687, 0.831],
4                    [0.692, 0.456], [0.241, 0.448] ] },  ...
5     ]
6 }
```

Listing 5.2: Excerpt showing the structure of the server response: a detection entry contains the object's class and ordered corner coordinates.

This structure is an industry standard, optimized to reduce network bandwidth usage and to be easily parsed by any client application.

#### 5.1.2.2 Concurrent Connection Management

The main Python server employs an asynchronous, thread-based architecture to efficiently handle multiple simultaneous connections while maintaining responsive performance.

Each client connection establishes a separate, persistent WebSocket channel, enabling bidirectional communication with minimal latency overhead. When a valid request is received, the `ObjectDetector.detect()` method is executed within a dedicated thread pool, allowing multiple requests to be processed in parallel without blocking the main event loop. In other words, this ensures that one client's detection task does not impact response times for other connected users.

Furthermore, a dictionary is used to track the processing state of each connection. The system ensures that if a client submits new frames while a previous detection is still running, the new request is discarded rather than queued. This design prevents memory buildup and avoids introducing latency, which is particularly critical in interactive applications that demand accurate, real-time responsiveness. By always operating on the most recent data instead of outdated frames, the system ensures that the virtual content remains aligned with the current state of the physical world.

Finally, the server is designed to handle failures while maintaining session stability. Invalid or corrupted frame data trigger error responses sent back to the client, enabling transparent recovery from temporary client-side issues without terminating the connection. For severe failures (such as network interruptions or unexpected disconnections), the server automatically cleans up associated client state and resources, allowing reconnection if desired. This approach ensures continuous responsiveness even under adverse conditions.

#### 5.1.2.3 Network Configuration Optimization

A final challenge concerns accessibility across diverse network environments. Many educational institutions employ restrictive firewall policies and NAT configurations that can block external access to locally hosted servers.

To address this, the implementation leverages the Ngrok framework, which provides persistent hostnames (currently formatted as `[assigned identifier].ngrok-free.app`), ensuring consistent endpoints across server restarts. Through secure tunneling, outbound connections are first directed to Ngrok's infrastructure, which then proxies the traffic back to the local server.

This approach eliminates the need for complex manual network configuration, lowering deployment barriers in educational settings. Additionally, Ngrok provides encrypted transport for image data, addressing potential privacy concerns when handling sensitive content.

### 5.1.3 Key Design Decisions and Analysis

To summarize, several choices shaped the server architecture, balancing performance, accessibility, and educational utility:

1. The server is **input-agnostic**, treating all frames equally regardless of source, which maximizes modularity and reusability;
2. While slightly more computationally expensive when compared to less detailed detectors, the use of **segmentation models** yields more accurate alignment of virtual objects, essential for precise spatial anchoring;
3. The **output normalization** approach guarantees interoperability across different platforms and devices with varying screen resolutions and aspect ratios;
4. The hybrid **asynchronous/threaded design** supports concurrent connections while maintaining responsiveness and preventing resource exhaustion;
5. Although it introduces an external dependency, Ngrok **eliminates network configuration barriers** that would otherwise limit deployment;
6. The emphasis on **ease of model training** reflects the prioritization of enabling the **diversification of educational content**, allowing anyone to easily create these models. This is a fundamental step towards allowing full experience customization.

Together, these decisions create a component that is independent, reusable, and modular, able to easily evolve alongside the larger system or even be repurposed for different detection contexts with minimal rework.

In Chapter 6, the full process of creating an educational module is presented, which includes both the training of a model and a deeper analysis of the expected accuracy and response times that result from this implementation.

## 5.2 Application Design and Scene Management

Moving onto the Unity implementation, one of the first and most critical decisions was to establish a clear organizational structure. To contextualize, as applications grow in complexity, maintenance and debugging become significantly more challenging without proper organization. This foundational design encompasses multiple aspects, including folder hierarchies for assets, data structures, code architecture, and the focus of this section - scene organization and management.

Scenes in Unity serve as separate “containers” that hold all assets, environments, and logic required for distinct parts of an application. This may mean specific areas, levels, or user interface environments. Because scenes define the boundaries between these parts, their structure and management directly impact user navigation, data persistence across different states, and resource management efficiency.

In traditional single-player applications, scene transitions are relatively straightforward: the “old” scene unloads, and a “new” one initializes in its place, fully resetting the application’s

state with new objects and logic. While this approach is sufficient in isolated scenarios, it becomes an increasingly complex challenge in multiplayer and MR applications.

In these environments, transitions must be carefully orchestrated to avoid breaking network synchronization, disconnecting users, or unnecessarily reinitializing hardware features. Given the variety of educational experiences that can be created, functionalities such as marker detection, hand tracking, or spatial mapping may be required only in specific modules. Unnecessarily running the device subsystems that support these functionalities not only wastes computational resources but may also degrade performance or introduce latency. Multiplayer scenarios further increase complexity, as all connected clients must maintain consistent state during transitions.

The scene management strategy was therefore motivated by the following challenges:

- How can the scene architecture be divided to maximize modularity?
- How can it be clear which functionalities are required in each scene?
- How can persistent data be maintained seamlessly across transitions?
- How can scene transitions be optimized, and when should they take place?

The adopted scene architecture is detailed in Subsection 5.2.1. Subsection 5.2.2 then describes the technical implementation details and defines the transition pipeline, while Subsection 5.2.3 summarizes the key decisions that support the proposed design.

### 5.2.1 Tiered Scene Architecture

Two key requirements were immediately apparent during planning. First, the application needed a standalone Main Menu scene to serve as a central hub for navigation and user interaction. Second, each educational module had to be developed as an independent unit, enabling multiple experiences to be worked on in parallel, and allowing them to be later reused or extended later without affecting other components.

Figure 5.5 shows a mock-up of the Main Menu interface.

Figure 5.5: *Mock-up for the main menu interface: the experience selection panel is shown the left, while lobby options are displayed on the right (to be discussed in subsequent sections).*

As can be seen, the left-hand window provides users with an intuitive starting point for accessing different educational experiences. This interface introduces a critical transition requirement: when a user chooses to `start` an experience, the system must identify the appropriate scene and coordinate the transition.

Adding onto this, the demands inherent to multiplayer and MR development also revealed that certain systems must persist across *all* scene transitions. Network connections, device configurations, and core management systems cannot be reinitialized every time the user navigates between the Main Menu and individual learning modules. Doing so would not only increase loading times and computational overhead but could also compromise stability in these complex scenarios.

With all these principles in mind, the scene architecture was structured into three distinct tiers, each with specific responsibilities and lifecycle requirements, refer to Figure 5.6:

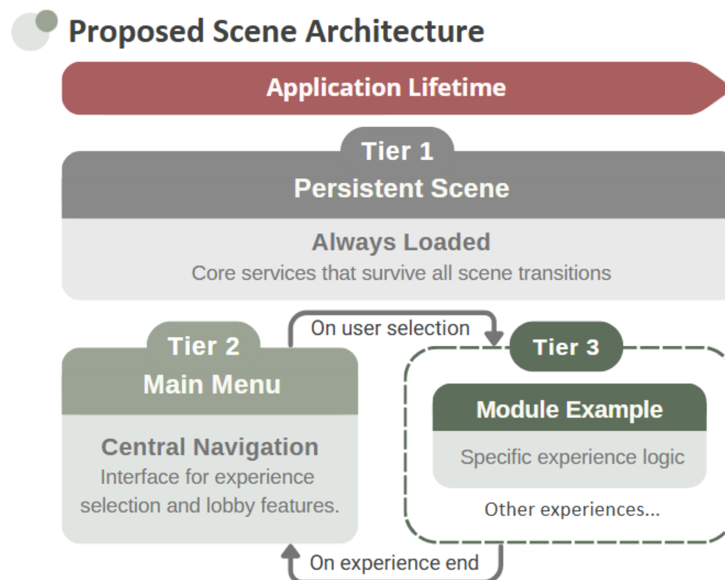



Figure 5.6: *Visual reference for the proposed scene architecture, with the identified layers and transition conditions represented throughout the application lifecycle.*

Tier 1. **Persistent Scene**

Serving as a foundational layer, this scene is loaded only once at application startup and remains active throughout the entire application lifecycle.

It contains all systems that must survive across scene transitions, including critical management components that coordinate application-wide functionalities:

- `NetworkManager`, `LobbyManager`, and `RelayManager`, which handle multiplayer coordination (detailed in Section 5.3).
- `PlayerManager`, responsible for client-side logic (detailed in Section 5.5).
- `SceneLoader`, which coordinates scene transitions.

Maintaining network-related systems in a persistent layer is particularly crucial for multiplayer stability, ensuring network connections, session data, and player-specific

states remain intact during transitions. This prevents repeated disconnections, data loss, and the computational overhead of re-establishing network sessions.

Tier 2. **Main Menu Scene**

Serving as the primary navigation hub, this scene presents the user interface for selecting educational experiences and accessing lobby functionalities (shown in Figure 5.5). It is the first scene users actually see upon launching the application, since the Persistence Scene has no visual component and runs only in the background.

Designed to be lightweight and isolated, this scene contains no simulation logic or heavy computational processes, leading to rapid loading. Because of this, it also functions as a reliable fallback: if an individual experience encounters issues, users can safely return to this scene with minimal disruption.

Tier 3. **Learning Modules (Immersive Experience Scenes)**

Each immersive learning experience is encapsulated within a dedicated scene - for clarity and consistency, this specific type of scene is referred to as a "learning module" throughout this text. Each of these modules is self-contained, managing all assets and logic required for its corresponding experience. Using a standardized interface approach (described in Subsection 5.2.2.1), specific subsystem requirements can be defined and associated, ensuring that only necessary hardware features are initialized.

This modular approach allows experiences to be developed, tested, and deployed independently. Developers can focus on creating engaging learning content without worrying about interfering with core application systems or other modules.

Together, these tiers form a coherent hierarchy: the Persistent Scene provides foundational services and system continuity, the Main Menu Scene handles user interaction, and Learning Modules deliver the educational content. This separation of concerns allows each layer to be optimized for its purpose while maintaining clear interfaces and predictable behavior across the system.

### 5.2.2 Technical Implementation

Having established the tiered architecture, it is now necessary to define how those layers translate into code and how transitions between them are managed. The diagram in Figure 5.7 illustrates the relevant classes that will be further discussed in this subsection.

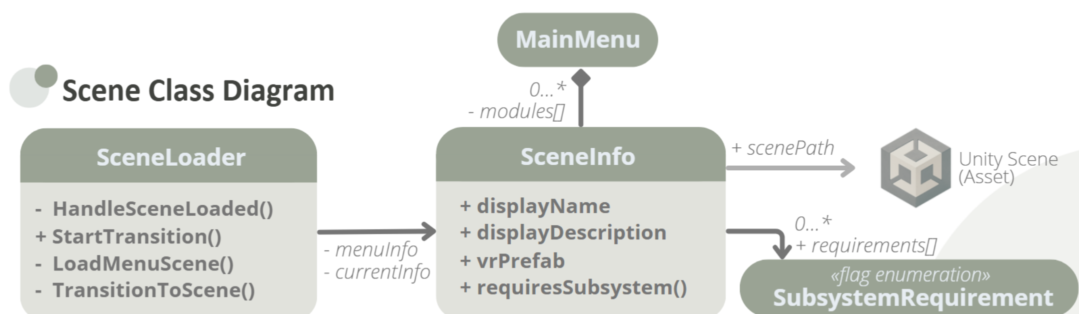



Figure 5.7: *Class diagram focusing on scene-related classes: `SceneInfo` and `SceneLoader`.*

#### 5.2.2.1 `SceneInfo` - Generalizing Scene Information

A transition requirement was introduced alongside the Main Menu: once a user chooses a learning experience, how does the system map the selection to a specific scene, and how to ensure the device's subsystems are prepared to support its requirements? To address this, the application employs a unified representation of scenes through a custom asset type called `SceneInfo`.

`SceneInfo` assets encapsulate all metadata relevant to a particular scene, leveraging Unity's *ScriptableObject* pattern to provide a data-driven approach to scene management. This avoids hardcoded logic and offers a single, centralized source of truth for scene configuration. Each `SceneInfo` object contains:

- **Display Information** — the name and icon shown in the Main Menu.
- **Environment Assets** — a backdrop prefab to be spawned for VR users.
- **Technical Details** — the Unity scene path and the `SubsystemRequirements`.

Building on this last item, the `SubsystemRequirements` solution allows each scene to specify which device features (device subsystems) it depends on. To illustrate, not every learning module needs every subsystem: a simple quiz may run without any special features, while other experiences may require both marker detection and voice tracking. To represent these variations, the application adopts a *flag-based enumeration*.

Each possible subsystem is encoded as a binary flag, making it possible to combine multiple requirements into a single field. An excerpt of the enumeration is shown in Listing 5.3.

```
1 [System.Flags]
2 public enum SubsystemRequirements
3 {
4     None              = 0,
5     MarkerDetection = 1 << 0, // 0001
6     VoiceInput      = 1 << 1, // 0010
7     HandTracking    = 1 << 2, // 0100
8     SpatialMapping  = 1 << 3  // 1000
9 }
```

Listing 5.3: Flag-based definition of subsystem requirements.

With this system, a simple scene may declare `None`, requiring no subsystems, while another module that uses marker detection and hand tracking would combine both flags, producing the value `0101`. During a scene transition, the `DeviceManager` (detailed in Section 5.5) simply checks these flags and toggles the subsystems 'on' or 'off' depending on whether its flag is present. This ensures only the necessary features are running, reducing overhead, avoiding compatibility issues, and keeping scenes lightweight.

Most importantly, this design allows new subsystems to be added in the future by simply extending the enumeration, without having to modify existing scene configurations or transition logic.

SceneInfo assets can be created directly in the Unity editor without writing any code. As shown in Figure 5.8, the flag-based enumeration appears as an intuitive dropdown menu in the inspector, making the setup accessible even for non-programmers and designers.

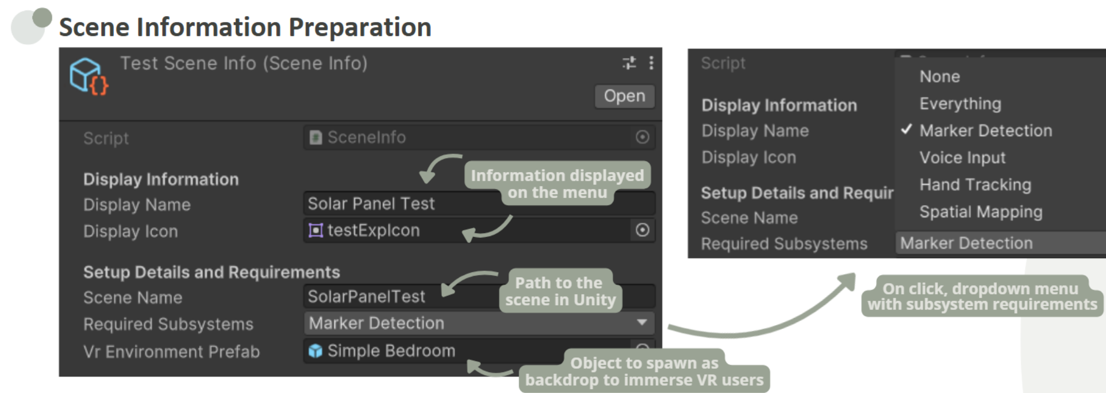



Figure 5.8: *Example of the creation of a `SceneInfo` object via the Unity editor interface.*

At runtime, the script that handles the Main Menu logic loads all SceneInfo assets from their dedicated folder, ensuring that the interface always reflects the current set of available learning modules without requiring manual updates.

#### 5.2.2.2 SceneLoader - Managing Scene Transition Logic

With SceneInfo providing the metadata structure, the responsibility of executing scene transitions falls to the SceneLoader component. This component implements the singleton pattern to ensure a single, globally accessible instance, ensuring that all transitions are coordinated from a central point throughout the entire lifecycle. It is included in the Persistent Scene.

Whenever a transition completes, this class broadcasts an OnSceneLoaded event, enabling other subscribing systems to respond independently. This allows interface handlers to update their elements, VR environments to configure appropriate backdrops, and device subsystems to change state as required by the new scene — all without the SceneLoader needing explicit knowledge of these specific systems, thus avoiding tight coupling.

Scene loading itself is handled asynchronously through coroutines that run parallel to the main thread. This approach prevents the latter from being blocked, keeping the interface responsive and allowing for smooth transitions even in cases where scenes involve significant assets or initialization steps.

Although multiplayer functionalities will only be explored in detail in Section 5.3, one important implication must be anticipated: when several users want to engage in the same learning experience as a group, scene transitions must occur in unison. In other words, users should not be moved to separate copies of the same scene, but rather participate in a single, shared transition that brings all clients into the same synchronized environment while maintaining consistent user states and data.

To address this, `SceneLoader` integrates Unity Netcode's `SceneManager`. In short, when a group host initiates a learning experience, this framework enables the propagation of this change across all other participants, transitioning them onto the shared scene simultaneously. Furthermore, it is also ensured that users joining an ongoing session are automatically placed into the currently active synchronized scene, preserving group consistency.

Bringing all the discussed elements together, the complete transition lifecycle is shown in Figure 5.9, with the process being summarized below.

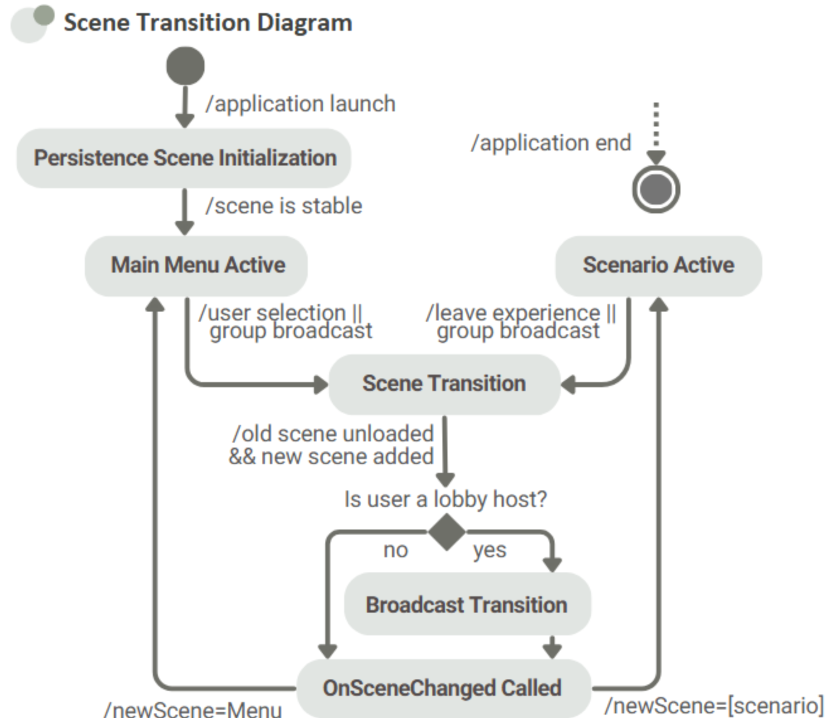



Figure 5.9: *Flowchart depicting the scene transition lifecycle.*

- **Application Launches:** the Persistent Scene initializes first, creating the foundational network infrastructure and initializing the core managers.
- **Menu Scene Loads:** Once the previous layer is stable, the system immediately loads the Main Menu scene additively, providing the user with both navigation and lobby options.
- **User Selection:** When a player requires a scene change (by selecting or leaving a learning experience), the public `SceneLoader.StartTransition()` method is called.
- **Transition Execution:** The transitions process begins in a parallel coroutine:
  - The active scene (whether Main Menu or an individual learning module) unloads, leaving only the Persistent Scene active.
  - If operating in multiplayer, the transition is triggered across all required clients, ensuring synchronized loading.
  - The new scene is added in asynchronously.
  - Upon completion, `SceneLoader` fires the `OnSceneLoaded` event allowing all subscribing systems to perform their respective tasks.

### 5.2.3 Key Design Decisions and Analysis

The main design choices described throughout this section are highlighted below:

1. The usage of a **persistent scene strategy** maintains critical systems across transitions, eliminating network reconnection overhead and hardware reinitialization delays;
2. The **tiered hierarchy** creates a clear separation of concerns, with each layer optimized for its specific purpose while maintaining clear interface contracts;
3. **Flag-based subsystem requirements** enable optimal resource management, ensuring only necessary hardware features are active in each scene;
4. The **data-driven `SceneInfo` approach** eliminates hardcoded logic through a single source of truth, and enables content scalability without core system modifications;
5. **Asynchronous transition pipelines** maintain responsiveness and smooth user experiences during scene changes;
6. Using **event-based notifications** during the transitions also decouples scene management from all other server logic, reducing dependencies;
7. Integration with **Unity Netcode's `SceneManager`** ensures synchronized multiplayer handling using standardized networking protocols.

The adopted scene architecture sought to balance modularity and flexibility while meeting complex multiplayer requirements, creating a stable and adaptable management framework that serves as a foundation for the remaining components.

## 5.3 Multiplayer and Lobby Management

With the foundations established, this section further details the implementation of one of the core project functionalities: the networking layer and inherent multiplayer management.

The implemented system proposes solutions to the following challenges:

- How to maintain client data (e.g. high-scores) between sessions?
- How to manage and synchronize groups of users in the application?
- How to achieve the illusion of shared-space between these users?

Subsection 5.3.1 begins by contextualizing the use of the Unity Gaming Services (UGS) as a response to the presented challenges, as well as highlighting the advantages over alternative networking solutions. Subsection 5.3.2 follows with the detailing of how this technology is applied in practice, with Subsection 5.3.3 summarizing the key points.

### 5.3.1 Unity Gaming Services and the Challenges of Multiplayer

Unity Gaming Services represents Unity Technologies' suite of backend networking solutions, designed to address the complex challenges inherent to multiplayer development.

Given the context of MR applications, these challenges become even more critical due to the demanding requirements of spatial synchronization and real-time collaboration. Consider

the following scenario: a student in Braga, wearing AR glasses, wants to study with a friend in Lisbon who is completely immersed in VR.

If one of these students rotates or moves a virtual object, how to ensure the rest of the group witnesses those changes seamlessly and in real-time? How to achieve perfect synchronization despite the amount of miles and varying network conditions between them? To take things further, what would happen if one of the student's connection faltered for even a second: would the other participants be faced with a frozen, glitched representation that breaks the immersive experience?

Every millisecond of latency, dropped package, or compatibility issue between different devices can shatter the illusion of shared presence that makes collaborative MR so potentially enriching for learning.

Many solutions have emerged to tackle these challenges, each with their own strengths and weaknesses, and it becomes crucial to explain why UGS was picked as the optimal choice, exploring how its particular approach aligns with the project's goals.

#### 5.3.1.1 Networking Solutions Comparison

While the core offer of many of these solutions is similar, each one possesses distinct advantages and limitations. In order to select the optimal solution, special attention was given to the specific requirements of MR environments, prioritizing high-frequency spatial updates and cross-platform compatibility, while also considering pricing fees.

With UGS having already been identified as the chosen networking solution, this subsection first examines alternative approaches, offering a comparative analysis to better understand this decision. The most relevant alternatives include:

- **Photon Networking [64]:** Another solution provided by Unity. It was established well before UGS, making it a popular choice due to its mature documentation and ease of implementation. However, several limitations become apparent.

  While it integrates well as a standalone, the architecture is optimized for older versions of Unity, often suffering from compatibility issues when paired with the platform's most recent tools. This happens both due to simple version issues, and for the lack of support for high-frequency updates and data synchronization - features that MR applications demand.

  Furthermore, its concurrent user-based pricing model becomes prohibitive for scenarios requiring large sessions, such as the typical classroom settings.

- **Mirror Networking [65]:** This option emerges as an open-source alternative that offers detailed customization over the networking implementation, along with no added usage fees. By providing full source code access, vendor lock-is also mitigated.

  However, this flexibility comes at the cost of requiring substantially more networking expertise, while also lacking a built-in cloud infrastructure, needed to store and manage data, that other solutions provide.

For MR applications in specific, and considering the existing time constraints, it is hard to justify all the custom development that would be necessary to implement both spatial synchronization and backend services.

- **Custom Networking:** At the far end of the control spectrum lies the option of building a custom networking solution from scratch. Although the time-related issues found in Photon arise again, this approach should still be analyzed as it offers the freedom to tailor every single aspect of the networking layer to the project's specific requirements while fully avoiding dependencies on third-party services.

  Achieving this requires using technologies like sockets or other low-level protocols to implement custom messaging protocols between data servers and users, raising substantial development overhead. The development team would also become responsible for security updates, platform compatibility issues, and NAT traversal.

  For most projects - this one included - such extensive level of control would be excessive and even counterproductive. However, exploring this option further reinforces the importance of leveraging a managed solution, as these handle all the mentioned backend issues automatically.

To summarize, Table 5.1 compares the advantages and disadvantages of each option.

Table 5.1: Comparison of popular networking frameworks and solutions.

| Solution | Advantages | Disadvantages |
|---|---|---|
| **Photon Networking** | • Easy prototyping and set up.<br>• Strong documentation and wide adoption.<br>• Mature ecosystem. | • Pricing based on concurrent users.<br>• Not optimized for frequent updates.<br>• Limited integration with recent Unity tools. |
| **Mirror Networking** | • Open-source and customizable.<br>• No usage fees.<br>• Strong community support. | • Requires deeper networking knowledge.<br>• No built-in backend.<br>• Spatial synchronization must be manually set up. |
| **Custom Networking** | • Full architectural control.<br>• Can be optimized for specific MR needs.<br>• No third-party dependencies. | • Requires networking expertise.<br>• Security support must be manually handled.<br>• Exceedingly high development and maintenance overhead. |

Having established these alternatives, Unity Gaming Services (UGS) emerges as an optimal solution because it offers a balanced "middle ground", providing just enough control to tailor networking to the project's most pressing needs, while offloading the most demanding backend tasks through managed services that reduce development overhead.

Three key factors make UGS a particularly well-suited solution. First, since Unity is the primary development platform, using a toolkit designed specifically for this ecosystem creates an optimized workflow. Instead of having to manage separate systems prone to compatibility issues, these services integrate seamlessly with Unity's latest frameworks.

Second, the pricing model aligns with the project's constraints and growth trajectory. It offers a free tier suited for early development and testing, while providing clear scalability paths through education-specific pricing tiers. This model removes financial barriers in the initial phases and prevents the need for major architectural changes as the project grows.

A final decisive factor lies in its cross-platform support. The project's requirements includes supporting various environments and operative systems, including Android, iOS, or Windows devices. By handling compatibility at the networking layer, UGS eliminates the need for custom adaptation work, reducing development time and maintenance complexity.

Table 5.2 completes the networking solution analysis with UGS's profile.

Table 5.2: Unity Gaming Services (UGS): Advantages and limitations analysis.

| Solution | Advantages | Disadvantages |
|---|---|---|
| **UGS** | • Seamless Unity integration.<br>• Education-friendly pricing model.<br>• Built-in cross-platform support.<br>• Managed backend services. | • Relatively new, with frequent documentation changes.<br>• Less customization compared to open-source alternatives. |

Each individual UGS service addresses distinct aspects of the multiplayer foundation, including user management and spatial synchronization to enable real-time collaboration across varying network conditions. The following subsections outline how these components are leveraged to meet the specific challenges of collaborative MR environments.

### 5.3.2 Technical Implementation

Classes relevant to multiplayer functionalities are illustrated in Figure 5.10.

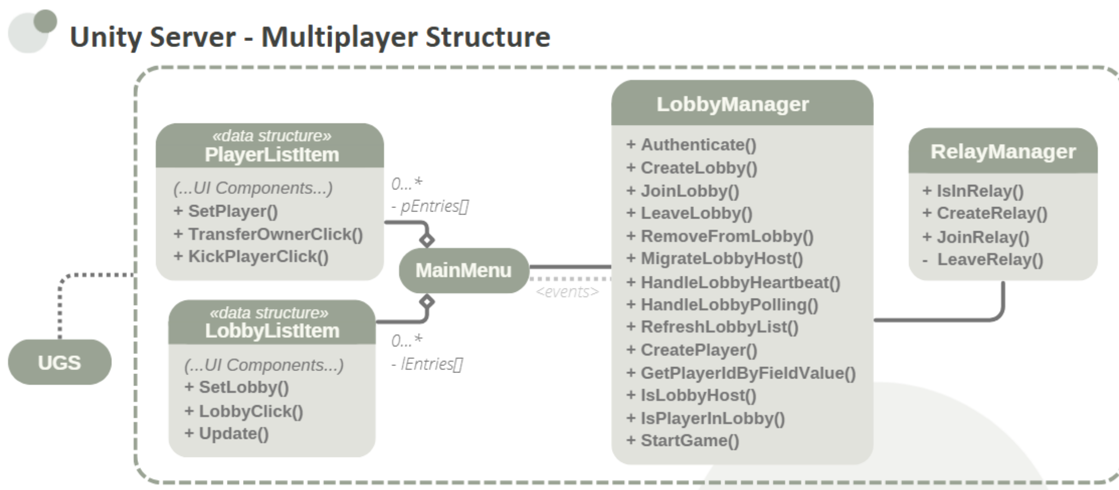



Figure 5.10: *Class diagram depicting the core components of the networking implementation.*

#### 5.3.2.1 User Authentication and Session Management

Before any multiplayer functionality can take place, players must be authenticated into the Unity Gaming Services. Authentication serves as the foundational layer that allows secure, coordinated experiences by assigning distinct identities to each connected client within the system.

This step is essential to ensure a secure and coherent networking experience. Multiplayer systems rely on the assumption that every participant is uniquely recognized by the server. Without this, operations such as player tracking, lobby participation, or the synchronization of game states across devices could not be securely or reliably maintained. For example, when two users join the same lobby, the backend must be able to differentiate their actions, or associate them with persistent data to track their learning progress between sessions. Put simply, two players can not be mistakenly treated as the same entity.

Considering the project's early stage of development, the implementation presently employs anonymous authentication. This allows users to receive a unique identifier without requiring account creation: when the application is launched for the first time on a given device, Unity generates a unique identifier and stores it as a local token on the device. Later launches simply fetch this token to maintain the same identity across sessions.

While this means that the identity is bound to a single device (i.e. if the user changes equipment, the server will treat them as a different identity), this choice reflects the project's present focus: developing the core multiplayer interactions and implementing the educational modules.

Once features that benefit from cross-device persistence - such as personalized learning progress or achievements - are implemented, focus may shift to designing a login interface. This initial mechanism can then be replaced with more robust providers, such as using email/password or third-party logins. UGS makes this transition straightforward, with the change not impacting the way the rest of the multiplayer experience works.

The authentication process takes place automatically upon application launch, being one of the background tasks ensured by the use of the Persistent Scene (refer back to Section 5.2). Listing 5.4 illustrates the implementation, which follows a simple three-step sequence:

```
1 public async void Authenticate() {
2     await UnityServices.InitializeAsync();
3     AuthenticationService.Instance.SignedIn += async () => {
4         RefreshLobbyList();
5     };
6     await AuthenticationService.Instance.SignInAnonymouslyAsync();
7 }
```

Listing 5.4: Method called on start up, allowing authentication into the UGS.

1. **Service Initialization:** `UnityServices.InitializeAsync()` establishes the connection to Unity's backend and prepares the authentication service. This is done asynchronously, allowing other background startup actives to take place in simultaneously.
2. **Event Registration:** The `SignedIn` event handler is registered before the authentication begins. This allows systems to declare any operations that should be triggered immediately after this process succeeds.
3. **Sign-In:** `SignInAnonymouslyAsync()` requests an identity from the backend. Using the "anonymous" method, Unity automatically handles the fetching (or otherwise creation) of any existing tokens at this stage, obtaining the associated user data.

With the current implementation, the only operation called upon successful authentication is the fetching of available lobbies from the backend. This ensures that by the time the player accesses the menu interface (i.e. when the Main Menu scene is added), the relevant lobby data is ready for display.

Lobby management will be further discussed throughout the next subsections.

#### 5.3.2.2 Lobbies and Relay System Overview

Once the user has an identity within the UGS, they can access the lobby and relay systems. These two components work together to support the complete multiplayer experience, each serving distinct but complementary roles in the process.

To better understand the relationship between these systems, the example of organizing a physical group study session can be considered. The lobby acts like a classroom space or meeting point: students can gather before a session begins, checking in on who else is attending and coordinating what they want to study until everyone arrives.

By contrast, the relay system operates behind the scenes. It is the invisible infrastructure and conditions the room provides to enable the work: the network connections, data pathways, and communication channels that allow real-time collaboration to function seamlessly. Just as students don't need to understand the building's network infrastructure to benefit from it, players are unaware of the relay's operation as it runs automatically in the background.

#### 5.3.2.3 Lobby System for Session Coordination

The `LobbyManager` class was developed to encapsulate all lobby-related functionalities, serving as the main point of interaction between the application and the UGS backend. Its methods are invoked by the `MainMenu` class in response to user interface actions, such as refreshing the list of available sessions or creating a new one.

In order to understand the implementation architecture, it is important to first contextualize how lobbies work within the UGS ecosystem. Conceptually, these are persistent data structures hosted on Unity's cloud to represent a group of users. They provide two primary

capabilities: tracking which players are currently present, and storing shared information about the session that all clients can access. These dimensions can be understood as two complementary data categories.

The first one encompasses **Lobby Metadata**, and serves as a general description of the session characteristics. It is stored in the form of key–value pairs (e.g. `num_player:4`). By default, Unity already provides some built-in keys (such as the maximum number of players or host identification), while others can be defined by the developer to suit the application's needs.

In this project, two additional keys were introduced. The `RelayCode`, visible only to members of the lobby, is used as a link to the relay service once the session begins, and will be further detailed in the upcoming subsection; as for the `LobbyState`, it tracks the current session phase with values of either "Waiting" or "InGame". This public status is displayed in the interface, allowing users to specifically look for lobbies that aren't already exploring an educational module, if they prefer to not join mid-session.

The second data category refers to **Player Data**. Different applications have varying requirements for how participants should be represented within a lobby. Like lobby metadata, this mechanism enables customized key-value pairs to represent tailored user profiles.

The current implementation stores two attributes: the `PlayerName` to provide identification within the interface, and a `NetworkClientID` which maintains the unique identifier assigned during authentication, necessary for backend operations that target a specific player.

With this context in mind, the implementation workflow becomes clearer. When the Main Menu interface was previously presented in Section 5.2, the description focused mostly on the educational module panel. Figure 5.11 now highlights the lobby panel and emphasizes its main components.

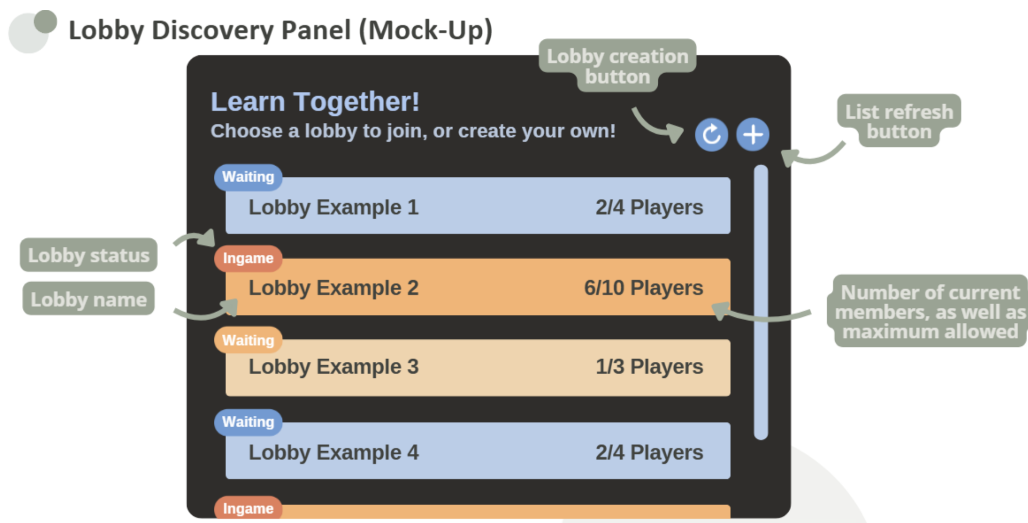



Figure 5.11: *Detailing of the right-hand panel of the main menu (previously shown in Figure 5.5), which enables lobby browsing and interaction.*

The `LobbyManager.RefreshLobbyList()` method is called when this menu is launched or whenever the player presses the refresh button, sending a request to the backend to retrieve available lobby metadata. For optimization, the results are filtered to display only those with available slots, with the most recent entries being shown first.

Each retrieved lobby is visually represented on screen using `LobbyListItemUI` objects, which translate backend data structures interactive elements in the interface (see Figure 5.10). This approach separates data management and presentation logic while ensuring consistent visual representation.

Beyond the listing panel, the interface provides three primary interaction points for lobby discovery and creation:

- **Lobby Creation:** Pressing the '+' button triggers `LobbyManager.CreateLobby()`, which requests the creation of a new lobby entry in the UGS cloud with default metadata values. The requesting user is automatically assigned as the lobby host, which grants additional privileges that will soon be detailed. In practice, a lobby can be created by anyone - for example, a teacher may open one for a classroom session, or a student may start one to study with another friend at home.
- **Refresh Lobby:** To prevent overwhelming the backend with excessive requests, lobby lists are updated through manual user action rather than continuous polling. The refresh button provides users control over when to retrieve updated information.
- **Lobby Entry:** Individual lobby entries, as represented by the `LobbyListItemUI` components, respond to user selection by invoking `LobbyManager.JoinLobby()` with the selected lobby's specific parameters. If this succeeds, the user is added to the lobby's participant list as a standard member.

When a user joins a lobby, the interface updates. As can be seen in Figure 5.12, the panel previously used for lobby discovery is replaced with a list of participants. This transition provides users with the necessary tools to coordinate their collaborative experience.

Figure 5.12: *Updated panel after joining a lobby: the illustrated scenario is specific to the host, with coordination privileges being shown.*

The player panel works similarly to the lobby list: Player Data structures are retrieved from the cloud, and `PlayersListItemUI` objects again serve as the mapping mechanism between backend data and visual representation. The lobby host is displayed first, followed by other participants in order of entry.

This time, the player panel implements automatic updates rather than manual refresh. This choice stems from the different nature of the information being provided. While lobby availability is something the user needs to actively search for, it makes sense to only fetch updates on demand to avoid unnecessary traffic. In contrast, the composition of a lobby may change at any moment, outside the player's control (for example, when someone else joins, leaves, or updates their profile data).

To ensure the participant information is always up-to-date, the `LobbyManager` registers event handlers that listen for UGS notifications. These are triggered by any relevant lobby changes, allowing the interface to react immediately and maintain a consistent, synchronized view across all connected clients.

In the new panel, different actions are made available:

- **Leave Lobby:** Users may leave the lobby at any time by pressing the '←' button, which invokes `LobbyManager.LeaveLobby()`. If the host exits, lobby ownership is automatically transferred to the longest-standing participant.
- **Host Privileges:** Hosts are granted additional controls that appear as supplementary buttons on each player entry. These include removing players from the lobby (`LobbyManager.RemoveFromLobby()`) or transferring lobby ownership to another participant - in other words, changing the host (`LobbyManager.MigrateLobbyHost()`).

Only the host may initiate an educational module, with the 'start' button being disabled for regular participants. This design prevents issues that could arise from multiple simultaneous initiation attempts, and allows for a more coordinated launch. This is the action that triggers the `RelayManager` to begin handling real-time networking responsibilities, as detailed in the next subsection.

Before this, as a final note regarding the lobby structures, these currently store only essential information required for basic multiplayer coordination, which reflects the project's early stage of development. However, the structures are designed to be extensible and can be easily adapted to support further customization in future iterations.

For instance, once more detailed authentication and user profile systems are implemented, player data could include customized usernames (as these are currently randomly generated for testing) or profile pictures. Lobby-level metadata may allow setting module preferences or specifying geographic regions to optimize matchmaking and improve the user experience.

#### 5.3.2.4 Relay System for Experience Synchronization

While the lobby system allows creating the collaboration groups, it does not enable real-time interaction by itself. The Relay infrastructure is still necessary to enable users to actually interact and see each other's contributions in real time.

In a more technical manner, it establishes peer-to-peer communication between participants, while automatically handling common networking issues such as NAT traversal or firewall restrictions. Without this component, players might struggle to connect if they are behind routers with restrictive inbound rules - a frequent scenario in educational contexts. To achieve this, Unity provides managed intermediate hubs (known as relay servers) to where messages are first forwarded, allowing all clients to exchange data even when direct connection is impossible.

The developed `RelayManager` class encapsulates all responsibilities related to this process, exposing only two methods for the rest of the application: `CreateRelay()` and `JoinRelay()`. These reflect the two "phases" into which the relay workflow can be divided - allocation creation and client connection, respectively.

Figure 5.13 illustrates the interactions between lobby participants (host or member) and the backend server, which occur entirely asynchronously.

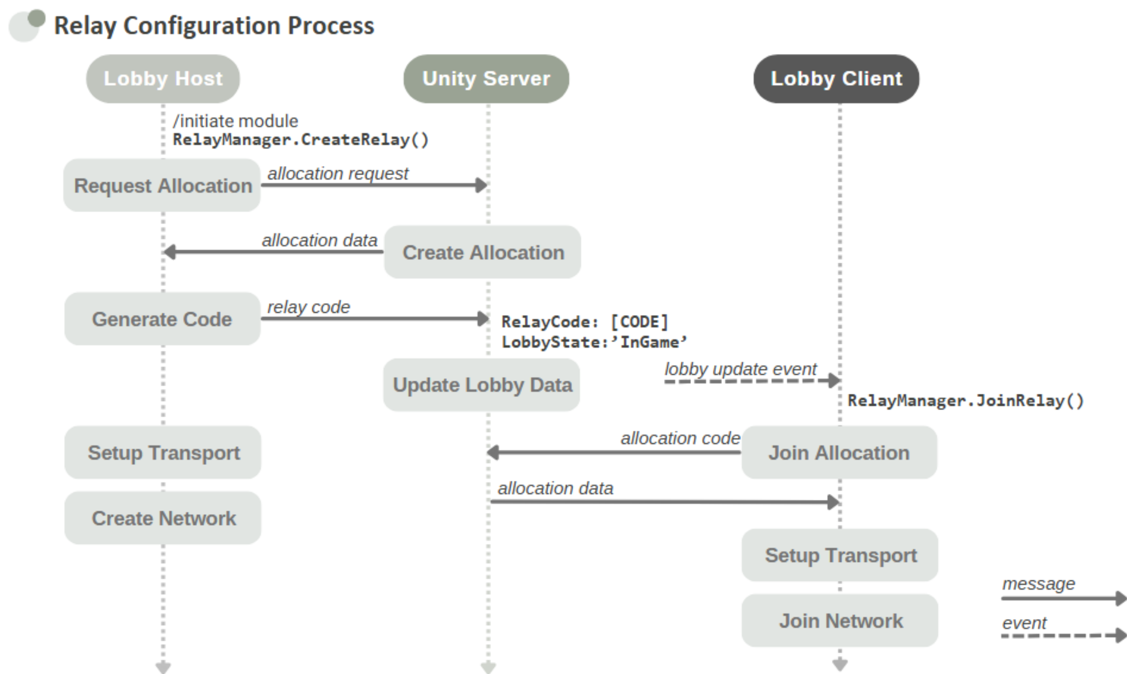



Figure 5.13: *Interaction diagram depicting the message flow between lobby participants and Unity's backend services during relay setup.*

When the lobby host initiates a module, the system must obtain a dedicated communication space in Unity's backend - this is known as an allocation. This is achieved by calling the `RelayManager.CreateRelay()` method, which performs the following steps:

1. **Allocation Request:** The system sends a request specifying the expected participant count. Unity's backend responds with connection endpoints and necessary authentication tokens.
2. **Join Code Generation:** A short, human-readable code is created to represent the allocation. This functions like a room key, allowing participants to connect without exchanging technical details such as IP addresses.
   This code is automatically stored in the lobby metadata using the `RelayCode` mentioned previously, making it immediately accessible to all lobby participants.
3. **Transport Configuration:** To ensure all traffic is routed through Unity's relay infrastructure, the Unity Transport component must be configured with the relay server data. This achieved by calling a built-in UGS method, `SetRelayServerData()`.
4. **Network Creation:** Finally, the `NetworkManager` is launched in host mode. This component, further described in Section 5.4, orchestrates object synchronization for connected clients.

Since the `LobbyManager` continuously listens for updates to lobby metadata, all connected participants automatically receive the updated information upon relay code addition. When this event occurs, `RelayManager.JoinRelay()` is invoked for each participant, following a parallel but simplified workflow to connect to the active session:

1. **Join Allocation:** Using the join code from the lobby metadata, the client requests access to the respective allocation. Unity's backend validates the code and returns the necessary connection parameters.
2. **Transport Configuration:** The same transport setup must be ensured, configuring the Unity Transport with the same relay server details as the host. This ensuring that all traffic flows through the same shared infrastructure.
3. **Client Initialization:** The `NetworkManager` is started in client mode, enabling the participant to enter and synchronize with the active session.

It should be noted that the participant limit specified in the allocation request matches the maximum number of players defined for the lobby at creation time, even if not all slots are occupied. If at any point this limit needs to be increased, the system seamlessly requests a new allocation with a higher capacity and generates a corresponding join code. This enables all clients to automatically reconnect without any impact to the experience.

Regardless of the followed path, once all participants complete this process they share a synchronized communication channel established through Unity's relay servers. Combined with the scene synchronization mechanism described in Section 5.2, every participant is also ensured to be placed within a shared virtual space.

Put simply, it means that at this stage the "illusion of a shared space" is fully prepared: the network infrastructure and the scene alignment set the stage upon which collaboration can then unfold. The only missing element is populating this environment with interactive objects and behaviors, ensuring that their movement and state are propagated consistently across all connected devices in real time.

### 5.3.3 Key Design Decisions and Analysis

Before advancing to the implementation of object interaction and synchronization, it is important to step back and review the proposed multiplayer architecture design.

To begin, the decision to clearly separate lobby coordination from relay synchronization offers a clean division of responsibilities. This approach continues to focus on allowing each service to be extended independently: for example, the relay system could be altered to use different connection protocols without affecting the lobby coordination in any way.

The selection of UGS further reinforces this modularity. It allowed the project to offload the low-level networking tasks (such as NAT traversal and session allocation) to a managed infrastructure, freeing the design to focus on collaboration interaction rather than how packets move behind the scenes. It also ensures the reliability required in an educational environment.

Beyond this, several specific design details can be highlighted:

1. At this stage, **lobby structures hold only the minimal information needed for basic coordination**. This reduces overhead in early development while keeping the design easily extendable for future work.
2. While seemingly minimal, the **differentiated polling approaches** between lobby discovery and participant monitoring prevent overwhelming the backend with unnecessary requests, maintaining real-time updates only where needed.
3. The **host-centered control** prevents race conditions or duplicate relay allocations that could arise from simultaneous initiation attempts. This ensures a clean startup and a single point of coordination.
4. The **mechanisms for transferring host ownership**, either voluntarily or automatically in case of error, ensure session continuity and and prevent sudden interruptions to collaborative workflows if the host disconnects.
5. Even at this early stage, the **interface was designed to be clear and intuitive**. This proves valuable for testing and iteration phases where quick onboarding allows focus to remain on evaluating the experience rather than navigating complex menus.

Lastly, network operations are inherently unpredictable, making it imperative that the system focuses on providing error handling. In this implementation, a particularly critical point refers to relay creation and connection, where multiple intervening services can cause unexpected failures.

Should any errors arise when creating an allocation, the system logs relevant information without crashing the application. This informs user of potential issues on their side and gives them the option to try again when these are resolved.

As for relay connection failures, the system follows a similar principle: errors are logged but do not block further attempts. The relay key persists in the lobby metadata, meaning the user can continue to retry joining if initial attempts fail. This ensures momentary network instability or service unavailability does not allow any players to be “left behind”.

## 5.4 Object Interaction and Networking

With the networking infrastructure established and participants connected within a synchronized virtual environment, the final step in creating the illusion of a shared space is enabling users to interact with objects in the scene. This is the most visible and interactive aspect of the multiplayer experience: when a user picks up or manipulates an object, all other participants must observe these changes in real time, regardless of their physical location or device type.

While single-user experiences are supported, from a technical standpoint these essentially operate as a "single-participant collaboration" using the same underlying mechanisms. Examining the system from a multiplayer perspective, as is done throughout this section, helps make the synchronization challenges and design choices clearer.

Object synchronization is particularly crucial in collaborative MR environments to sustain the sense of presence, yet it is precisely the unique characteristics of these applications that introduce additional complexity. Some of the digital objects are not purely virtual: their position and behavior are anchored to real-world markers and reference points. While marker detection has been addressed in Section 5.1, their placement and synchronized interactions still require further handling.

Furthermore, a representation challenge arises from the differing ways AR and VR users perceive their environments. AR users operate in a hybrid space where virtual and physical elements coexists, making their experience depend heavily on spatial alignment between virtual content and real-world markers. By contrast, VR users exist in fully immersive environments with no visible physical markers, necessitating alternative virtual representations to ensure they experience the same interactions.

Simply put: how can the system deliver representations tailored to each device type without compromising either experience? The solution must ensure AR users can take full advantage of precise spatial mapping, while allowing VR users to fully interact with marker-dependent objects despite the absence of the real-world anchors.

Finally, not all objects are marker-based. Many are standard virtual elements that can be freely manipulated in the shared space: they are not inherently connected to a physical marker. The architecture must clearly differentiate these object types and optimize their interaction rules.

In short, the implementation must address the following core questions:

- How can different experiences associate specific virtual objects to detected markers?
- How can object positions and states be synchronized across users in real time?
- How should to handle the distinction between standard manipulable objects and marker-anchored objects?

- How can users see a consistent, optimized representation of objects suited to their device capabilities?

This section details the strategies and implementation techniques used to address these challenges, leveraging the final Unity Gaming Services (UGS) service employed in this project: Netcode for Game Objects. The description begins by detailing this technology and how it ties into the object lifecycle management and synchronization mechanisms, through Section 5.4.1.1, followed by 5.4.1.2 which details how the distinction between object types is handled. Section 5.4.2 concludes with a critical analysis of the implemented solution.

### 5.4.1 Technical Implementation

To support the implementation discussion, Figure 5.14 provides a class diagram illustrating the most relevant components.

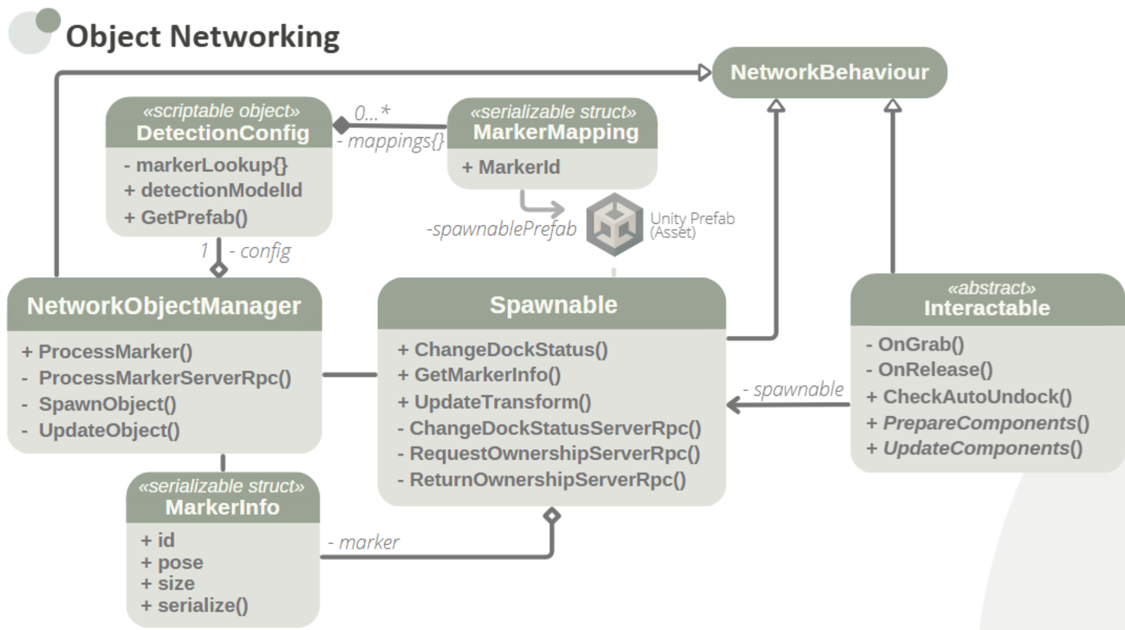



Figure 5.14: *Diagram depicting classes relevant to network object management.*

Before delving into the more complex architectural components, some foundational classes can be briefly contextualized. One of these is the `MarkerInfo` data class.

Recalling the object detection pipeline from Section 5.1, the Python server returns a 2D representation of each detected marker in a frame, along with its class label. However, Unity operates in 3D space, so this information is insufficient for accurate object placement.

While the specific conversion mechanisms will be detailed in Section 5.5 when examining the complete client-side workflow, the key point is that clients are capable of performing this dimensional transformation, transforming the 2D corner points into the corresponding 3D positions within the shared scene space.

Thus, the `MarkerInfo` provides a standardized 3D representation of a marker that is shared between the client and server. This structure stores the marker's 3D pose (encompassing its center position and rotation), dimension, and identifier (matching the class returned by the detection model). Implemented to be serializable, it is optimized for network transmission between client and server instances.

As emphasized throughout this report, different learning modules may utilize different types of virtual objects depending on the educational context: an immersive experience about astronomy might display virtual celestial bodies upon detecting a physical model of the solar system, while a virtual laboratory could trigger experiments based on the recognition of certain physical lab equipment. This context explains the role of the `MarkerMapping` and `DetectionConfiguration` classes.

The `MarkerMapping` class is a simple data structure that associates a marker identifier (the class label returned by the detector) with a specific Unity prefab. It essentially encodes a rule: "when a marker with this label is detected, this object should be associated".

`DetectionConfiguration` acts as a container for multiple `MarkerMapping` rules, essentially acting like a ruleset for a specific module context. Through the public `GetPrefab()` method, it provides a centralized lookup mechanism to efficiently retrieve the appropriate prefab given a marker label. It also specifies the identification of the detection model that recognizes the target markers - this value is fetched when building a detection request.

Implemented as a `ScriptableObject`, `DetectionConfiguration` allows designers and non-programmers to create, edit, and manage these association rules directly in the Unity Inspector, making the process intuitive and accessible (see Figure 5.15).

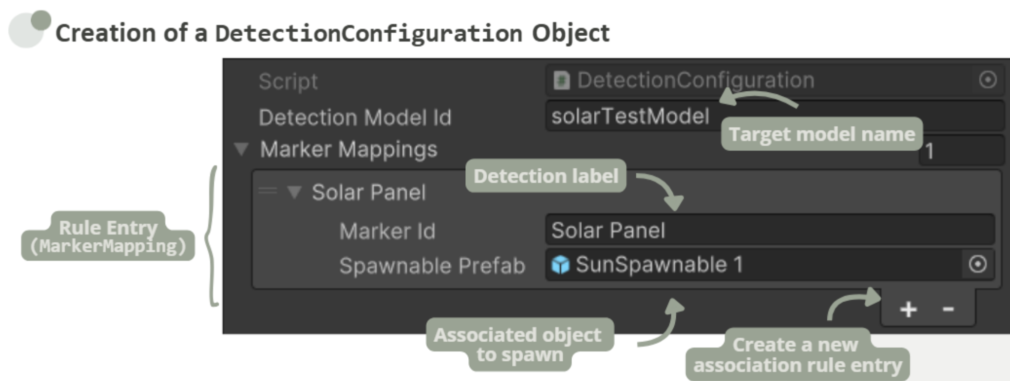



Figure 5.15: *Example of creating an object association ruleset in the Unity inspector: for each mapping, the marker identification/label can be typed out, with the prefab being drag-and-dropped from Unity's asset list.*

Each educational module scene (see Section 5.2) can reference a specific `DetectionConfiguration`, fundamentally defining its ruleset for object association. This scene-based approach provides two key advantages: efficient marker-to-object lookup during runtime, and error prevention by enabling potentially stray detections unsupported within the current scene's configuration to be ignored, thereby avoiding runtime exceptions.

#### 5.4.1.1 Network Object Lifecycle Management

Excluding the classes discussed up to this point, three others remain - all of which inherit from `NetworkBehaviour`. Part of the Unity's Netcode for GameObjects framework, this base class enables inheritors to participate in the networking layer: they can synchronize states across server and clients, respond to network events, and exchange information through specialized methods known as Remote Procedure Calls (RPCs).

To understand what this means in practice, consider how traditional Unity objects only exist on the individual device that instantiated them. When a `NetworkBehaviour` is instantiated, however, it becomes a network identity (it is "network coded"): its lifecycle and state are no longer local to a single client, but are instead coordinated by the server and consistently replicated across all connected participants.

Together with the relay configuration described in Section 5.3, this ensures that object creation, updates, and destruction are propagated in real time to all users in the allocation. In other words, when a user interacts with an object, every participant observes the same modifications simultaneously. This also extends to late joiners: when a new participant connects, they automatically receive the full synchronized state of the scene, including all objects already present.

Netcode for GameObjects further enhances synchronization through the use of Remote Procedure Calls. Unlike regular method calls that only execute in the local machine, a RPC method allows its execution to be triggered on a remote participant. In practice, a class can declare methods as either `[ServerRpc]` or `[ClientRpc]`.

Although these methods are technically defined on both sides, they behave like private mailboxes: When a `[ServerRpc]` is invoked on a client instance, the "letter" is delivered only to the server, which alone executes the method. Conversely, when the server invokes a `[ClientRpc]`, the "letter" is delivered to all connected clients, allowing the server to issue a synchronized action that every participant executes simultaneously.

When the relay preparation process was explored (see Figure 5.13), the last step involved the session lobby host initiating a `NetworkManager` - they act as both a server and a client, making the responsibilities of executing `[ServerRpc]` methods fall upon their side. If this host disconnects, the system transfers this responsibility to another participant, ensuring that no data is lost and the session continues without interruption.

Focusing again on the classes implemented for this project, the `NetworkObjectManager` orchestrates the lifecycle of marker-associated objects throughout a learning session, handling steps from their creation to synchronization across clients. Any scenes pertaining to learning modules that utilize marker interaction include one of these managers as a central component.

Each `NetworkObjectManager` instance maintains two key attributes: a reference to a

`DetectionConfiguration`, defining the module's ruleset for marker-to-object mapping, and a dictionary that tracks active objects in the scene, keyed by the marker label that triggered their creation. This registry prevents duplicate spawns and facilitates object updates.

To make this more clear, Figure 5.16 illustrates the manager's workflow.

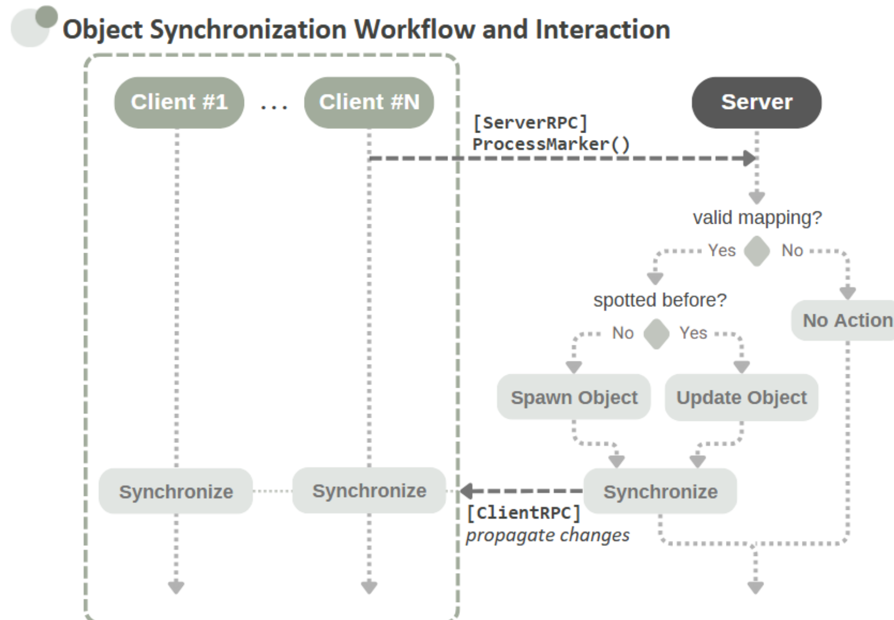



Figure 5.16: *Diagram illustrating the RPC calls between server and clients, along with the operational pipeline that manages object creation, updates, and synchronization across the network.*

The `NetworkObjectManager` exposes a server-authoritative RPC method, `ProcessMarkerServerRpc()`, which handles incoming marker detections from clients. Server authority is critical here: this guarantees that only one entity processes object spawning and updates, ensuring consistency in the dictionary of active objects. If multiple clients detected the same marker and attempted to update this list independently, race conditions would arise, leading to duplicate spawns or mismatched states across devices.

The pipeline unfolds as follows:

- **Detection Notification:** A client notifies the server of a marker detection through a `[ServerRPC]`, sending a `MarkerInfo` object containing the marker's pose and size in Unity's 3D coordinate space.
- **Validation and Lookup:** The server verifies whether the detected marker has an associated prefab in the current scene's `DetectionConfiguration`. If no association exists, the detection is ignored to prevent errors.
- **Detection Handling:** The manager consults its internal dictionary to determine whether the marker has already been registered within the session. The process then branches:

  - **Instantiation:** If the marker is new, `SpawnObject()` is called. The manager retrieves the associated prefab and instantiates it at the marker's 3D position, matching its rotation and scale as well. The instance is registered as a networked object, making it visible to all connected clients. The label-object pair is added to the tracking dictionary.
  - **Updating:** If the marker has already been tracked, `UpdateObject()` is called. The respective object's position, rotation, and scale are updated to match the latest marker information. The changes are automatically propagated in real time, keeping all clients synchronized.

This design balances responsibilities: clients focus on lightweight detection and notification, while only the host handles authoritative processing and global synchronization. Through the asynchronous nature of RPCs and control centralization, the system ensures smooth performance, avoids inconsistencies, and maintains a coherent environment across all connected devices.

#### 5.4.1.2 `Spawnable` vs. `Interactable` Distinction

The final remaining question is exactly *what* objects are spawned in the scene, and how they function within the MR environment. The architecture must simultaneously address the representation challenge posed at the start of this section while optimizing interaction logic across device types.

To meet these requirements, this project proposes a two-tier object architecture that aims to separate spatial anchoring from interaction logic. These two tiers are represented by the `Spawnable` and `Interactable` classes, serving distinct but complementary roles.

To understand this architecture, consider the analogy of a tray holding several items. When the tray is moved, all items resting on it move together while maintaining their relative positions - the items are spatially anchored to the tray. Individual items, however, can be picked up and manipulated independently; once removed, they no longer follow the tray's motion - they become unanchored.

Shifting to the context of marker-based mixed reality, this analogy translates to how virtual objects anchor to physical markers. For instance, when a physical circuit board is detected on a table, a learning module might respond by overlaying a virtual visualization of current flow. As the board is moved, the anchored virtual content remains aligned. However, if the experience allows users to extract individual virtual components (such as electrons in the flow), those elements must be able to detach from the marker and behave independently. The physical marker serves as the reference point, while the virtual objects handle their own interaction logic.

Mapping this distinction to the system architecture:

- `Spawnable` Objects: These are the objects instantiated and handled by the `NetworkObjectManager`. They represent the virtual conceptualization of the physical

marker, implementing the "tray" concept by remaining aligned with the marker's pose and dimensions.

This design provides a straightforward solution to the representation challenge outlined earlier. For AR users, the `Spawnable` object remains invisible within the environment: as the physical marker is already present, they experience seamless alignment without visual duplication. Conversely, for VR users who can not see the physical marker, the `Spawnable` renders a virtual representation, providing essential spatial context for understanding the anchor points that guide the experience's interactions.

- `Interactable` Objects: These represent the manipulable virtual elements: the "items on the tray". They are responsible for processing user input actions and enabling direct manipulation of virtual content.

  Implemented as an abstract class, it defines a set of standard responses to user input (e.g. how to respond to grabs or releases). Other objects can then extend these behaviours to fit their own interaction logic. For example, virtual objects representing atoms may trigger a bonding animation when released near another, whereas a virtual circuit component might snap into a slot on a board.

  As for *how* the input actions are recognized, the OpenXR framework described on Section 3 finds its first use: it provides a series of methods like `OnGrab()` or `OnRelease()` that are automatically called upon user interaction regardless of whether the input comes from hand tracking in AR, controller buttons in VR, or any other variant. This approach allows the generalization of common actions without requiring any device-specific knowledge.

  Unlike `Spawnable` objects, these are always visible to users, providing the visual and tactile elements that enable meaningful interaction within the virtual environment.

Figure 5.17 provides an example of the respective views AR and VR users would have within the same experience.

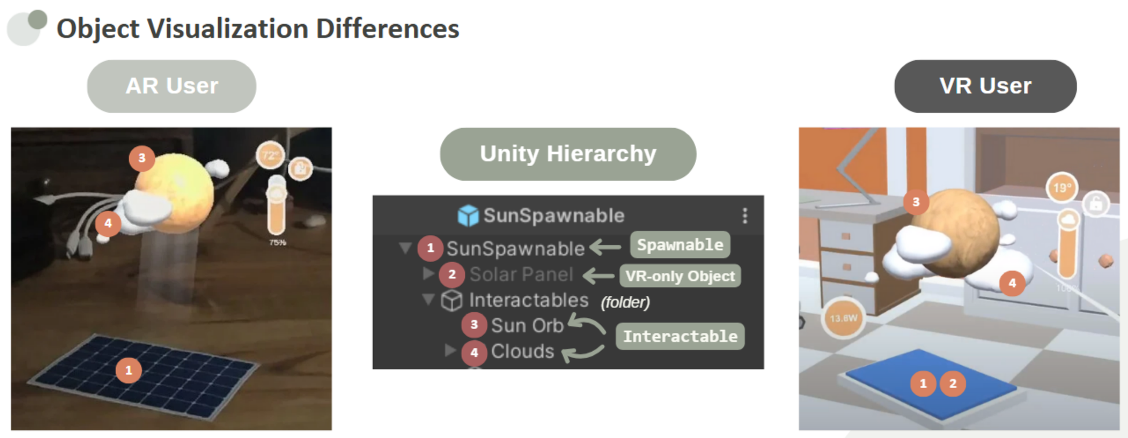



Figure 5.17: *Example of how object visualization varies across device types: the virtual solar panel is made visible only to VR users, conceptually replacing the physical printed marker. The sun and the clouds are* `Interactable` *objects, enabling manipulation.*

The figure also demonstrates how these objects are organized within the scene: as can be seen, when a `Spawnable` is instantiated, all `Interactable` components it contains are nested as child objects. This hierarchy directly implements the tray analogy: when the parent `Spawnable` moves (following marker updates), all child `Interactable` objects move accordingly.

For flexibility, the `Interactable` defines a default response to a particular interaction case - mimicking the concept of "removing items from the tray" through a docking and undocking behaviour. When a user releases an object after manipulation, the `OnRelease()` method is invoked. The system then compares the positions of the `Interactable` object and its `Spawnable` parent: if the distance exceeds a configurable threshold, the parent-child relationship is broken (undocking), allowing the object to operate independently in the scene. Conversely, if the object is brought back within the threshold distance, the hierarchical relationship can be restored (redocking).

A final consideration involves object ownership: since objects are initially spawned by the network server, non-host clients must require ownership over them when interacting. When *OnGrab()* is invoked on an `Interactable`, the system automatically requests ownership via a `[ServerRPC]`, acquiring authoritative control over the object. This once again ensures consistent behavior and synchronized states across all participants.

In summary, spatial anchoring and user interaction are clearly separated for flexibility: *Spawnable*s provide a shared positional context, and *Interactable*s define reusable, device-independent interaction logic. Together, they enable a coherent MR experience across both AR and VR users.

A complete example of the usage of this system will be provided in Chapter 6, when testing is further discussed.

### 5.4.2 Key Design Decisions and Analysis

Throughout this section, the most relevant design choices were discussed in detail. A summary is provided below:

1. The **separation of `Spawnable` (anchoring) and `Interactable` (manipulation) objects** cleanly decouples spatial alignment from interaction logic, facilitating reuse through interfaces and enabling cross-device compatibility;
2. **Server-authoritative synchronization** ensures consistency, with marker detections processed exclusively by the host through asynchronous `ServerRpc` calls. This prevents race conditions and guarantees a single authoritative state across participants;
3. The **ownership-request protocol for object manipulation** reinforces this consistency by dynamically transferring control during interactions;
4. The strategy of **providing virtual representations of markers for VR users** allows them to maintain context, leveraging the full capabilities of AR cameras while

maintaining fairness across other devices.

5. The implementation described in section provided a first example of how using **OpenXR as the abstraction layer supports cross-platform compatibility**, enable the interaction paradigms to extend to a wide range of hardware with minimal adaptation.

In addition, as to prevent VR users from being locked out of marker-dependent experiences, the system provides mechanisms to manually spawn marker-associated objects even without detection, extending their range of possible interactions.

However, as a last note, it is necessary to mention one unresolved edge case: remote collaboration between AR users. The current implementation assumes marker visibility within a shared physical space. If two AR users are collaborating remotely, only one of them would actually see local physical markers. This creates an imbalance, as one user can anchor and interact with precision, while the other is effectively blind to the reference.

A straightforward solution is to extend the strategy developed for VR: when a marker cannot be seen locally, the system provides a virtual representation in the scene.

In order to recognize when this fallback situation needs to occur, the system could incorporate mechanisms such as comparing the geographical locations of participating users. This remains an opportunity for future work.

## 5.5 Device Management

The previous sections established the foundational infrastructure for collaborative mixed reality experiences, comprising the two server-side components illustrated in Figure 4.1: a Python-based detection server for object detection and pose estimation, and a Unity-based server for scene management, real-time synchronization of users within a shared virtual space, and virtual content lifecycle management. However, one critical component remains to complete the architecture: the client-side implementation.

This part of the system refers to the logic running on each individual device, functioning like the "brain" of each participant. It is responsible for handling interaction and communicating with both the Python and Unity servers. The discussion of two critical requirements has been left for this section, as they are intrinsically tied to device-specific considerations.

First, the system must account for the variety of supported hardware, leveraging OpenXR and other architectural paradigms to guarantee a unified interaction model and consistent session lifecycle across devices. Second, the "missing link" in the marker detection pipeline must be clarified. While Python's detection algorithms and Unity's object management have already been described, several gaps remain: how the frames used for detection are actually obtained, how to ensure that these captures are suitable for a detection server that is input-agnostic, and, most importantly, how to translate the resulting normalized 2D detections into Unity's 3D coordinate system so that markers can be accurately aligned.

In short, this component resolves several of the open questions posed in the preceding sections:

- How can the system recognize and handle different device types?
- How should the experience be adapted depending on different device characteristics?
- How to optimize device resources throughout the course of a session?

In the case of AR, the implementation must further address:

- How to complete the marker detection workflow, communicating with both the Python and Unity servers?
- How to obtain and prepare frame captures from the camera feed for marker detection?
- How to convert the 2D detection coordinates into 3D positions for spatial alignment in the Unity space?

For clarity, the description of the implementation is divided as follows: Subsection 5.5.1 examines how the general handling of devices is attained, namely exploring how OpenXR is leveraged to account for their varying characteristics and the management lifecycle throughout a session. Subsection 5.5.2 then narrows the focus to AR-specific marker detection, including camera management, communication with the remaining components, and the logic required to convert detection results into spatial coordinates. Finally, Subsection 5.5.3 summarizes the key design decisions discussed in this chapter.

### 5.5.1 Technical Implementation - General Overview

Figure 5.18 illustrates the class diagram depicting the core device management architecture.

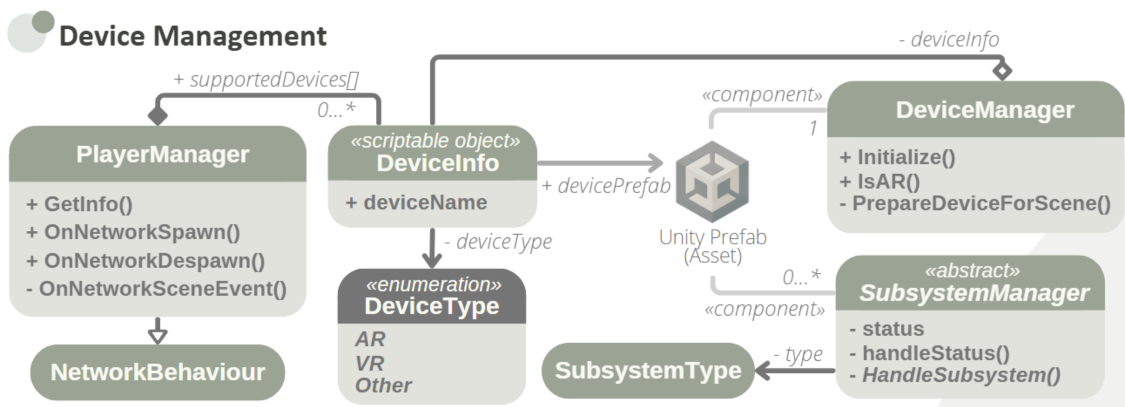



Figure 5.18: *Diagram with the device management classes used across all MR hardware.*

To explore this architecture, Subsection 5.5.1.1 begins by examining the implementation strategies that allow the system to create a consistent device interface, regardless of hardware type, focusing primarily on the `DeviceInfo` class. Subsection 5.5.1.2 then describes how these abstractions are applied throughout a session, detailing the device management lifecycle from the moment the application is launched.

#### 5.5.1.1 Handling Different Devices

While OpenXR provides a strong foundation for interoperability, it does not automatically resolve every device-specific considerations. Its main strength lies in abstracting low-level hardware differences, but each individual device still requires some tailored configurations to fully leverage their unique capabilities.

In practice, OpenXR provides what is known as a “rig”: a prefab that translates a device’s setup into Unity, allowing the user to see and interact with the virtual content. These rigs include components that generalize camera, controller, and tracking systems: they can be thought of as a blueprint, with each manufacturer then implementing slight variations to optimize performance for their hardware. For example, a HoloLens rig might include specialized gaze interaction components, while a Varjo headset rig recommends high-resolution camera configurations. The result is a collection of prefabs that still conform to the same underlying structure, while introducing small, device-specific differences.

This raises a key question: how can the application ensure that, upon launch, each participant is automatically assigned the correct rig variant, providing an experience optimized for their specific hardware? To achieve this, the system needs a single source of truth: a standardized abstraction that stores device information and makes it easily accessible to other components.

This challenge mirrors a problem addressed earlier, when scene generalization was discussed (Section 5.2): the solution is using a `SciptableObject`. Thus, `DeviceInfo` objects are implemented to centralize all relevant device information into a lightweight, easily accessible data structure. Each instance defines three attributes:

1. **Device Name:** Corresponds to Unity’s `SystemInfo.deviceModel`, which provides the technical identifier of the hardware on which the application is running (e.g., “MagicLeap2” or “OculusQuest”). Necessary for matching a participant’s device at runtime.
2. **Device Prefab:** A reference to the prefab containing the OpenXR rig variant configured specifically for the device. These incorporate manufacturer-specific recommendations, including the necessary hierarchy of components for that device’s interaction model: camera setups, input mappings, tracking modules, and any device-specific optimizations.
3. **Device Type:** An enumeration classifying the device as ‘AR’, ‘VR’ or ‘Other’. The final category allows the application to run on desktop systems, providing a “dummy” device configuration that simulates XR behavior through keyboard and mouse input.

As with other `ScriptableObjects` used throughout the system, these objects can be created and configured directly within the Unity Inspector without altering code: a developer simply specifies the device identifier, drags in the prefab reference, and the system automatically incorporates the new device into its supported hardware list - see Figure 5.19.

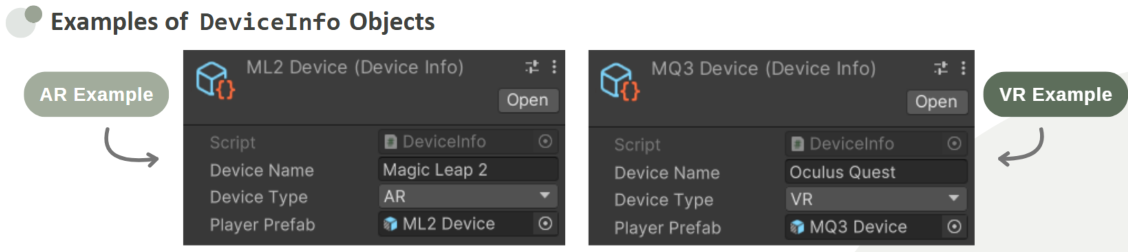



Figure 5.19: *Example of the creation of `DeviceInfo` assets for AR and VR devices - respectively MagicLeap2 and MetaQuest3.*

This architecture maintains a separation of concerns by isolating device-specific logic within individual prefabs and ensures that, despite hardware differences, all devices can be integrated into the system through a consistent abstraction. This enables extensibility for supporting new hardware types without requiring changes to the core application logic.

In addition, it supports independent device updates and even custom device configurations, meaning that if a manufacturer releases updated rig prefabs or if developers need to create specialized configurations for particular use cases: adding support for a new device is as simple as defining a new `DeviceInfo` and linking its corresponding prefab.

#### 5.5.1.2 Device Lifecycle and Subsystem Management

With the device abstractions defined, the final challenge lies in ensuring each client is assigned the correct rig, and enabling a consistent device management approach throughout the session. The architecture outlined so far has briefly touched upon some details of this process, but this subsection consolidates those elements to clearly examine the complete workflow.

A first key to this process is the implemented `PlayerManager` class, the only component placed in the Persistent Scene that has not yet been discussed (refer to Section 5.2). It serves a simple but critical task, maintaining a single attribute: a registry listing all `DeviceInfo` configurations representing the supported hardware devices. During application startup, parallel to the initialization of all the aforementioned persistent systems, it queries `SystemInfo.deviceModel` to identify the hardware platform currently in use. Using this identification, it searches through the registry to find the matching `DeviceInfo` entry, instantiating the corresponding prefab rig as a child object.

Despite its simplicity, the design details behind the `PlayerManager` serve a strategic purpose. By being placed in the Persistent Scene, the system ensures that each participant receives their optimal rig configuration at the earliest possible moment, ready for use with no further delay. The object persists throughout the session, meaning the player's rig remains stable across all scene changes with no need for re-initialization. Beyond this, the `PlayerManager` is a `NetworkBehaviour`, meaning visual elements can be attached to the rig, effectively serving as a virtual avatar of the user that all other players within a shared network session can see. This enhances the sense of presence and collaboration.

After this initial setup, the responsibility of handling device resources and behaviour changes falls to two components embedded within each rig prefab:

1. `DeviceManager`: This is a required component present on all rig prefabs. Its role is to ensure that the user always receives an optimal experience adapted to their specific device type.

   While its responsibilities can be extended in future iterations, it is currently tasked with providing appropriate immersion throughout scenes. When a scene change occurs, it responds to the `OnSceneChange` event by invoking `PrepareDeviceForScene()`. This method adapts the user's visual setup based on their device type and the requirements of the new scene, by cross-referencing data from both the current `DeviceInfo` and `SceneInfo`. For example, during menu navigation, while AR users continue to see their physical surroundings, VR users are placed within a virtual environment to enhance immersion.

   In short, this mechanism allows each device to adapt dynamically to the context of a new scene.

2. `SubsystemManager`: Implemented as an abstract base class, it enables the configuration of specific, customized tasks (such as marker detection) while providing a generalized resource management approach. As has previously been expressed, since certain behaviours are not necessary for all educational experiences, this class allows these subsystems to automatically toggle themselves as needed throughout the session.

   Each individual manager declares two attributes: a `SubsystemType` enumeration that identifies what task the specific subsystem is used for, and a flag indicating whether it should currently be active. Listing 5.5 illustrates the base class implementation:

```
1  protected SubsystemType _type;
2  private bool isEnabled;
3
4  /... Start() and OnDestroy() subscribe and
5          unsubscribe from scene change events .../
6
7  void Update() { if (isEnabled) HandleSubsystem(); }
8  protected abstract void HandleSubsystem();
9
10 // Method invoked when OnSceneChange event occurs
11 void handleSubsystemStatus(/.../) {
12     isEnabled = e.sceneInfo.RequiresSubsystem(_type);
13 }
```

Listing 5.5: Abstract `SubsystemManager` implementation.

   When a scene transition occurs, `handleSubsystemStatus()` is invoked. The `SceneInfo` data structure for the new scene is retrieved and used to verify whether the task managed by this subsystem is required, updating the status value accordingly. If active, the abstract `HandleSubsystem()` method is called on each frame update, allowing specific implementation to perform their personalized tasks.

This architecture provides a consistent interface that enables developers to extend the base rigs and introduce highly specific behaviors (for example, frame capturing and transmission, or custom gesture recognition) without burdening the entire system: the relevant subsystems are only activated when required. The abstraction even enables the same task to be implemented differently for different device rigs, allowing behaviors to be finely tuned to the unique characteristics or limitations of a particular device without affecting others.

Using this design, a subsystem needed for marker detection can remain inactive in menu scenes, avoiding unnecessary processing and power consumption, and automatically activates in educational scenes where object recognition is essential.

Given this cue, and to better understand this architecture, the following subsection examines one specific subsystem implementation in detail: the AR-specific marker detection system that completes the computer vision pipeline described in earlier sections.

### 5.5.2 Marker Tracking in Augmented Reality

Marker detection is one of the specialized subsystems that certain devices - namely, those equipped with color cameras and depth-sensing capabilities - may implement. These conditions are, in general, met by all modern AR hardware.

To manage this subsystem, the `DetectionManager` class is implemented. This component is illustrated in Figure 5.20, along with the remaining classes that complete the implementation.

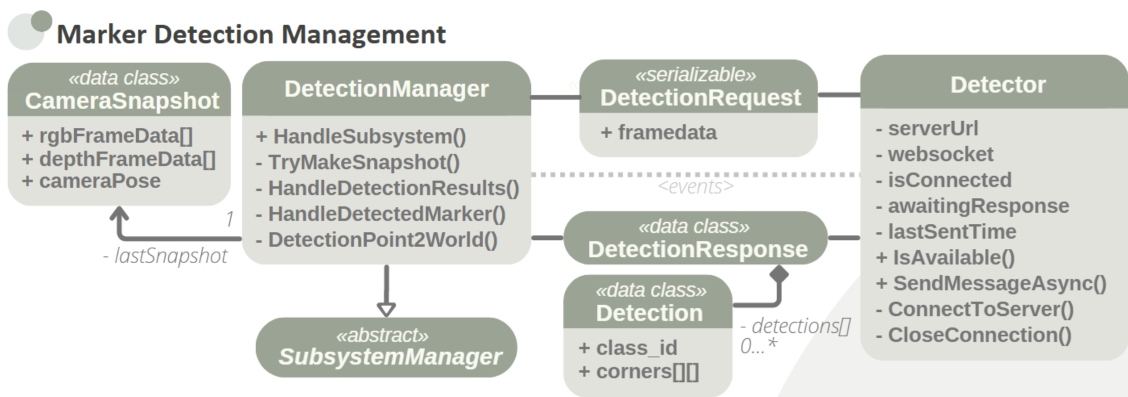



Figure 5.20: *Diagram depicting classes relevant to complete the marker detection workflow.*

Subsection 5.5.2.1 begins by revealing how the 2D detections can be converted to Unity's 3D space for object alignment. Once the requirements for this operation are contextualized, Subsection 5.5.2.2 details how the necessary data is obtained and sent to the detection server, focusing on the role of the `DetectionManager` class to finally complete the detection pipeline.

Lastly, Subsection 5.5.2.3 provides some final considerations regarding camera configurations before Subsection 5.5.3 summarizes the key design decisions.

#### 5.5.2.1 The Challenge of Shared Space for Spatial Alignment

Before detailing how communication with the detection server is achieved, it is first necessary to clarify *what* information must be first gathered and transmitted. What data is necessary to ensure not only accurate detection, but also precise spatial alignment?

At first glance, the answer seems simple: a frame capture from the device's camera, working like a "screenshot" of the user's vision at a certain point in time, so that objects can be detected on it. But in the greater scheme of the process, this alone is not enough.

As explained in Section 5.1, the Python server returns, per detection, the four 2D corner coordinates that specify where the marker appears in the image. However, two major challenges remain.

First, these detections only exist in image space: they describe horizontal and vertical placement (up/down, left/right), but not how far the points are from the user. This reconstruction into the 3D space requires depth information. Fortunately, AR devices are equipped with depth sensors, providing images in which each pixel encodes the estimated distance from the camera. When a color frame and a depth frame are properly aligned, a 2D detection coordinate can be matched with its depth value, yielding a 3D point relative to the device. Some nuances of this depth–color alignment are discussed in Subsection 5.5.2.3.

With this first issue addressed, a second still remains. Each device effectively considers itself the origin of its own miniature universe, defining its own understanding of what '*up*', '*right*' and '*forward*' mean based on the user's position and orientation. Thus, even once the 3D coordinates are obtained, they are still expressed in the local coordinate system of the device. To enable collaboration, these must be transformed into a shared reference frame that all devices can understand.

A key to solving this lies in the fact that users are already positioned within a global coordinate system: the Unity scene itself (see Figure 5.21). Here, the device rig's camera plays a crucial role, as its pose (position and orientation) at the moment of frame capture defines the mapping between the device's local space and Unity's world space.

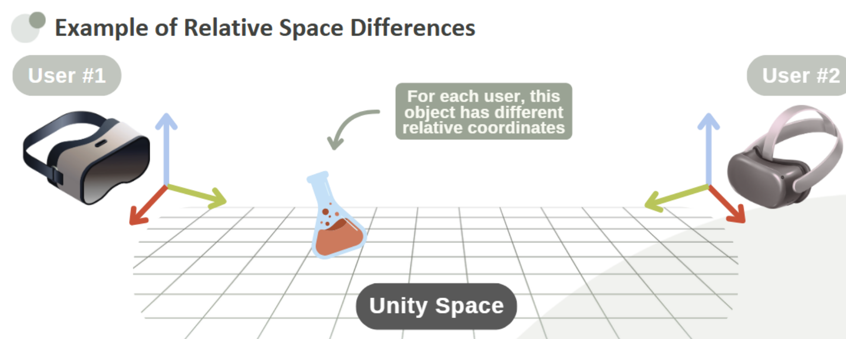



Figure 5.21: *Example illustrating a major challenge in spatial alignment: users places within a common coordinate space (Unity) have different understandings of a same object.*

In other words: detections first exist "in the user's eye" (device space) and must be "translated" into the shared stage (Unity space). This transformation is handled by using Unity's built-in `ScreenToWorldPoint()` method, which projects a point from screen coordinates into Unity world coordinates, given its depth value and the camera pose at capture time.

The process applied to each received detection follows the sequence illustrated in Figure 5.22:

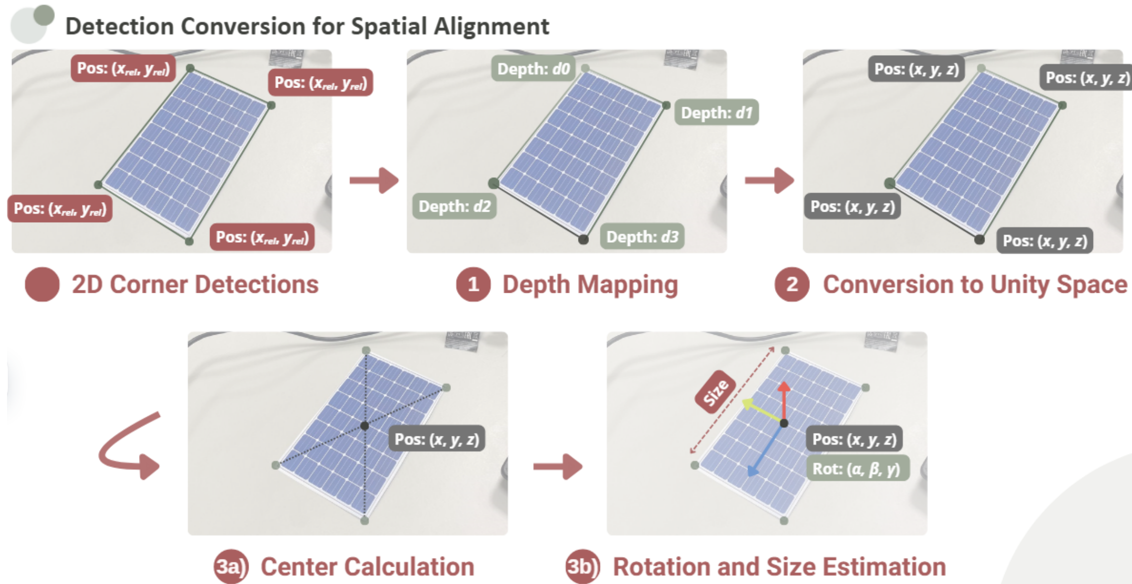



Figure 5.22: *Detection conversion pipeline for spatial alignment.*

1. **Obtaining 3D Corners:** Each corner coordinate $(x, y)$ from the RGB frame is matched with its depth value from the depth frame, producing $(x, y, d)$. The units for a depth frame are not standardized to a single system like meters or feet, but OpenXR provides methods to convert these values into Unity units to ensure consistency.
2. **Converting to Unity:** These relative coordinates are transformed into Unity world space using the camera pose and the built-in `ScreenToWorldPoint()` method.
3. **Marker Pose Calculation:** From the four 3D corners:
   a) The marker's center is found by averaging the coordinates.
   b) The orientation and size are derived from the dominant side vector;

To ensure consistency, whenever the client wants to make a detection request, all this information must be captured simultaneously - as if "freezing" a single moment in time. For this purpose, the `PixelSensorSnapshot` data class is implemented to store:

1. **Color Frame:** Providing RGB data for each pixel;
2. **Depth Frame:** Providing distance-to-camera values for each pixel;
3. **Camera Pose:** Recording the exact position and orientation of the device's camera at capture time.

With these elements combined, the system can reconstruct detections into the shared 3D space, ensuring markers appear consistently aligned across all collaborating users.

#### 5.5.2.2 Completing the Marker Detection Workflow

With the `PixelSensorSnapshot` structures contextualized, all that remains is integrating them into the detection pipeline, providing a comprehensive view of the complete workflow. This section details how communication with the Python detection server is implemented, as well as how responses are handled and integrated back into the Unity environment.

The detection workflow revolves around two main classes:

- `Detector`: An auxiliary class that orchestrates the socket communication with the Python server. It encapsulates all logic required for connecting, transmitting requests, and handling responses.
- `DetectionManager`: A `SubsystemManager` implementation that orchestrates client-side detection operations across frame capture, server communication, and spatial alignment. As a `SubsystemManager`, it is only active within learning modules that the marker detection task to be fulfilled.

The `Detector` class is included in all device rigs capable of marker detection and features a single configurable attribute: the Python server endpoint (exposed via Ngrok, as detailed in Section 5.1). It provides a series of utility methods to establish and close connections and, most importantly, a public method that enables other components to send data to the server without blocking the Unity main thread - `SendMessageAsync()`.

The complete detection workflow, involving a user device, the Python server, and the Unity networking server, is illustrated in Figure 5.23.

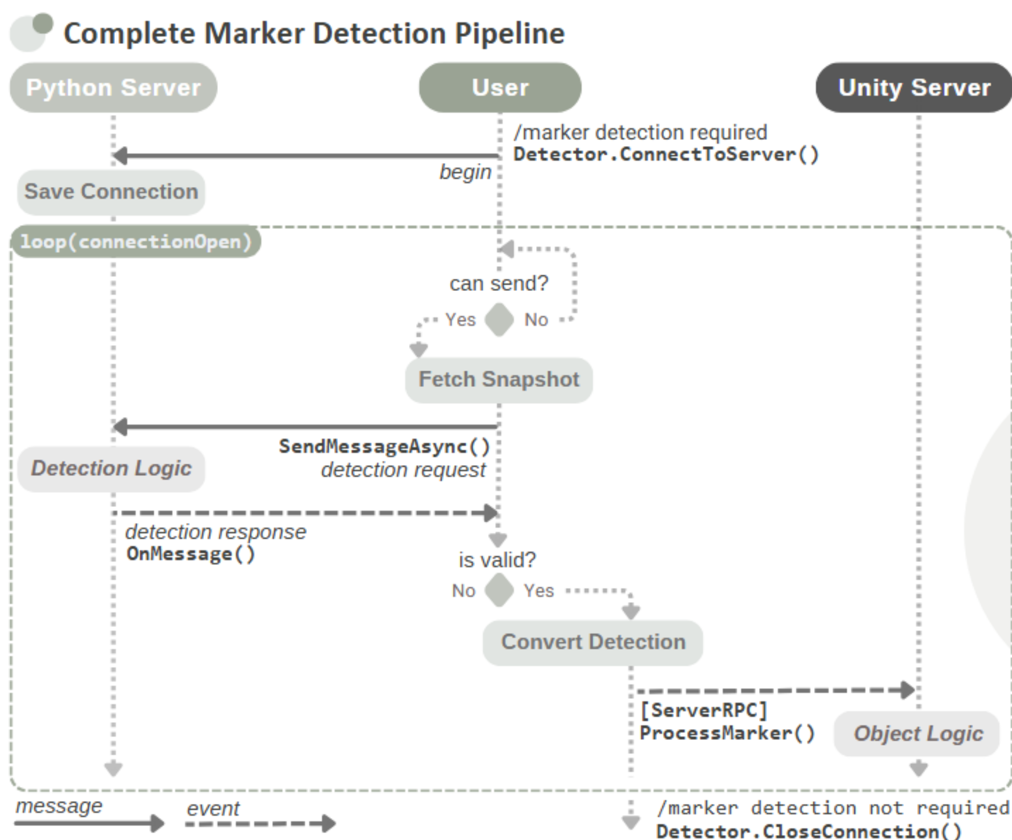



Figure 5.23: *Marker detection interaction diagram.*

As illustrated, the workflow proceeds through the following key stages:

- **Server Connection:** To avoid unnecessary resource usage, connections are established only when strictly necessary. When a user begins a learning module requiring marker detection, `Detector.ConnectToServer()` is invoked, establishing a WebSocket connection. Upon successful connection, the `Detector` begins listening for socket events, in particular `OnMessage()` for handling detection responses and `OnError()` for properly dealing with potential failures.
- **Sending Requests:** As a `SubsystemManager`, the `DetectionManager` executes its core functionality in `HandleSubsystem()`, called every frame when the subsystem is active. The request cycle unfolds as follows:
  - **Availability Check:** Before a new request is issued, the system verifies that the `Detector` is ready via `Detector.IsAvailable()`. This ensures that the connection is stable and enforces a strict request/response rhythm: no new request is sent until the previous one has either been answered or considered expired after a timeout. This prevents both server flooding and indefinite blocking in the case of unexpected errors or stale responses.
  - **Frame Capture:** When a new request is due, a `PixelSensorSnapshot` is captured - a "freeze frame" of the user's current visual state containing a synchronized color frame, depth frame, and camera pose data.
  - **Message Transmission:** This snapshot is serialized into a `DetectionRequest`, which only contains the color frame. This ensures only the information strictly necessary for detection is transmitted, optimizing bandwidth. This structure is sent via `Detector.SendMessageAsync()`.
- **Response Handling:** Whenever a server reply is obtained, the message is deserialized into a `DetectionResponse`. If valid detections are present, the `OnDetectionReceived` event - to which the `DetectionManager` subscribes - is triggered. This event-based mechanism keeps responsibilities decoupled: the `Detector` handles only message exchange, while the `DetectionManager` integrates the results into the application context.
- **Marker Processing:** When notified of new detections, the `DetectionManager` applies the conversion steps detailed in Section 5.5.2.1: the 2D coordinates are mapped into the 3D space, reconstructing the marker's pose (center, size, and orientation) within the Unity scene. This data is then transmitted to the `NetworkObjectManager` via `[ServerRPC]` calls (refer back to Section 5.4), enabling synchronized object alignment across all connected clients.
- **Server Disconnection:** When a user exits a module that no longer requires marker detection, the connection is closed via `Detector.CloseConnection()`, releasing resources and preventing lingering sessions. This is also enforced on application quit, avoiding errors or memory leaks.

In summary, the workflow is built around an event-driven communication model: the `Detector` manages connections and message exchange, allowing higher-level components to subscribe to its events and integrate detections into the context of the application context.

Furthermore, error control is handled at multiple layers. The implementation includes a timeout detection for unusually delayed responses, allowing new requests to be issued without disrupting the user experience; since the Python server is prepared to manage duplicate requests, potential conflicts are automatically avoided. In addition, the system transparently recovers from dropped connections through reconnection attempts. Together, these strategies ensure that the detection pipeline remains resilient and responsive, preserving a seamless experience for users even under unreliable network or server conditions.

#### 5.5.2.3 Final Considerations about Frame Capturing and Distortion

As a final note, one challenge cannot be overlooked - one that exemplifies the very problem that motivated this project and that makes the maintenance of MR applications particularly complex: device variety and its impact on camera systems.

As discussed in Section 5.1, the Python-based detection server was designed to be input-agnostic, meaning it processes received frames without knowledge of their origin and performs detection without additional pre-processing. While this improves response times by eliminating an additional step, the deeper motivation behind this choice is that such processing makes no practical sense at any other point in the system other than on the originating device itself.

Only the device possesses knowledge of its own camera characteristics and limitations. AR camera systems can vary widely across manufacturers and device generations: differences in color calibration, orientation, distortion patterns, or capture resolutions can all influence quality and, in result, detection accuracy. Much of this variation stems from the rapid pace of innovation in this field, where manufacturers continue to experiment with different approaches to camera integration and image processing pipelines.

In practice, this means that each device rig must include scripts capable of both operating the device's camera and processing their frames appropriately. In other words, these scripts must ensure the correct mechanisms to convert raw sensor data into images that the detector can reliably interpret.

Fortunately, device manufacturers typically provide comprehensive documentation and development kits that include complete camera operation classes, covering common tasks like streaming, frame capturing, and format conversion, along with detailed guides on how to adapt these functionalities to various specific needs.

For instance, the MagicLeap 2 device - the AR hardware used for testing - demonstrates a particularly interesting configuration challenge: its color and depth cameras operate in completely different projection systems, with the color camera using a traditional pinhole model, while the depth camera employs a fisheye configuration. In everyday terms, it is as if one eye looks through a standard lens while the other peeks through a wide-angle peephole - the pixels do not directly match up between the two.

For reference, figure 5.24 illustrates this effect.

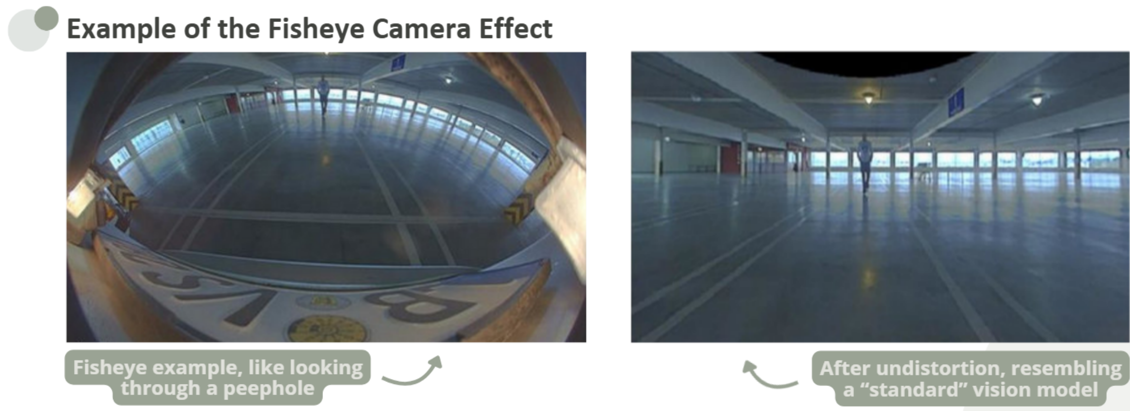



Figure 5.24: *Example of the fisheye effect, comparing an original capture with its undistorted version after correction algorithms are applied [66].*

This distinction creates a spatial challenge: detections are computed on the color frames, meaning their coordinates are defined in that camera's projection system. Yet, determining the 3D position of detected markers requires depth values from the fisheye camera. To reconcile the two, either the depth image must be undistorted and registered to the color camera, or each detection coordinate must be transformed into the depth camera's system.

In either case, the manufacturer provides a detailed description on how the camera's intrinsic and extrinsic parameters can be used to perform these transformations. The implementation therefore consists of following these specifications to prepare the necessary conversion methods. Since these steps are device-specific and well-documented, they are not further examined in this report. However, the resulting class can be consulted in Appendix A.

This camera pre-processing requirement represents the only component of the entire system architecture where manufacturer-specific implementations truly cannot be avoided. However, considering the context of the broader system design, the necessary setup remains relatively minimal: as long as the `DetectionManager` class ensures that appropriately formatted frames are fetched and sent to the detection server, the remainder of the system operates seamlessly across device types.

In addition, it is worth noting that many modern AR devices do share similar camera systems and approaches, which helps minimize these compatibility issues. The most significant challenges arise only with devices featuring particularly unique camera configurations or more experimental optical systems.

### 5.5.3 Key Design Decisions and Analysis

This device-specific management step, while unavoidable, demonstrates the effectiveness of the overall architecture: by isolating hardware-specific requirements within individual device rigs and maintaining standardized interfaces for all other system components, the

impact of device variety is localized and manageable. This separation ensures device fragmentation is contained, preventing complexity from cascading into networking, detection, or interaction layers.

Beyond this, several other design strategies reinforce the system's modularity and extensibility:

1. The **centralized device abstraction** through `DeviceInfo` objects provides a unified interface and isolates hardware-specific configurations within individual rig prefabs. This enables seamless addition of new devices without altering the core architecture and allows clear management of the supported devices list.
2. The **subsystem activation strategy**, implemented via the `SubsystemManager`, prevents unnecessary resource consumption by enabling specialized behaviors only when required by specific educational modules. This allows fine-tuning functionalities for particular experiences without impacting the rest of the system.
3. The placement of the `PlayerManager` in the persistent layer ensures an optimal rig configuration at the earliest possible moment and eliminates re-initialization overhead during scene transitions.
4. The usage of **event-driven communication** reduces unnecessary polling, ensuring critical updates propagate immediately when available.

On the specific case of client-side marker detection:

1. The `Detector` and `DetectionManager` classes maintain a **clear separation between socket communication and spatial integration**, enabling independent updates and testing of each component.
2. The **capturing and conversion pipelines employ standardized structures**, ensuring the detection and networking layers operate on common formats, regardless of the device source.

All these decisions highlight a recurring principle: complexity must be embraced where necessary - particularly at the device layer, which is most affected by fragmentation - but abstracted at higher levels. This preserves consistency and modularity across the system.

With object detection, Unity networking and scene management, object management, and device implementation now fully detailed, the foundational infrastructure of the application is complete. As this concludes the implementation chapter, the next step shifts from design to practical evaluation: these architectural decisions need to be tested and examined in practice, validating their effectiveness across different devices and collaborative scenarios.

To do so, the following chapter provides a walkthrough of the creation of a sample learning module, detailing how objects are prepared and how model training is conducted, while analyzing key aspects such as multiplayer synchronization, cross-device interaction functionality, and the performance and accuracy of the detection model.

# 6 Module Development and Evaluation

This chapter presents the testing process and results evaluation of the developed system. Building on the implementation details outlined in Chapter 5, this analysis aims to demonstrate how the adopted architecture performs in practice, assessing its effectiveness across different devices, interactions, and detection tasks.

To approach this with a realistic use case, the complete development of a sample educational module will be presented, encompassing all of the system's main components: object preparation, model execution, networking, synchronization, and interaction. As such, before delving into the testing phase, it is necessary to concisely contextualize the theme of this module, clearly establishing the interaction goals and the intended evaluation process.

Leading the tests with a sample module draws upon the fundamental principles of this dissertation, and the core educational contribution that it aims to achieve: the creation of tangible learning experiences where abstract concepts, often limited to static textbook graphics, are transformed into interactive phenomena that students can directly manipulate and observe.

The developed demonstration module focuses on a relevant example of this challenge: the concept of solar energy production. Equation 6.1 and Figure 6.1 illustrate the conventional way in which this concept is presented in textbooks.

$$P = I_{\text{panel}} \times A \times \eta = I_{\text{DNI}} \times \cos(\theta) \times A \times \eta \quad (6.1)$$

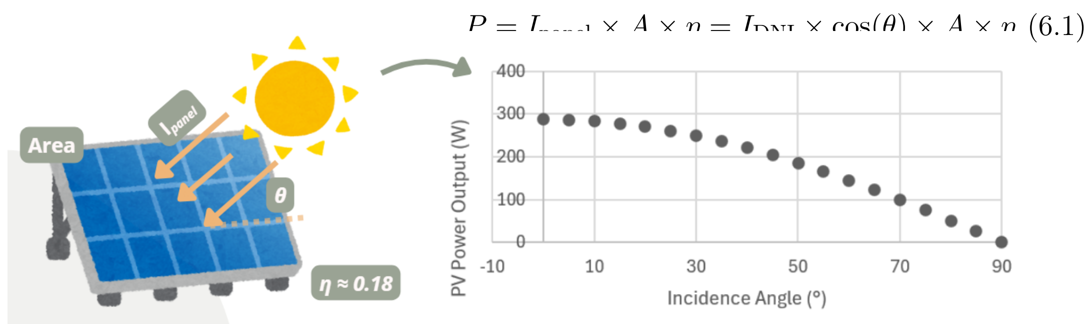



Figure 6.1: *Conventional schematic representation of the solar panel power output, expressed as a function of direct normal irradiance ($I_{DNI}$), incidence angle ($\theta$), panel area ($A$), and efficiency ($\eta$), according to models reported in the solar energy literature [67, 68].*

Traditional education often presents solar energy as a collection of equations and efficiency curves: students are left memorizing static values rather than developing a true, intuitive understanding of *how* each underlying phenomenon impacts the system on its own. How can they truly grasp the way sunlight, angle, and weather affect energy production, if these notions exist only as intertwined variables on paper? Learners need opportunities to manipulate and directly experience each factor separately, bridging theory and practice in a way that enhances both engagement and educational impact.

It is within this context that the proposed educational module on solar energy production emerges. Designed to demonstrate the interaction and synchronization capabilities of the platform, it creates an immersive and collaborative learning scenario in which users can experiment with the various environmental variables and immediately observe their individual impact on energy production. The developed module allows users to:

- Use miniaturized solar panel replicas as physical markers to trigger the appearance of virtual objects - a Sun and clouds - upon detection.
- Adjust the position of the virtual Sun through gestures, altering the angle of light incidence on the solar panel.
- Modify the level of cloud cover, simulating different atmospheric conditions.
- Observe how these changes affect the calculated power output, allowing all connected learners to observe how these physical variables influence energy production efficiency.

While simplified, this scenario demonstrates the system's ability to transform abstract concepts into observable interactions. It also illustrates the potential for integration into educational contexts focused on sustainability and environmental awareness, offering a pedagogical tool that combines immersive technology with clear learning objectives.

Tying back to the main goals of this chapter, the evaluation methodology follows the analysis of the demonstration module:

1. **Object Preparation and Interaction:** Section 6.1 describes the creation of the `Interactable` objects that support the module's interaction goals, further highlighting the implemented generalization mechanisms. The testing focus is on ensuring their manipulation is consistent across different device rigs - in other words, if cross-device functionality is attained.
2. **Cross-Device Multiplayer Validation:** After integrating the objects into an application scene, Section 6.2 walks through the complete user flow, from launching the application and navigating menus, to joining a shared session. The aim is to test the sample module in a group, verifying how the system handles real-time synchronization, ensuring that any change made by one user is immediately visible to all others.
3. **Detection Model Training and Evaluation:** Finally, Section 6.3 covers the training of the YOLO model used to recognize the solar panel replicas. Both speed and accuracy are evaluated, showing how the model integrates into the scene to meet the module's requirements and complete the experience.

It is relevant to note how these components can be developed and tested independently, and in any sequence, before integration into the complete system. This fact further highlights how the modular architecture enables efficient parallelized development.

## 6.1 Object Preparation and Interaction

This section illustrates how concrete interaction requirements can be translated into application-ready objects. At the core of this process lies the abstraction paradigm implemented in the `Interactable` class, as introduced in Section 5.4.

Throughout this section, these behaviors are evaluated using Magic Leap 2 and the Meta Quest 3 devices - examples of Augmented and Virtual Reality technologies, respectively. These allow a preliminary assessment of whether interaction remains consistent across distinct MR paradigms.

Given the extensive detail regarding device rigs in Section 5.5, the description of how these components are prepared will be very briefly summarized in Subsection 6.1.1, with Subsection 6.1.2 then presenting the implementation of the objects and demonstrating their behavior within those setups.

The main testing goals for this stage are:

i) **Cross-device Consistency:** Verify that interaction is consistent across both device types, and that the visual VR representation of objects is appropriately applied.
ii) **General Interaction Behaviors:** Confirm that core `Interactable` features, such as docking and undocking from the `Spawnable`, function as expected.
iii) **Module-specific Behaviors:** Validate the implementation of the module-specific interactions. This encompasses ensuring that moving the Sun and clouds updates their intrinsic parameters (e.g. angle and light intensity) and that the associated energy conversion formula responds to these changes accordingly.

These initial tests are conducted in a single-player setup with objects being automatically spawned, as marker-based detection will be introduced in later phases.

### 6.1.1 Preparing Device Rigs

In order to enable object manipulation in the context of MR development, device rigs configured for the target devices must be prepared, enabling the user's input to be accurately tracked. A brief overview of the preparation process for a device rig and its corresponding `DeviceInfo` is as follows:

1. The OpenXR Unity plugin is configured according to the respective manufacturer's recommendations. This step produces a base device rig prefab, which includes the logic necessary to appropriately track a given device's inputs;
2. `DeviceInfo` objects are prepared with the required parameters and registered in the supported devices list.

Figure 6.2 shows the finalized setup.

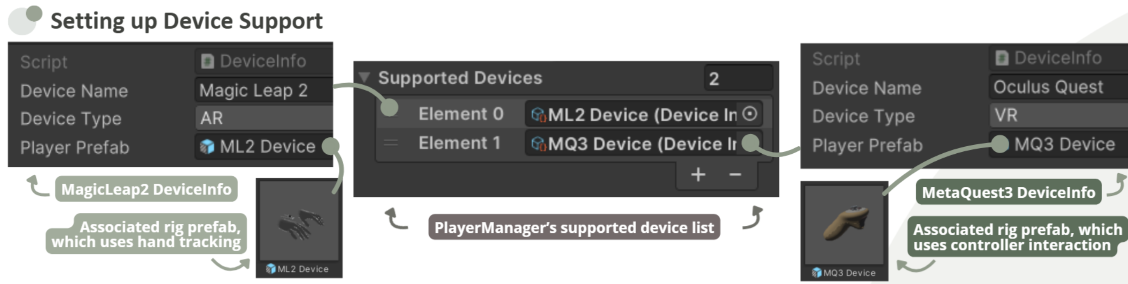



Figure 6.2: *Finalized setup, visualized in the Unity Inspector: two* `DeviceInfo` *assets linked to their corresponding device prefabs and interaction types.*

The figure also highlights the default visuals for each rig, showing how each interaction type is visually conveyed: the AR device shows a digital representation of the user's hands, while the other displays a virtual controller.

With these configurations in place, the interactive objects can now be introduced.

### 6.1.2 Preparing and Testing Custom `Interactable` Objects

To ground the discussion that follows, Figure 6.3 showcases the Unity prefab that supports all of the module's interaction logic.

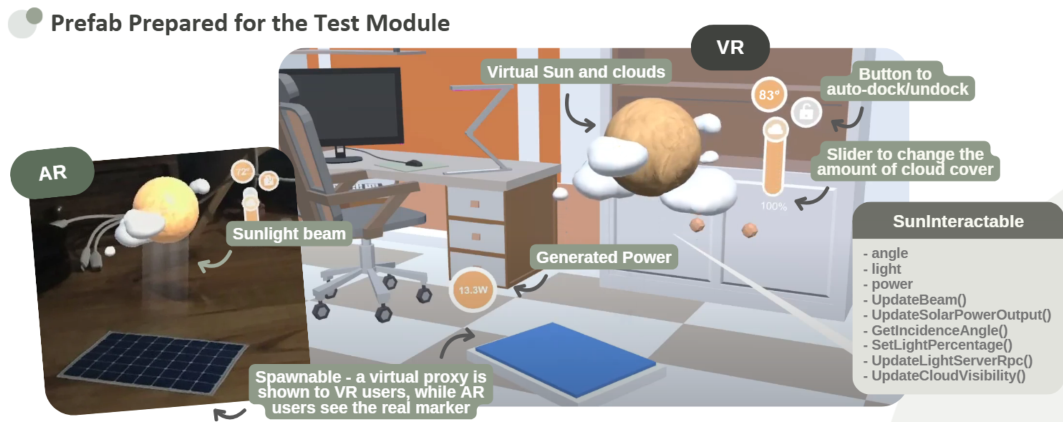



Figure 6.3: *Prefab prepared for the test module, shown in both AR and VR. Key interface elements are highlighted, including the virtual Sun with clouds, value visualizations, and other interactive components. Both the physical marker and its virtual proxy are visible in their respective views.*

With the proposed architecture, all interactions can be set up by extending a single abstraction: the `Interactable` component introduced in Section 5.4. For this module, this is implemented through the `SunInteractable` class (Appendix B), orchestrating the simulation of how varying atmospheric conditions affect photovoltaic energy production. In other words, it transforms the abstract principles depicted in Figure 6.1 into a manipulable, observable experience in the 3D space.

At the core of this implementation are three key variable attributes, which represent: the incidence angle (`angle`), the available light intensity (`light`), and the resulting power output (`power`). All of these are implemented as a `NetworkVariable`, meaning that whenever a user's actions modifies one of them, the update is instantly synchronized across all connected devices. This ensures a consistent shared experience without requiring each client to perform redundant calculations.

To understand how these attributes are updated, it is helpful to walk through the interaction capabilities provided by the presented prefab. Each of these elements ties directly into the methods of `SunInteractable`, shaping the flow of the simulation. Throughout the enumeration below, the functioning of each component is demonstrated and verified using the prepared rigs, offering a tangible illustration of the described behavior:

1. **Solar Panel:** This object represents the anchoring `Spawnable`. It visually tracks a target physical marker, serving as the parent for the remaining components. It provides the reference point for positioning, orientation, and calculations of solar incidence.
2. **Atmospheric Condition Control:** A simple slider interface allows users to vary cloud coverage. Adjustments trigger two effects: a visual animation of clouds rising or falling in the scene, and an update to the available sunlight intensity (`light`). These changes immediately propagate into the solar power calculation, letting users isolate this variable and experiment with weather conditions in an intuitive way.
   In Figure 6.4, varying cloud coverage values are shown, serving as a test of this implementation.

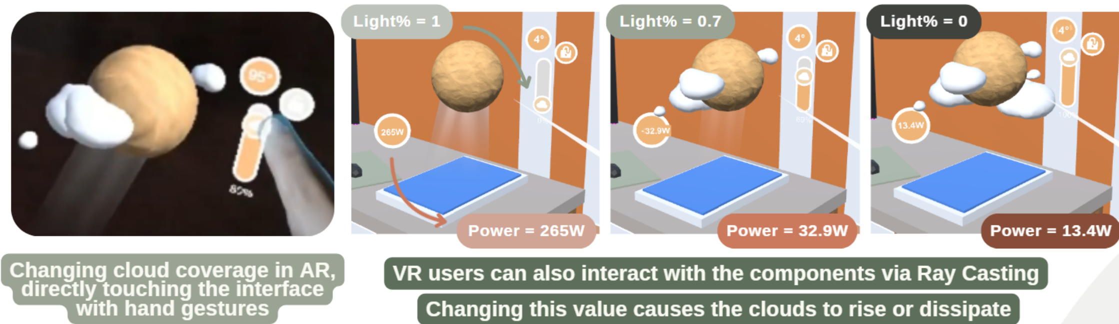



Figure 6.4: *Testing atmospheric control: the slider interface lets users adjust cloud cover.*

   As can be seen, changes in cloud coverage affect the estimated power output as expected: lower coverage increases available sunlight, resulting in higher solar power generation. This test validates the implementation of the `light` variable.
3. **Beam Visualization:** Also visible above, a `LineRenderer` connects the Sun to the solar panel, making the incidence of light visible. Its thickness and transparency adjust dynamically to the cloud coverage, reinforcing the visual metaphor of irradiance being weakened by atmospheric conditions.
4. **Virtual Sun:** Users can grab and reposition the Sun using standard OpenXR interaction paradigms. Because the `SunInteractable` component is attached to it, the

Sun becomes the reference for the docking/undocking mechanics introduced earlier (see Section 5.4). In practice, its distance to the solar panel determines whether it follows the panel's movements or acts independently.

As the Sun is moved, the system continuously recalculates the solar incidence angle (`angle`) using vector algebra, with an angle of 0 signifying the Sun is directly above the panel. This value is displayed in real time, with users getting direct feedback on how the solar incidence angle affects the power output on its own. Figure 6.5 illustrates the calculation of this value and showcases interaction with the object.

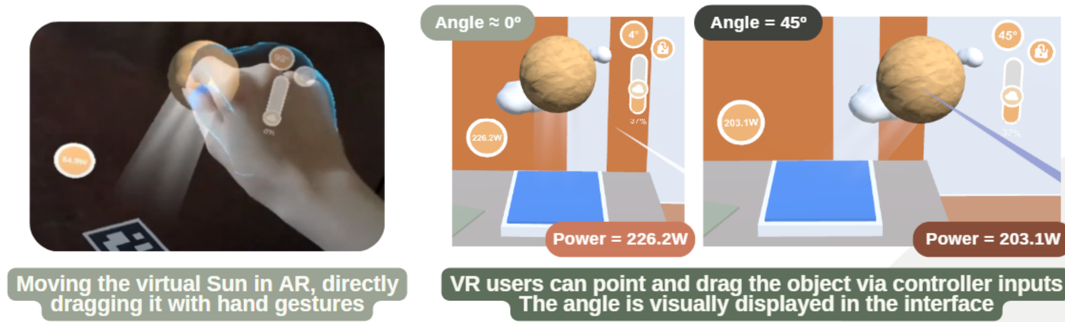



Figure 6.5: *Testing with the virtual Sun, which enables users to adjust the angle of incidence.*

This confirms the correct behavior of the `angle` variable while reinforcing a key pedagogical concept: panel orientation is crucial for maximizing solar energy production. When the Sun is positioned right overhead, light reaches the panel more directly, leading to greater energy capture. At lower angles, the same sunlight is distributed across a wider area, reducing efficiency.

The calculation of the energy output itself directly follows Equation 6.1, with the results being displayed in the interface in real time, as shown throughout the provided figures. Figure 6.6 further compares the simulated values with the theoretical graph, providing a final validation of this computation.

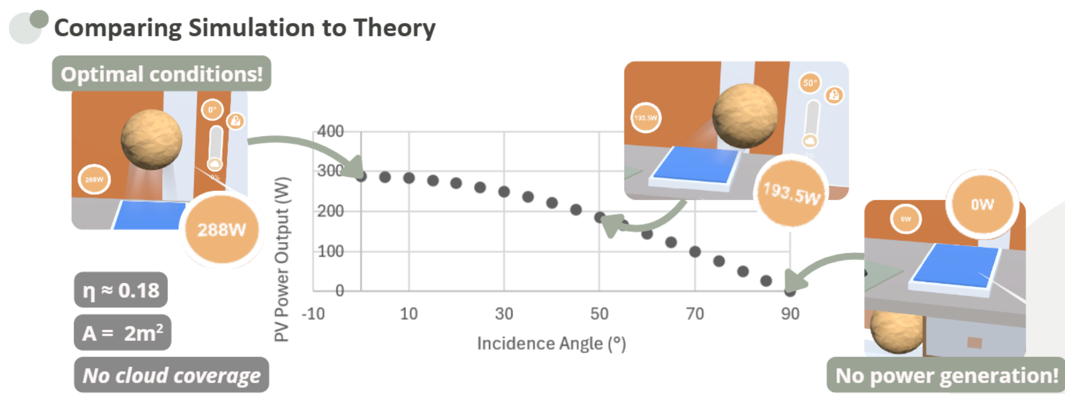



Figure 6.6: *Reference points from the textbook graph were selected replicated in the simulated module, allowing a direct comparison between computed outputs and the expected values.*

For this simulation, the base conditions used when generating the original graph were replicated: no cloud coverage and a standard photovoltaic panel configuration, with only

the Sun's angle of incidence being varied. This testifies to the correct implementation of the formula, with users now being able to truly view *why* this phenomenon affects the generated output the way it does.

In short, by combining these mechanisms, the `SunInteractable` bridges theoretical knowledge and direct experience: rather than studying efficiency curves in abstract, learners can adjust and simulate different conditions with their own hands, immediately observing their effect on energy output. This direct cause-and-effect interaction is where the educational value resides: it turns static equations into observable dynamics that can be studied in isolation.

The tests executed during this first stage confirm the main goals outlined at the beginning of the section: interaction remains consistent across devices, the fundamental `Interactable` features behave as intended, and the solar module's dynamics respond as intended to the user's input. Furthermore, the described approach illustrates how additional `Interactable` objects of varying complexity can be developed, highlighting the platform's scalability for future educational modules.

Building on these results, Section 6.2 provides a video demonstration in a multiplayer setting, offering a more dynamic visualization of these behaviours and confirming that the same interaction paradigms extend seamlessly from single- to multi-user scenarios.

## 6.2 Cross-Device Multiplayer Validation

Integrating the created objects into the main application requires minimal additional development: a Unity scene is prepared to encapsulate the module's logic, becoming immediately available for use upon registry. This section provides an overview of this process before extending the previous single-user tests into a multiplayer validation scenario.

Summarizing, the transition from individual object testing to a executable educational module involves:

1. **Object Integration:** The previously tested `SunInteractable` prefab is placed within a new Unity scene, maintaining the configured networking and interaction properties.
2. **Unity Registration:** The scene is then registered in Unity's build settings to enable transitions.
3. **Configuring a `DeviceInfo`:** A `DeviceInfo` object is prepared, indicating the necessary device subsystems - for this example case, Marker Detection will be required later. The main menu interface automatically has access to this asset.

Once these steps are completed, the module becomes immediately accessible through the platform's existing main menu navigation. This separation of content creation from technical configuration allows educators and developers to focus on pedagogical objectives rather than underlying implementation details. Figure 6.7 shows the finalized setup.

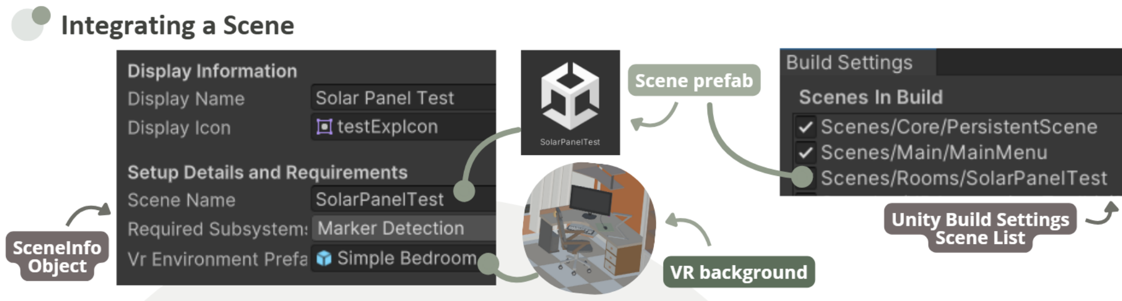



Figure 6.7: *Finalized setup, visualized in the Unity Inspector: the prepared `SceneInfo` is shown, along with the list of scenes included in the build.*

To evaluate multiplayer functionality, a demonstration was conducted with two participants: one using a Magic Leap 2 (AR) device and the other a Meta Quest 3 (VR). Crucially, both users connected to separate networks, simulating the more complex challenge of distributed collaboration rather than optimized local conditions. The test followed a systematic progression through the complete user experience:

i) **Lobby Management:** One participant creates a new lobby while the other joins through the provided menu flow. This step verifies if lobby discovery, creation, and connection occur reliably, even across independent network protocols.

ii) **Scene Transition:** Once the lobby is established, the host initiates the solar energy module. This test confirms if all participants are correctly transitioned to the intended scene, under a shared relay system.

iii) **Cross-Device Interaction:** Both users interact with the simulation using their respective input paradigms (hand tracking for AR, controller input for VR). This verifies whether object interactions are propagated across clients in real time, while maintaining visual consistency across heterogeneous devices.

A video demonstration of this complete user flow is accessible through Figure 6.8, illustrating these various stages in practice from both users' perspectives. A generic marker is used for anchoring, with custom object detection being left for testing in the following section.

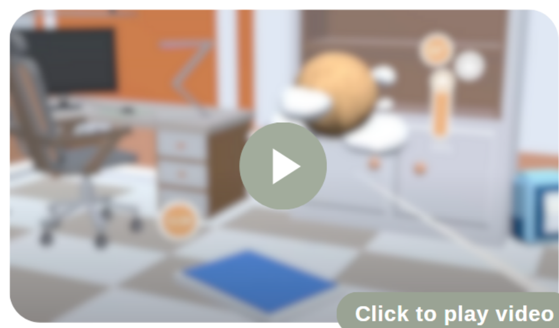



Figure 6.8: *Placeholder image linking to the demonstration video.*

The demonstration validated the key networking requirements of the system. Lobby creation and joining were completed without errors - even under distinct network conditions - and all participants were successfully transitioned into the solar energy module once initiated by the host.

Most importantly, interaction synchronization behaved consistently across participants: when a user manipulated a virtual object, the effect was propagated almost instantly, with no noticeable delays. From a user perspective, these interactions appear natural and entirely continuous, sustaining the illusion of a shared space. This responsiveness is particularly relevant for educational contexts, where fluid collaboration is critical to maintaining engagement.

Upon repeating the complete flow across three tests, performance metrics were gathered for analysis. The wait times for scene transitions and lobby update propagation were estimated via empirical observation, while the times required to synchronize object and variable updates to all users were measured using Unity's Diagnostic Data tools. These results are summarized in Table 6.1.

Table 6.1: Summary of performance metrics obtained from three tests.

| Operation | Average Delay | User Perception |
|---|---|---|
| Lobby Updates | 1–3 s | Minimal, most updates are on demand |
| Scene Transitions | 2–4 s | Slight pause before module is interactive |
| Object Updates | 23–41 ms | Perceived as instantaneous |
| Variable Updates | 18–35 ms | Perceived as instantaneous |

While the lobby and scene transitions introduced short pauses, these are expected considering Unity's backend loading and relay processes. In future iterations, providing visual feedback during these transitions could further improve the user experience by making the short delays more transparent.

More critically, all interactive updates - which include object movements and shared variable reassignments - were **consistently propagated across clients in less than 50 ms**. This value is not arbitrary and will be particularly relevant in later testing: prior work in human–computer interaction identifies delays under approximately 100 ms as essentially imperceptible for synchronous visual and interaction tasks [69]. Maintaining operations within this range ensures that collaboration feels continuous and natural.

Finally, using distinct device paradigms demonstrates the platform's ability to address fragmentation and interoperability, the key challenges of this dissertation. Users are assigned the correct rigs at launch, and can seamlessly engage with the simulation using optimal interaction and visualization paradigm (with VR users being shown proxies for complete immersion). This proves that **heterogeneous devices can participate in a single collaborative session**, receiving the most effective tools for their hardware, without requiring any manual configuration.

While further validation in realistic classroom settings is necessary to fully assess scalability and the pedagogical effectiveness, the initial evaluation provided in this section confirms the system has the capability of supporting deployment across different hardware and network conditions - which are typically the norm in these real-world scenarios.

## 6.3 Detection Model Training and Evaluation

Up to this point, all meaningful interaction has relied solely on virtual elements. In the previous demonstration, a generic black-and-white fiducial marker was used to provide visual cues of the virtual object's positioning and functioning. Its tracking was handled by the Magic Leap's built-in marker detection capabilities, with no involvement from the implemented Python detection server. Although using generic markers represents a valid approach - and demonstrates the platform's compatibility with alternative detection protocols - they carry no real **pedagogical meaning**: the marker does not represent any concept within the learning scenario.

The pedagogical significance of MR lies in bridging the gap between the physical and virtual worlds, transforming abstract concepts into tangible, manipulable experiences. In the context of the solar energy module, this bridge is established through the use of physical solar panel replicas that students can handle, examine, and position while observing the way the virtual objects respond to these changes. This allows learners to interact with representations of the very devices they are studying, rather than with "meaningless" shapes.

To complete the module's intended functionality, a custom YOLO model must be trained to recognize the solar panel replicas. Subsection 6.3.1 presents the complete training process, from dataset preparation and annotation through the chosen training parameters, concluding with a critical analysis of model performance using standard computer vision metrics. Subsection 6.3.2 then demonstrates the integration of this trained model into the educational module, establishing communication between the Unity and Python components and evaluating the system in its finalized state. The evaluation methodology addresses three core requirements:

i) **Model Performance:** Assess the model's accuracy, precision, and inference speed, along with key aspects of the training process such as convergence time and efficiency, ensuring the approach remains practical for deployment in educational settings.

ii) **Server Communication:** Verify if the Python server remains accessible even when operating behind firewalls or across different network configurations, validating the tunneling mechanism established with Ngrok.

iii) **Tracking Validation:** Test the complete detection pipeline, confirming whether virtual objects are correctly spawned, tracked, and synchronized with the physical replicas throughout the interaction.

### 6.3.1 Model Training and Performance Analysis

To serve as the detection target for this module, a printed solar panel image was used. While these paper-based markers are simplified for demonstration purposes, the same workflow can readily extend to more sophisticated physical objects - such as miniature photovoltaic modules in this particular case -, depending on the desired level of realism and educational context.

Training a detection model requires providing examples of *what* it should identify. This requires the creation of a dataset of images that captures the target object across different conditions. These variations allow the model to learn to recognize the object regardless of variations in lighting, orientation, background, and viewing angle - conditions that inevitably occur during AR use.

For this validation, a dataset of 37 images was collected. While relatively small, it captures sufficient variation for the model to learn the solar panel replicas' distinctive visual characteristics, especially given its simplicity. In addition, using a pre-trained model as base, makes this dataset sufficient for a proof-of-concept implementation.

Following collection, manual annotation was performed using Label Studio. As discussed in Section 5.1, this process involves defining both the object class ('Solar Panel') and the precise pixel boundaries it occupies within each image through segmentation masks. Figure 6.9 illustrates this preparation.

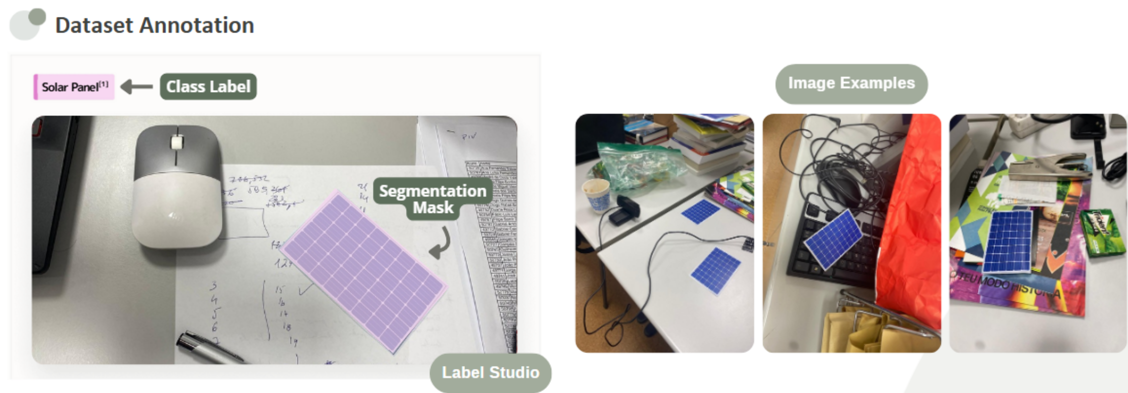



Figure 6.9: *Annotation process using Label Studio, showing segmentation mask definition for the solar panel replica.*

Following annotation, the dataset was split into two subsets: 30 images are used for training and 7 for validation (an approximately 80/20 split). The first set is fed to the model during training, while the second enables the assessment of the model's performance on unseen data. Label Studio generates a configuration file defining dataset paths and class labels, serving as the "instruction manual" for training.

Moving onto model training, `yolo11n-seg.pt` is used as a pre-trained base to leverage transfer learning benefits (refer to Section 5.1). The training script used is shown in Listing 6.1.

```
1 model = YOLO("yolo11n-seg.pt")
2 model.train(data="model_configuration.yaml", imgsz=640,
3             batch=8, epochs=100, workers=0, device="cpu")
```

Listing 6.1: YOLO model training script used for testing.

In this case, the choice of model hyperparameters balances dataset characteristics with computational constraints. Taking into consideration the small dataset, the 100-epoch limit

prevents overfitting: a situation where the model simply memorizes the training images and loses the ability to generalize to unseen ones. Furthermore, CPU-based training accommodates hardware limitations, though GPU acceleration would significantly reduce processing time.

The model training **successfully completed 100 epochs over approximately 12 minutes on CPU hardware**, and exhibited rapid convergence with the most significant improvements occurring in the initial iterations - this suggests a even lower epoch limit could have been used. Despite the small dataset, the model demonstrates strong performance metrics suitable for real-time deployment in the educational module.

Table 6.2 summarizes the key evaluation metrics for both detection and segmentation tasks on the validation set.

Table 6.2: Performance metrics of the trained model.

| Metric | Value | Interpretation |
|---|---|---|
| Box Detection mAP | 99.5% | Near-perfect object localization |
| Segmentation mAP | 99.5% | Near-perfect pixel-level segmentation |
| Precision | 99.4% | Very few false detections |
| Recall | 100% | All solar panels correctly detected |
| Training Time | 11.99 min | Efficient considering CPU hardware |

The model demonstrates remarkable performance across all evaluated metrics. Both box and segmentation Mean Average Precision (mAP) scores reach 99.5% - put simply, this means the model can localize solar panels and delineate their exact shapes at the pixel level accurately. A perfect recall of 100% shows that no solar panel instances were missed, while precision at 99.4% confirms that false positives are extremely minimal - the impact of these misclassifications will be analyzed later in this subsection.

Complementing these quantitative results, Figure 6.10 illustrates examples from the model's output, highlighting the accuracy of the pixel-level masks and confirming that even in cases where multiple instances appear within the same image, all solar panel replicas are correctly detected.

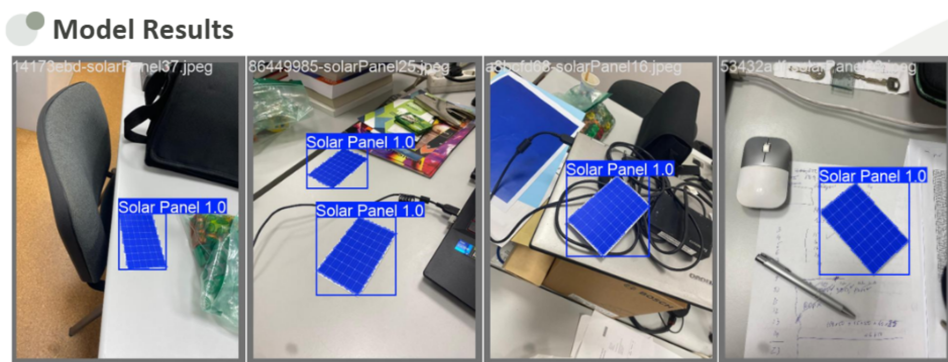



Figure 6.10: *Examples from four validation images.*

The final model file has approximately 5.89 MB, making it suitable for deployment across different hardware configurations without memory concerns. Although resource constraints prevented GPU testing, such configurations would likely enable faster training while maintaining the same efficiency benefits. CPU-based training nonetheless confirmed the approach's computational efficiency and broad compatibility.

To analyze these results from an educational stand-point, the near-perfect accuracy ensures that students can interact with the system without being distracted by inconsistent or unreliable detections, allowing the focus to remain on the underlying educational concepts. Furthermore, the lightweight nature of the model means it can run effectively on standard classroom hardware without requiring specialized equipment, supporting accessibility and scalability across different learning environments. Ensuring detection remains accurate under varied lighting and backgrounds enables the activities to be conducted in a wide range of settings, from bright laboratories to less controlled environments.

Beyond these performance metrics, it is also important to consider some specific practical scenarios that may arise in real classroom environments:

- **Visual Similarity:** Objects with similar colors or shapes may occasionally resemble the target objects, potentially causing misclassifications. This is an especially likely event when using simple markers with limited distinctive features - in this test's case, any blue and white objects could lead to a false detection.
- **Marker Multiplicity:** Multiple replicas of the solar panel may appear simultaneously in the scene. With the model being able to detect multiple instances of the same object within an image, it raises questions about how the system should interpret and track each instance.

Figure 6.11 illustrates these scenarios:

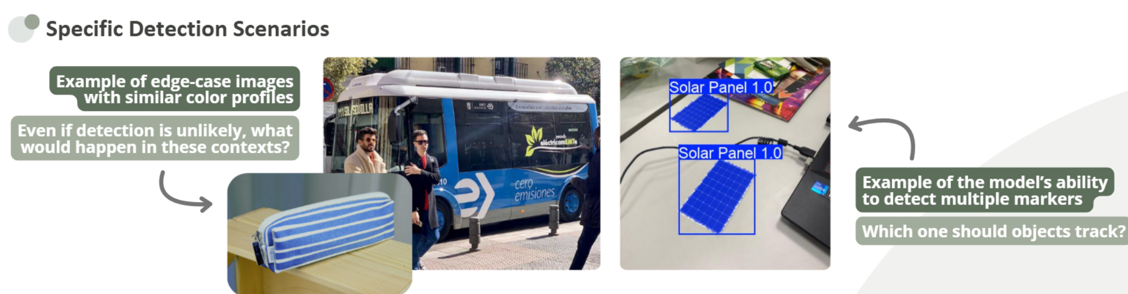



Figure 6.11: *Example of color similarity challenges (left) and proof of multiple marker detection (right).*

To further comment on each case, the continuous nature of the detection pipeline plays a key role in mitigating occasional misclassifications. Consider how the system operates in real time and continuously: interactions rely on sustained observation rather than a single frame. In practice, this means that temporary false detections in individual frames do not permanently disrupt overall tracking: object continuity is preserved across successive frames, ensuring stable and reliable interaction.

Regarding multiple markers, the desired behavior ultimately depends on the design of each educational module. In the context of the solar energy module, multiple panels could be interpreted in different ways: the system might track only the panel closest to the user as the reference point for interaction, or it could integrate multiple panels simultaneously, expanding the area of incidence and dynamically influencing energy calculations. Although this feature is not currently implemented, considering such possibilities highlights how module-specific logic can enrich the learning experience by implementing interaction dynamics between the physical objects and leverage the flexibility of the platform.

Overall, this training example demonstrates the accessibility and ease of setup of the proposed approach. The entire pipeline - from data collection through model deployment - requires minimal technical expertise and standard hardware, yet is proven to achieve exceptional performance metrics. Of note, while relatively simple markers were used, the methodology scales to more complex physical objects without fundamentally altering the training process - in other words, more complex detections do not necessarily translate into more complex training processes, as the exact same workflow still applies.

### 6.3.2 Module Integration and Tracking Validation

With the model trained and validated, the final consideration becomes its integration into the educational module. This subsection briefly describes the necessary configuration and examines the model's performance during practical deployment.

From the Unity side, the necessary setup is minimal. First, `Detector` and `DetectionManager` components are added to the rigs pertaining to devices capable of object detection, making them capable of making detection requests and treating the responses (Section 5.5). The module scene is then extended to include a `NetworkObjectManager` object (Section 5.4), using a configuration that maps the 'Solar Panel' labeled detections to the prefab created for this learning module - this step ensures that when a client detects the presence of a solar panel replica in their physical environment, the virtual Sun and clouds are aligned atop of it. After deploying the Python server with the trained model and ensuring Unity has the endpoint provided by Ngrok, the simulation can be launched and tested.

Figure 6.12 illustrates some of the reference setup and showcases what happens when a replica is detected in the user's environment.

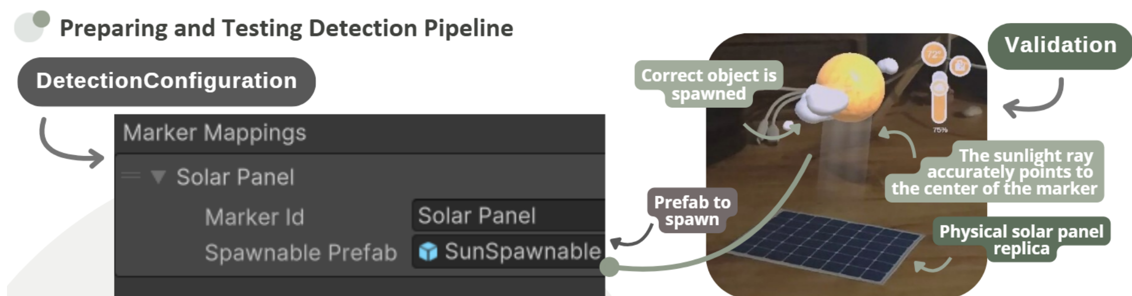



Figure 6.12: *When a solar panel replica is detected in the user's physical environment, the virtual interactive elements are automatically spawned and aligned with the marker position.*

This final evaluation verifies that the solar panels are reliably detected, spawn the intended virtual objects and maintain accurate tracking, embedding meaning into the collaborative simulation and completing the pedagogical cycle envisioned by the module.

Furthermore, testing the system while connecting through institutional firewalls and mobile hotspots demonstrates the correct functioning of the Ngrok tunneling mechanism, as access to the detection service remained consistent under all conditions. This confirms the approach's viability for diverse classroom environments, where network restrictions are often present.

The deployed model **achieves an average inference time of 45–55 ms per detection request**. Even though the client still has to perform additional post-processing to spawn the corresponding virtual objects - a step that varies slightly across devices - **the total time from initiating a detection request to the object appearing in the scene remains below 100 ms under typical circumstances**. This ensures that marker detection and virtual object spawning appear instantaneous to users, comfortably beneath the perceptual threshold for continuous interaction.

The introduction of object detection also does not compromise the real-time synchronization performance established in the previous multiplayer validation, with the propagation of updates across clients continuing to occur in under 50 ms. This demonstrates that the detection pipeline operates independently from the core networking architecture, preserving a seamless collaborative experience and preventing bottlenecks.

Most importantly, this completed system successfully bridges the gap between abstract concepts and meaningful tangible interaction. Students can engage with the physical solar panel replica, understanding contextual meaning, while simultaneously observing how its manipulation affects the virtual simulation. This dual-layer interaction — physical experimentation combined with virtual feedback — represents the educational value proposition that motivated this dissertation.

The evaluation across all three testing phases confirms that the proposed architecture addresses the fundamental challenges of cross-device compatibility, real-time synchronization, and physical-digital integration. The modular design enables efficient development and deployment of educational content, while the robust networking foundation supports scalable collaborative learning experiences.

While future work intentions will be discussed in the following chapter, it is worth emphasizing the importance of extending testing to the actual target user group: students. Future implementations should integrate the platform into solar energy education, leveraging both inquiry-based and collaborative learning models. This enables further assessment of the platform's actual educational impact, ensuring that immersive, mixed-reality experiences translate into meaningful learning gains across diverse educational levels.

# 7 Conclusions and Future Work

Mixed Reality offers unprecedented opportunities to blend the physical and digital worlds in learning environments, yet this potential has often remained unrealized due to interoperability barriers: the technological fragmentation that hinders truly meaningful collaborative experiences. This dissertation addressed this challenge by designing and implementing a modular client–server architecture capable of enabling cross-device collaboration, real-time synchronization, and seamless integration of physical and digital objects within shared immersive environments.

The system effectively creates the illusion of a shared space, allowing users to collaborate as if occupying the same room, regardless of their device type or actual physical location. This interoperability preserves the strengths of both AR and VR without compromising capabilities: AR participants manipulate objects through gestures and spatial mapping, while VR participants interact with the same virtual content via controllers. All users perceive and co-exist within the same coherent environment.

The contributions of this work are both technical and pedagogical. On the technical side, a dual-server architecture separates experience management from object detection: Unity coordinates multiplayer interactions, while a dedicated Python server performs optimized marker detection. This design ensures scalability, performance, and the ability to integrate new devices or educational modules without system-wide rebuilds. By employing abstractions, fine-tuned communication protocols, and generalized interaction models, heterogeneous devices can share a common experience, offering a practical solution to the fragmentation of MR platforms.

From an educational perspective, the platform demonstrates that abstract theoretical concepts can be transformed into tangible, interactive experiences. Learners can manipulate and observe dynamic phenomena, moving beyond static textbook representations. This fosters engagement, experimentation, and deeper conceptual understanding. Validation through a solar energy module confirms that networking, synchronization, and detection operate within perceptual thresholds necessary for fluid collaboration, providing evidence of the platform's practical viability.

Reflecting on the work as a whole, this dissertation not only achieved its stated objectives but also establishes a foundation for understanding how immersive technologies can truly serve education. The open-source, adaptable design encourages continued exploration and the creation of customized educational experiences across diverse contexts, even with minimal technical knowledge.

## 7.1 Future Work

While this work provides a solid foundation, numerous opportunities remain to enhance both pedagogical value and technical robustness:

- **Pedagogical Assessment:** As previously mentioned, the rigorous evaluation of the platform's effectiveness in real educational contexts is a critical next step. Studies should be made across varying educational levels and STEM subjects, comparing traditional teaching methods with the proposed Mixed Reality experiences. These evaluations should analyze learning outcomes, overall student engagement, and knowledge retention, providing clear evidence of the platform's pedagogical potential.
- **Platform Scalability:** On the same vein, testing the platform with larger user groups is necessary to truly validate its viability in real-world classroom deployments. These studies may also seek to incorporate a broader range of devices, creating increasingly complex multi-platform scenarios to fully demonstrate the system's adaptability.
- **Lobby Extensions:** In order to improve user experience, the overall lobby functionality could benefit from several interface and usability improvements. These may include easier navigation, enhanced communication tools such as chat or voice channels, and a dynamic notification system to inform users of ongoing sessions or collaborative activities. Additional organizational features, such as filtering or sorting available learning modules, could further streamline user access and engagement.
- **Personalization and Gamification:** Personalization can also be approached at user-level, creating features that allow learners to save their progress, customize their own profiles and learning spaces, and even create their own avatars. These features could deeply strengthen user engagement and motivation, making each user's experience truly their own. By incorporating mechanics such as achievement tracking or challenges, the platform could also provide a more rewarding learning experience.
- **Adaptive Learning:** Furthermore, by saving these personalized user preferences, it becomes possible to enable real-time adaptation of learning modules based on learner behavior. Predictive models could suggest tailored exercises, identify areas where students struggle, and provide feedback to instructors, supporting personalized learning paths and fostering deeper engagement.

Beyond this, the very expansion of educational modules is, by design, an ongoing and continuous process. The platform's goal is to provide an infrastructure that allows new topics, challenges, and pedagogical approaches to be added indefinitely. This ensures that the platform remains relevant and open to the creativity of both educators and learners, allowing learning to grow and adapt organically over time.

# Bibliography


[1] S. G. Weinbaum. "Pygmalion's Spectacles". In: *Wonder Stories* (June 1935) (cit. on p. 1).

[2] A. C. Clarke. *The City and the Stars*. Harcourt Brace, 1956. ISBN: 978-0-15-617803-8 (cit. on p. 1).

[3] Grand View Research. *Virtual Reality in Healthcare Market Size, Share and Trends Analysis Report*. 2024 (cit. on p. 1).

[4] A. A. Mazhar and M. M. A. Rifaee. "A Systematic Review of the use of Virtual Reality in Education". In: *2023 International Conference on Information Technology (ICIT)*. 2023, pp. 422–427. DOI: 10.1109/ICIT58056.2023.10225794 (cit. on p. 2).

[5] M. Alkhattabi. "Augmented and Virtual Reality in U.S. Education: A Review". In: *International Journal of Educational Research* (Apr. 2024). DOI: 10.13140/RG.2.2.15768.93441 (cit. on p. 2).

[6] Market.us. *AR and VR in Education Market Size / CAGR of 20.2%*. Market Research Report. Over 60% of US colleges expected to offer VR-based courses by 2024, up from 30% in 2022. Apr. 2024. URL: https://market.us/report/augmented-and-virtual-reality-in-education-market/ (cit. on p. 2).

[7] M. C. Johnson-Glenberg et al. "Embodied mixed reality with passive haptics in STEM education". In: *Frontiers in Virtual Reality* 4 (2023), p. 1047833. DOI: 10.3389/FRVIR.2023.1047833 (cit. on p. 2).

[8] T. Bezmalinovic. *Virtual reality is facing an old threat again: fragmentation*. https://mixed-news.com/en/vr-industry-fragmentation-threat/. Mar. 2024 (cit. on p. 2).

[9] M. L. Heilig. "The Cinema of the Future". In: (1955). Describes a multi-sensory theater vision, later implemented in the Sensorama (1962). URL: https://gametechdms.wordpress.com/wp-content/uploads/2014/08/w6_thecinemaoffuture_morton.pdf (cit. on p. 7).

[10] I. E. Sutherland. "A head-mounted three dimensional display". In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. New York, NY, USA: ACM, 1968, pp. 757–764. DOI: 10.1145/1476589.1476686 (cit. on p. 7).

[11] L. J. Prinzel et al. *Head-Up Displays and Attention Capture*. Tech. rep. NASA/TM-2004-213000. Hampton, VA: NASA Langley Research Center, 2004, pp. 1–2. URL: https://ntrs.nasa.gov/citations/20040065771 (cit. on p. 8).

[12] R. T. Azuma. "A Survey of Augmented Reality". In: vol. 6. 4. Aug. 1997, pp. 355–385. DOI: 10.1162/pres.1997.6.4.355 (cit. on p. 8).

[13] National Research Council. *Funding a Revolution: Government Support for Computing Research.* Chapter 10: Virtual Reality Comes of Age. Washington, DC: National Academies Press, 1999. Chap. 10, pp. 226–274. ISBN: 0-309-06278-0. URL: https://nap.nationalacademies.org/read/6323/chapter/12 (cit. on p. 8).

[14] Z. Chen, J. Li, Y. Liu, et al. "Augmented reality and virtual reality displays: emerging technologies and future perspectives". In: *Light: Science & Applications* 10.216 (2021). DOI: 10.1038/s41377-021-00658-8 (cit. on p. 9).

[15] M. Billinghurst, A. Clark, and G. Lee. "A Survey of Augmented Reality". In: *Foundations and Trends in Human-Computer Interaction* 8.2-3 (2015), pp. 73–272. DOI: 10.1561/1100000049 (cit. on p. 9).

[16] J. J. LaViola Jr., E. Kruijff, R. P. McMahan, D. Bowman, and I. P. Poupyrev. "3D User Interfaces: Theory and Practice". In: *Addison-Wesley Professional.* Addison-Wesley, 2017. ISBN: 978-0-321-98037-7 (cit. on p. 9).

[17] P. Luckey. *Oculus Rift: Step Into the Game.* Kickstarter Campaign. 2012. URL: https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game (cit. on p. 9).

[18] Niantic, Inc. *Pokémon GO.* Mobile Application. 2016. URL: https://pokemongolive.com/ (cit. on p. 9).

[19] T. Howe et al. "Influence of Pokémon Go on Physical Activity: Study and Implications". In: *American Journal of Preventive Medicine* 52.6 (2016). Study of 25 million US users showing 144 billion additional steps in first 30 days, pp. 827–829. DOI: 10.1016/j.amepre.2016.12.012 (cit. on p. 9).

[20] S.-C. Yang, S.-H. Liu, Y.-C. Su, and C.-P. Lin. "An adoption framework for mobile augmented reality games: The case of Pokémon Go". In: *Computers in Human Behavior* 76 (2017), pp. 267–276. DOI: 10.1016/j.chb.2017.07.030 (cit. on p. 9).

[21] S. Ibáñez-Sánchez, M. Flavián, C. Orús, and S. Belanche. "Augmented reality filters on social media. Analyzing the drivers of playability based on uses and gratifications theory". In: *Psychology & Marketing* 39.7 (2022), pp. 1352–1370. DOI: 10.1002/mar.21639 (cit. on p. 9).

[22] The Motley Fool. *Meta Platforms Has Spent $46 Billion on the Metaverse Since 2021.* The Motley Fool. Apr. 2024. URL: https://www.fool.com/investing/2024/04/01/meta-platforms-has-spent-46-billion-on-the-metaver/ (cit. on p. 9).

[23] Google. *Google Maps adds augmented reality navigation with Live View.* TechRadar. Aug. 2019. URL: https://www.techradar.com/news/google-maps-augmented-reality-apple-maps (cit. on p. 9).

[24] HealthTech Magazine. *How AR & VR in Healthcare Enhances Medical Training.* HealthTech Magazine. June 2023. URL: https://healthtechmagazine.net/article/2022/12/ar-vr-medical-training-2023-perfcon (cit. on p. 9).

[25] J. Jerald. "Latency Requirements for Head Mounted Displays: Avoiding Motion Sickness through Low Latency VR". In: *Proceedings of IEEE Virtual Reality* (2002), pp. 67–72. DOI: 10.1109/VR.2002.996519 (cit. on p. 10).

[26] Meta. *Meta Quest – MR, VR Headsets Accessories.* 2025. URL: https://www.meta.com/quest/ (cit. on p. 10).

[27] H. Corporation. *HTC VIVE – VR, AR, and MR Headsets, Glasses, Experiences.* 2025. URL: https://www.vive.com/ (cit. on p. 10).

[28] V. Corporation. *Valve Index® – Upgrade Your Experience.* 2025. URL: https://www.valvesoftware.com/index (cit. on p. 10).

[29] Microsoft. *Microsoft HoloLens – Mixed Reality Devices and Documentation.* 2025. URL: https://learn.microsoft.com/en-us/hololens/ (cit. on p. 10).

[30] I. Magic Leap. *Magic Leap – Augmented Reality Solutions.* 2025. URL: https://www.magicleap.com/ (cit. on p. 10).

[31] A. Arshad et al. "WebXR Lighting Estimation and Physically Based Rendering for Realistic Augmented Reality". In: *Journal of Imaging* 9.3 (2023), p. 63. DOI: 10.3390/jimaging9030063 (cit. on p. 10).

[32] P. Milgram, H. Takemura, A. Utsumi, and F. Kishino. "Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum". In: *Telemanipulator and Telepresence Technologies (SPIE Proceedings, Vol. 2351).* 1994, pp. 282–292. DOI: 10.1117/12.197321 (cit. on p. 11).

[33] Apple. *Introducing Apple Vision Pro: Apple's first spatial computer.* 2023. URL: https://www.apple.com/newsroom/2023/06/introducing-apple-vision-pro/ (cit. on p. 13).

[34] Varjo. *Varjo Aero: highest-fidelity prosumer virtual reality headset.* 2021. URL: https://varjo.com/press-release/varjo-introduces-varjo-aero-headset-to-bring-the-highest-fidelity-virtual-reality-for-professionals-and-leading-edge-vr-users-alike/ (cit. on p. 13).

[35] Nreal. *Nreal Air: lightweight augmented reality glasses.* 2022. URL: https://vr-compare.com/headset/nrealair (cit. on p. 14).

[36] Rokid. *Rokid Max: next-generation AR smart glasses.* 2023. URL: https://www.global.rokid.com/products/rokid-max (cit. on p. 14).

[37] V. Schwind, K. Reinhardt, R. Rzayev, N. Henze, and K. Wolf. "Virtual reality on the go: a study on social acceptance of VR glasses". In: *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct.* MobileHCI '18. New York, NY, USA: ACM, 2018, pp. 111–118. DOI: 10.1145/3236112.3236127 (cit. on p. 14).

[38] Apple. *ARKit introduced at WWDC 2017*. 2017. URL: https://apple.fandom.com/wiki/ARKit (cit. on p. 14).

[39] Google. *ARCore: Google's augmented reality SDK*. 2018. URL: https://developers.google.com/ar (cit. on p. 14).

[40] TechInsights. *iPhone 15 Pro Max Rear LiDAR Camera Process Flow Analysis*. Technical Analysis. Analysis of Sony IMX591 LiDAR SPAD sensor with photocathode design innovations. 2023. URL: https://www.techinsights.com/blog/iphone-15-pro-max-rear-lidar-camera-process-flow-analysis (cit. on p. 14).

[41] M. Akçayır and G. Akçayır. "Advantages and challenges associated with augmented reality for education: A systematic review of the literature". In: *Educational Research Review* 20 (2017), pp. 1–11. DOI: 10.1016/J.EDUREV.2016.11.002 (cit. on p. 15).

[42] A. Banjar, X. Xu, M. Z. Iqbal, and A. Campbell. "A systematic review of the experimental studies on the effectiveness of mixed reality in higher education between 2017 and 2021". In: *Computers & Education: X Reality* 3 (2023), p. 100034. DOI: 10.1016/J.CEXR.2023.100034 (cit. on p. 15).

[43] Nanome Inc. *Nanome: Virtual Reality for Drug Design and Molecular Visualization*. Software Platform. 2024. URL: https://nanome.ai/ (cit. on p. 15).

[44] Labster ApS. *Labster Virtual Labs for Universities and High Schools*. Educational Platform. 2024. URL: https://www.labster.com/ (cit. on p. 15).

[45] University of Colorado Boulder. *PhET Interactive Simulations*. Educational Resource. 2024. URL: https://phet.colorado.edu/ (cit. on p. 15).

[46] zSpace Inc. *zSpace Mixed Reality Learning Platform*. Educational Technology Platform. 2024. URL: https://zspace.com/ (cit. on p. 15).

[47] International GeoGebra Institute. *GeoGebra Augmented Reality*. Mathematical Software. 2024. URL: https://www.geogebra.org/ar (cit. on p. 15).

[48] MEL Science. *MEL Science Virtual Reality Chemistry*. Educational Platform. 2024. URL: https://melscience.com/ (cit. on p. 15).

[49] Victory XR. *VictoryXR Campus*. Virtual Reality Educational Platform. 2024. URL: https://victoryxr.com/ (cit. on p. 15).

[50] TimeLooper. *TimeLooper Virtual Reality Experiences*. VR Experience Platform. 2024. URL: https://timelooper.com/ (cit. on p. 15).

[51] N. T. Lee and R. Ray. "Ensuring equitable access to AR/VR in higher education". In: *Brookings TechTank* (Sept. 2022). URL: https://www.brookings.edu/articles/ensuring-equitable-access-to-ar-vr-in-higher-education/ (cit. on p. 16).

[52] The Khronos Group Inc. *OpenXR Specification*. Khronos Registry. Version 1.1.50. 2024. URL: https://registry.khronos.org/OpenXR/specs/1.1/html/xrspec.html (cit. on p. 17).

[53] W3C Immersive Web Working Group. *WebXR Device API*. W3C Recommendation. Jan. 2023. URL: https://www.w3.org/TR/webxr/ (cit. on p. 17).

[54] G. Bradski and A. Kaehler. *OpenCV: Open Source Computer Vision Library*. 2000. URL: https://opencv.org/ (cit. on pp. 18, 29).

[55] E. Olson. *AprilTag: A Robust and Flexible Visual Fiducial System*. https://github.com/AprilRobotics/apriltag. Official GitHub repository and documentation. 2011 (cit. on p. 18).

[56] M. Kalaitzakis, B. Cain, S. Carroll, A. Ambrosia, C. Whitehead, and N. Vitzilaios. "Fiducial Markers for Pose Estimation: Overview, Applications and Experimental Comparison of the ARTag, AprilTag, ArUco and STag Markers". In: *Journal of Intelligent & Robotic Systems* 101.4 (2021), p. 71. DOI: 10.1007/s10846-020-01307-9 (cit. on p. 18).

[57] Unity Technologies. *Unity Technologies*. Unity Package Documentation. Version 2.4.4. 2024. URL: https://unity.com/ (cit. on p. 23).

[58] Unity Technologies. *Unity Gaming Services*. Unity Documentation. 2024. URL: https://docs.unity.com/ugs/ (cit. on p. 23).

[59] Unity Technologies. *Netcode for GameObjects*. Unity Package Documentation. Version 2.4.4. 2024. URL: https://docs.unity3d.com/6000.1/Documentation/Manual/com.unity.netcode.gameobjects.html (cit. on p. 23).

[60] Aymeric Augustin. *websockets: Library for building WebSocket servers and clients in Python*. Python Package Documentation. Version 15.0.1. Library for building WebSocket servers and clients in Python with focus on correctness, simplicity, robustness, and performance using asyncio. 2024. URL: https://websockets.readthedocs.io/ (cit. on p. 23).

[61] A. Laird. *pyngrok: A Python wrapper for ngrok*. Python Package. Version 7.3.0. Python wrapper for ngrok that manages its own binary, enabling secure tunnels from public URLs to localhost. 2024. URL: https://pyngrok.readthedocs.io/ (cit. on p. 23).

[62] AltexSoft. *The Good and the Bad of C# Programming*. Technical Analysis. Oct. 2021. URL: https://www.altexsoft.com/blog/c-sharp-pros-and-cons/ (cit. on p. 24).

[63] Heartex. *Label Studio: Open Source Data Labeling Tool*. 2020. URL: https://labelstud.io/ (cit. on p. 27).

[64] P. Engine. *Photon Engine – Multiplayer Game Development Made Easy*. 2025. URL: https://www.photonengine.com/ (cit. on p. 40).

[65] M. Networking. *Mirror Networking – Open Source Networking for Unity*. 2025. URL: https://mirror-networking.com/ (cit. on p. 40).

[66] S. Yogamani, C. Eising, J. Horgan, G. Sistu, P. Varley, D. O'dea, M. Uřičář, S. Milz, M. Simon, K. Amende, C. Witt, H. Rashed, S. Chennupati, S. Nayak, S. Mansoor, X. Perrotton, and P. Pérez. *WoodScape: A Multi-Task, Multi-Camera Fisheye Dataset for Autonomous Driving*. Nov. 2019. DOI: 10.1109/ICCV.2019.00940 (cit. on p. 70).

[67] A. Smets, K. Jäger, O. Isabella, R. van Swaaij, and M. Zeman. *Solar Energy: The physics and engineering of photovoltaic conversion, technologies and systems.* UIT Cambridge Limited, 2016 (cit. on p. 73).

[68] M. Tanrioven. *Photovoltaic Systems Engineering for Students and Professionals: Solved Examples and Applications.* 1st. CRC Press, 2023. DOI: 10.1201/9781003415572. URL: https://doi.org/10.1201/9781003415572 (cit. on p. 73).

[69] R. Albert, A. Patney, D. Luebke, and J. Kim. *Latency Requirements for Foveated Rendering in Virtual Reality.* ACM Transactions on Applied Perception, Vol. 14, No. 4, Article 25. Sept. 2017. URL: https://doi.org/10.1145/3127589 (cit. on p. 81).

# A Depth Undistortion Utility Class for Magic Leap 2

The class included in this Appendix implements the undistortion algorithms mentioned in Section 5.5.2.3, following the Magic Leap 2 device documentation.

```
using System;
using UnityEngine;
using Unity.Collections;
using System.Runtime.InteropServices;
using MagicLeap.OpenXR.Features.PixelSensors;

public static class DepthUndistortion
{
    /// <summary>
    /// Undistorts a depth image using pinhole camera intrinsics
    /// </summary>
    /// <param name="distortedDepth">Input distorted depth data</param>
    /// <param name="width">Image width</param>
    /// <param name="height">Image height</param>
    /// <param name="intrinsics">Camera intrinsics with distortion
    ↪ coefficients</param>
    /// <returns>Undistorted depth data</returns>
    public static byte[] UndistortDepthImage(byte[] distortedDepth, int width,
    ↪ int height, PixelSensorPinholeIntrinsics intrinsics)
    {
        // Create output array
        byte[] undistortedDepth = new byte[distortedDepth.Length];

        // Convert to float spans for processing
        ReadOnlySpan<byte> inputByteSpan = distortedDepth.AsSpan();
        ReadOnlySpan<float> inputFloatSpan = MemoryMarshal.Cast<byte,
        ↪ float>(inputByteSpan);

        Span<byte> outputByteSpan = undistortedDepth.AsSpan();
        Span<float> outputFloatSpan = MemoryMarshal.Cast<byte,
        ↪ float>(outputByteSpan);
```

```
28
29          // Extract intrinsic parameters
30          float fx = intrinsics.FocalLength.x;
31          float fy = intrinsics.FocalLength.y;
32          float cx = intrinsics.PrincipalPoint.x;
33          float cy = intrinsics.PrincipalPoint.y;
34
35          double k1 = intrinsics.Distortion[0];
36          double k2 = intrinsics.Distortion[1];
37          double p1 = intrinsics.Distortion[2];
38          double p2 = intrinsics.Distortion[3];
39          double k3 = intrinsics.Distortion[4];
40
41          // Process each pixel in the undistorted image
42          for (int y = 0; y < height; y++)
43          {
44              for (int x = 0; x < width; x++)
45              {
46                  // Convert pixel coordinates to normalized coordinates
47                  double xn = (x - cx) / fx;
48                  double yn = (y - cy) / fy;
49
50                  // Apply distortion model to find corresponding distorted
                    ↪  coordinates
51                  Vector2 distortedCoords = ApplyDistortion(xn, yn, k1, k2, p1, p2,
                    ↪  k3);
52
53                  // Convert back to pixel coordinates
54                  float distortedX = (float)(distortedCoords.x * fx + cx);
55                  float distortedY = (float)(distortedCoords.y * fy + cy);
56
57                  // Bilinear interpolation to get depth value
58                  float depth = BilinearInterpolate(inputFloatSpan, width, height,
                    ↪  distortedX, distortedY);
59
60                  // Store in output
61                  int outputIndex = y * width + x;
62                  outputFloatSpan[outputIndex] = depth;
63              }
64          }
65
66          return undistortedDepth;
67      }
68
69      /// <summary>
70      /// Applies the distortion model to normalized coordinates
71      /// </summary>
```

```
72      private static Vector2 ApplyDistortion(double xn, double yn, double k1,
        ↪  double k2, double p1, double p2, double k3)
73      {
74          double r2 = xn * xn + yn * yn;
75          double r4 = r2 * r2;
76          double r6 = r4 * r2;
77
78          // Radial distortion
79          double radialDistortion = 1 + k1 * r2 + k2 * r4 + k3 * r6;
80
81          // Tangential distortion
82          double tangentialX = 2 * p1 * xn * yn + p2 * (r2 + 2 * xn * xn);
83          double tangentialY = p1 * (r2 + 2 * yn * yn) + 2 * p2 * xn * yn;
84
85          // Apply distortion
86          double xd = xn * radialDistortion + tangentialX;
87          double yd = yn * radialDistortion + tangentialY;
88
89          return new Vector2((float)xd, (float)yd);
90      }
91
92      /// <summary>
93      /// Performs bilinear interpolation on the depth data
94      /// </summary>
95      private static float BilinearInterpolate(ReadOnlySpan<float> data, int width,
        ↪  int height, float x, float y)
96      {
97          // Handle boundary cases
98          if (x < 0 || x >= width - 1 || y < 0 || y >= height - 1)
99          {
100             return 0f; // or some default depth value
101         }
102
103         int x1 = (int)Math.Floor(x);
104         int y1 = (int)Math.Floor(y);
105         int x2 = x1 + 1;
106         int y2 = y1 + 1;
107
108         // Ensure we don't go out of bounds
109         x2 = Math.Min(x2, width - 1);
110         y2 = Math.Min(y2, height - 1);
111
112         float dx = x - x1;
113         float dy = y - y1;
114
115         // Get the four surrounding pixels
116         float q11 = data[y1 * width + x1];
117         float q12 = data[y2 * width + x1];
```

```
118            float q21 = data[y1 * width + x2];
119            float q22 = data[y2 * width + x2];
120
121            // Bilinear interpolation
122            float top = q11 * (1 - dx) + q21 * dx;
123            float bottom = q12 * (1 - dx) + q22 * dx;
124
125            return top * (1 - dy) + bottom * dy;
126        }
127
128        /// <summary>
129        /// Creates a lookup table for faster undistortion (recommended for real-time
           ↪  processing)
130        /// </summary>
131        public static void CreateUndistortionLookupTable(int width, int height,
           ↪  PixelSensorPinholeIntrinsics intrinsics,
132            out Vector2[] lookupTable)
133        {
134            lookupTable = new Vector2[width * height];
135
136            float fx = intrinsics.FocalLength.x;
137            float fy = intrinsics.FocalLength.y;
138            float cx = intrinsics.PrincipalPoint.x;
139            float cy = intrinsics.PrincipalPoint.y;
140
141            double k1 = intrinsics.Distortion[0];
142            double k2 = intrinsics.Distortion[1];
143            double p1 = intrinsics.Distortion[2];
144            double p2 = intrinsics.Distortion[3];
145            double k3 = intrinsics.Distortion[4];
146
147            for (int y = 0; y < height; y++)
148            {
149                for (int x = 0; x < width; x++)
150                {
151                    double xn = (x - cx) / fx;
152                    double yn = (y - cy) / fy;
153
154                    Vector2 distortedCoords = ApplyDistortion(xn, yn, k1, k2, p1, p2,
                       ↪  k3);
155
156                    float distortedX = (float)(distortedCoords.x * fx + cx);
157                    float distortedY = (float)(distortedCoords.y * fy + cy);
158
159                    lookupTable[y * width + x] = new Vector2(distortedX, distortedY);
160                }
161            }
162        }
```

```
163
164         /// <summary>
165         /// Fast undistortion using pre-computed lookup table
166         /// </summary>
167         public static byte[] UndistortDepthImageFast(byte[] distortedDepth, int
            ↪ width, int height, Vector2[] lookupTable)
168         {
169             byte[] undistortedDepth = new byte[distortedDepth.Length];
170
171             ReadOnlySpan<byte> inputByteSpan   = distortedDepth.AsSpan();
172             ReadOnlySpan<float> inputFloatSpan = MemoryMarshal.Cast<byte,
                ↪ float>(inputByteSpan);
173
174             Span<byte> outputByteSpan = undistortedDepth.AsSpan();
175             Span<float> outputFloatSpan = MemoryMarshal.Cast<byte,
                ↪ float>(outputByteSpan);
176
177             for (int y = 0; y < height; y++)
178             {
179                 for (int x = 0; x < width; x++)
180                 {
181                     Vector2 distortedCoords = lookupTable[y * width + x];
182                     float depth = BilinearInterpolate(inputFloatSpan, width, height,
                        ↪ distortedCoords.x, distortedCoords.y);
183                     outputFloatSpan[y * width + x] = depth;
184                 }
185             }
186
187             return undistortedDepth;
188         }
189     }
```

# B *SunInteractable* Class - Handling the Testing Module's Interaction Requirements

The class included in this Appendix supports the description provided in Section 6.1.2.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5  using DG.Tweening;
6  using Unity.Netcode;
7
8  [RequireComponent(typeof(LineRenderer))]
9  [RequireComponent(typeof(Rigidbody))]
10 [RequireComponent(typeof(MeshRenderer))]
11 public class SunInteractable : Interactable
12 {
13     private static float _panelArea  = 2f;     // m2
14     private static float _panelEffic = 0.18f; // 18%
15
16
17     private NetworkVariable<float> _angle = new NetworkVariable<float>(90f,
       ↪  NetworkVariableReadPermission.Everyone,
       ↪  NetworkVariableWritePermission.Server);
18     private NetworkVariable<float> _light = new NetworkVariable<float>(1.0f,
       ↪  NetworkVariableReadPermission.Everyone,
       ↪  NetworkVariableWritePermission.Server);
19     private NetworkVariable<float> _power = new NetworkVariable<float>(1.0f,
       ↪  NetworkVariableReadPermission.Everyone,
       ↪  NetworkVariableWritePermission.Server);
20
21
22     private LineRenderer _beam;
```

```
23      [SerializeField] private Text _angleText;
24      [SerializeField] private Text _lightText;
25      [SerializeField] private Text _powerText;
26      [SerializeField] private Slider _lightSlider;
27      [SerializeField] private Transform cloudParent;
28
29
30      // Inherited methods - component preparation and updating
31      public override void PrepareComponents()
32      {
33          _beam = GetComponent<LineRenderer>();
34          _beam.startWidth = 0.03f;
35          _beam.endWidth   = 0.05f;
36
37          _light.OnValueChanged += (oldValue, newValue) => {
38              _lightSlider.SetValueWithoutNotify(newValue);
39              _lightText.text = $"{Mathf.Round(newValue * 100)}%";
40              UpdateCloudVisibility();
41          };
42
43          _angle.OnValueChanged += (oldValue, newValue) => {
44              _angleText.text = $"{newValue}°";
45          };
46
47          _power.OnValueChanged += (oldValue, newValue) => {
48              _powerText.text = $"{Mathf.Round(newValue * 10f) / 10f}W";
49          };
50      }
51
52      public override void UpdateComponents()
53      {
54          if (IsServer) { GetIncidenceAngle();
55                          UpdateSolarPowerOutput(); }
56
57          UpdateBeam();
58      }
59
60      private void UpdateBeam() {
61          MarkerInfo panel = spawnable.GetMarkerInfo();
62
63          _beam.enabled = true;
64          _beam.SetPosition(0, transform.position);
65          _beam.SetPosition(1, panel.Pose.position);
66
67          _beam.endWidth = Mathf.Max(0.05f * (1f - _light.Value), 0.03f);
68
69          Color currentColor = _beam.startColor;
70          currentColor.a = 1f * (1f - _light.Value);
```

```
71            _beam.startColor = currentColor;
72        }
73
74        // Method to calculate power output
75        private void UpdateSolarPowerOutput() {
76            float dirNormIrr = 800f * Mathf.Exp(-3f * _light.Value);
77            float incFactor  = Mathf.Cos(_angle.Value * Mathf.Deg2Rad);
78
79            _power.Value = dirNormIrr * incFactor * _panelArea * _panelEffic;
80        }
81
82        // Methods relating to incidence angle
83        private void GetIncidenceAngle()
84        {
85            MarkerInfo panel      = spawnable.GetMarkerInfo();
86            Vector3 panelNormal  = panel.Pose.rotation * -Vector3.forward;
87            Vector3 sunDirection = (transform.position -
            ↪   panel.Pose.position).normalized;
88
89            _angle.Value = Mathf.Round(Vector3.Angle(panelNormal, sunDirection));
90        }
91
92        // Methods relating to light percentage / cloud visibility
93        public void SetLightPercentage(float value)
94        {
95            if (_light.Value == value) return;
96
97            UpdateLightServerRpc(value);
98        }
99
100       [ServerRpc(RequireOwnership = false)]
101       private void UpdateLightServerRpc(float value)
102       {
103           _light.Value = value;
104       }
105
106       private void UpdateCloudVisibility()
107       {
108           if (cloudParent == null) return;
109
110           int totalClouds     = cloudParent.childCount;
111           int cloudsToEnable = Mathf.RoundToInt(totalClouds * _light.Value);
112
113           for (int i = 0; i < totalClouds; i++)
114           {
115               GameObject cloud = cloudParent.GetChild(i).gameObject;
116               if (i < cloudsToEnable)
117               {
```

```
118                 if (!cloud.activeSelf)
119                 {
120                     Vector3 topPosition    = cloud.transform.localPosition;
121                     Vector3 bottomPosition = topPosition + new Vector3(0, -1.5f,
                        ↪  0);
122                     cloud.transform.localPosition = bottomPosition;
123                     cloud.SetActive(true);
124                     cloud.transform.DOLocalMove(topPosition, 0.5f);
125                 }
126             }
127             else
128             {
129                 if (!DOTween.IsTweening(cloud.transform) && cloud.activeSelf) {
130                     Vector3 topPosition    = cloud.transform.localPosition;
131                     Vector3 bottomPosition = topPosition + new Vector3(0, -1.5f,
                        ↪  0);
132
133                     cloud.transform.DOLocalMove(bottomPosition,
                        ↪  0.2f).OnComplete(() =>
134                     {
135                         cloud.SetActive(false);
136                         cloud.transform.localPosition = topPosition;
137                     });
138
139                 }
140             }
141         }
142     }
143 }
```