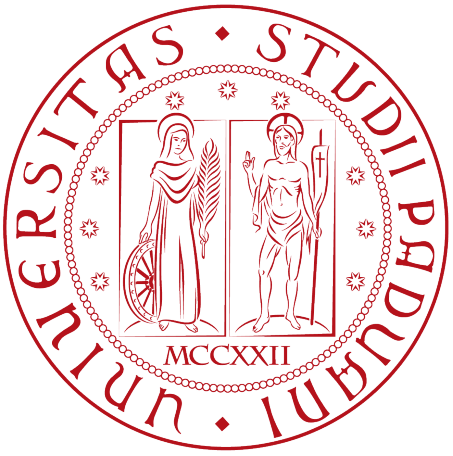
**University of Padua**

Department of Mathematics "Tullio Levi-Civita"

Master's Degree in Computer Science


# A Framework for Client-Server Objects Streaming in VR


Federico Carboni
2020286

*Supervisor*

Prof. Claudio Enrico Palazzi

*Co-Supervisors*

Prof. Dario Maggiorini, University of Milan
Prof. Davide Gadia, University of Milan



A.Y. 2021-2022

Federico Carboni: *A Framework for Client-Server Objects Streaming in VR,* , © December 2022.

# Abstract

In the wake of the technological improvements of the last decade, such as the development of the network infrastructure, and thanks to the need for connection caused by the lockdowns used to restrict the spread of the Covid-19 Pandemic, there has been an increased interest in VR and AR technologies. Many tech companies are investing in designing their version of "Metaverse": a multi-user virtual environment in which people can remotely interact and share experiences. All these prototypes have the same problem: before entering a room or world, the client device needs to download the data of the entire environment, this slows down the user experience and can clog up the memory of the local device. The aim of this thesis work is to develop a possible solution to overcome these problems, namely create a 3D Objects Streaming System, that shares and keeps synced with the Clients only the entities that they can currently see, perceive, and interact with.

*"I guess you guys aren't ready for that yet. But your kids are gonna love it."*

— Marty McFly

# Acknowledgments

In primis, ringrazio mio relatore Prof. Claudio Enrico Palazzi, per le indicazioni, i consigli e il supporto offerto durante la realizzazione del progetto e la stesura di questo documento.

Un doveroso ringraziamento va anche al Prof. Dario Maggiorini e al Prof. Davide Gadia, docenti dell'Università degli Studi di Milano, per le dritte e il confronto riguardo i temi della tesi.

Grazie di cuore ai miei genitori, che mi hanno sempre sostenuto e senza i quali non sarei potuto arrivare fin qui.

Grazie a tutti i colleghi che ho incontrato in questo percorso e con cui ho avuto l'onore di lavorare, a Chiara, Luca, Nicola e gli amici che sono sempre rimasti al mio fianco e con cui in questi anni ho condiviso le esperienze più importanti.

First of all, I thank my supervisor Prof. Claudio Enrico Palazzi, for the guidance, the advices, and the support offered during the project execution and during the writing of this document.

A dutiful thanks goes to Prof. Dario Maggiorini and Prof. Davide Gadia as well, teachers of the University of Milan, for the suggestions and for the exchanges on the topics of the thesis.

A heartfelt thanks to my parents who have always supported me and without whom I couldn't have gotten this far.

Thanks to all the colleagues that I have met along this journey and with whom I had the honor of working, to Chiara, Luca, Nicola, and the friends who always stayed by my side and with whom in these years I shared the most valuable experiences.

*Padova, December 2022* Federico Carboni

# Contents

# List of Figures

# List of Tables



# List of Listings

# Chapter 1

# Introduction

In recent years, the Virtual and Augmented Reality (VR/AR) Technologies have experienced a growing interest. These technologies themselves are not really a new invention; to mention some examples, Sega [1] announced the *Sega VR headset* back in 1991, and Google developed the *Google Glass* in 2012: an AR device that became famous as one of the biggest failures of the company, but that even nowadays inspires the future of wearable devices. An example of this are the Ray-Ban Stories [2] glasses, born by a partnership between Meta and EssilorLuxottica or the futuristic Mojo Lens [3] by Mojo Vision.

Now that VR headsets are becoming more affordable and accessible, this allows people to immerse themselves into virtual worlds without having to leave their homes. Additionally, the network infrastructure in many rich countries can now support gigabytes of data-stream: this opens the door for multiplayer AR/VR interactions, where people can meet and share experiences. Moreover, the Covid-19 pandemic has given a boost to all the technologies that offer some way to interact remotely. This includes video-games, VR, chats, video-calls and so on.

For these reasons, many big tech companies targeted VR and AR as the new frontier for the human-computer interactions, trying to build social platforms that can include multiple services, all available in a single place. A popular name with which these kinds of social platforms are called today is *Metaverse*.

Meta (previously known as Facebook) was one of the first companies that attempted this, initially in 2019, by building a series of quite affordable devices: the Meta (formerly named Oculus) Quest Headsets and then, in 2022, developing its own application, where people could meet and interact using their virtual reality devices, called Horizon Worlds. Meta is the leading company in the sector, but it is followed by many other big-tech companies, all trying to place themselves as the new VR standard. Some of the most important ones are: Google, especially in the AR field, with the Google Lens service; HTC with the HTC Vive, a series of high-end, mostly wired devices; Unity with the Unity Engine, used to build most VR and AR games and experiences; Microsoft with the Hololens mixed-reality devices.

## 1.1 Problems with Multiplayer VR Platforms

The current VR platforms suffers from some problems that hold back the progress of this means of communication, affecting the final user's enjoyment: first, the absence of an effective locomotive system that permits to walk or run in the virtual environment limits the freedom of the gameplay. There are some devices under development, such as the Omni One [4] or the Cybershoes [5], but they are still prototypes and there is not yet an established standard; secondly, the effects of latency or motion sickness in VR games can degrade the user experience more significantly than the computer or console counterpart, given the higher immersion.

Despite these open challenges, together with Meta's Horizon Worlds, many other alternative metaverse [6] applications that try to achieve the same shared experience, have been developed by different companies. Some of the systems most used by VR owners are: Horizon Worlds, Roblox, Decentraland, VR-Chat.

In particular, VR-Chat [7] is one of the most used between Oculus Quest owners; in this game you can explore the existing worlds and even upload your custom-made ones, created using the Unity Engine [8]. Once inside the game you can teleport from one room (world) to another using portals, and before entering, the room will be fully downloaded into the local memory of the device. This procedure is common to all existing VR social games, but it can cause tedious waiting times before the actual connection, and it can strongly limit the size of the environment and the amount of objects contained in each scene. This is particularly meaningful for stand-alone devices like Quest and Quest 2, which have relatively little storage memory (between 64 and 250 GB). Moreover, if not well optimized, the presence of many entities to be rendered on the scene can easily overload the mobile Graphic Processor in use. We focused on this last issue to develop the starting point for a possible solution.

## 1.2 A Possible Solution for scenes download: 3D Objects Streaming

One possible approach to solving the aforementioned problem is to displace the game computation and elaboration away from the final user's machine using *Cloud Gaming* [9]. This involves a Central Server that runs the game, receives the user inputs from the Client (for example, moving or jumping), and it streams the rendered images back. While this method has become more popular in the last couple of years, it is not very efficient, given that the streaming technology was invented to stream realistic images and not 3D models. This results in a possible compression of the game-rendered image that is sent to the final user to overcome the traffic congestion and in some inevitable delay between the player actions and the perceived movement. One better solution, instead, could be to send directly to the Clients the information it needs to reconstruct the 3D Models and their behaviors. The Client will require more power to render the objects received, but this method reduces the byte-stream size, and it avoids the delay between the user input and the movement or actions on the screen.

The solution explored in this thesis work consists of a real-time, objects, and entities streaming framework. Our framework is composed of two types of components: a *Server* that contains the game files and the *Clients*, which are initially empty and will dynamically populate the scene using the content received via network streaming from the Server. The framework's Server is able to assign a priority to each object of the

scene with respect to the connected VR Players, which is calculated on the basis of their position. The Server is also responsible for the serialization of the object's data that need to be progressively streamed to the Client's device. This allows the Client device to store the bare minimum information locally, while ensuring a consistent experience to the players. Moreover, the framework includes an objects' sync mechanism that is able to keep the status of the objects' properties consistent between the Clients and the Server.

This project has been designed and developed in collaboration with professors Maggiorini and Gadia from the University of Milan, benefiting their knowledge in the field of video-game programming and, specifically, VR and AR development.

Several metrics have been used as support to demonstrate the performance of our priority-based streaming system, such as the network traffic, the Meta Quest's FPS, CPU and GPU usage. In particular, it has been challenging to identify a metric that could objectively explain how, from the user perspective, and in critical conditions of the network, it is more important to stream certain elements of the scene, for example, closer or bigger, with a higher priority. For this purpose, three metrics have been used: $\Delta E2000$ [10], *HDR-VDP* [11], and *Pixel Comparison*. All these three metrics have the same purpose: to highlight the difference between the client's view's perspective and the scene as it would be displayed when fully loaded.

The contribution of our work to existing technologies can be summarized as follows:

- implementing a complete Server-Client multiplayer environment for VR devices, where the scene is dynamically streamed to the –initially empty– Clients;
- implementing serialization and deserialization techniques for the Unity Objects' components *Mesh* and *Material* so that they can be streamed over the network;
- implementing a system of priority with which to send the objects of the scene, based on the Player's avatar position or field of view, to improve the user perception while minimizing the resource utilization;
- implementing a objects' properties sync system from the Server to the Clients;
- implementing a Client-to-Server position and rotation update mechanism when the user grabs an object in the VR environment;
- implementing a two-handed grab system for the *Unity XR Interaction Toolkit*.

## 1.3 Document Structure

The thesis document is structured as follows: the Chapter 2 contains the related works and the main existing technologies used in the project; the Chapter 3 contains a general description of the structure of the project, along with the most significant technology choices that have been made; the Chapter 4 contains a description of the serialization methods used to stream the object's components and properties; the Chapter 5 and Chapter 6 describe the specific functionality of each device-type; the Chapter 7 discuss some VR-specific consideration made to improve the user experience; the Chapter 8 discusses the performance of the solution through the tests performed; finally, Chapter 9 summarizes the final considerations and the possible future development of the project.

# Chapter 2

# Background

## 2.1 Related Works

Although the literature lacks of a comprehensive exploration of the field of *3D Scene Streaming and Updating in VR Environment*, there are ongoing researches especially in the context of *Mesh Streaming* or *Mesh Streaming Optimization*. The emphasis of most works is on remote meetings, holopresence, or gaming applications, trying to enhance the user experience through dynamic and context-specific objects' streaming protocols. The most interesting academic publications related to our work are described below.

### 2.1.1 FLoD: A Framework for Peer-to-Peer 3D Streaming

One of the first projects found during research related to our work is *FLoD* [12], a P2P streaming 3D framework that adapts the JPEG 2000-based 3D mesh streaming in order to operate in a P2P architecture. The paper addresses the problem of streaming a common 3D scene to millions of users simultaneously: a classic client-server approach would consume a massive amount of bandwidth and CPU resources on the Server side, due to the additional complexity of dealing with 3D objects. As a solution, they propose to use a P2P network, so as to improve the scalability and affordability of the 3D scene streaming, based on the observation that users navigating through a 3D scene may own similar content due to overlapped visibility.

More in detail, the FLoD Framework executes some recurrent main tasks:

- **Partition**: divides the scene into blocks or cells, so the global knowledge is not required for visibility determination;
- **Fragmentation**: divides a 3D object into pieces, to transmit it progressively;
- **Prefetching**: predicts data usage and generates objects or scene requests to lower the latency;
- **Prioritization**: orders the objects that a Client needs to obtain in a scene;
- **Selection**: determines the peers to connect and pieces to obtain.

The workflow that a node entering the FLoD system performs is as follows.

- the joining node enters the VE (Virtual Environment) system, it receives the list of AoI (Area of Interest) neighbors and cell size from a *gateway server*;

- it requests the cell descriptions. Once obtained, the node requests for 3D objects;
- using visibility determination, it produces a prioritized piece request list and obtains the pieces;
- if the local cache does not have the desired data, requests are sent to the other data source nodes, until eventually they will reach the server;
- the node moves by sending a position update to the *overlay*, which forwards the update to AoI nodes;
- it disconnects from all the neighbors when leaving the system.

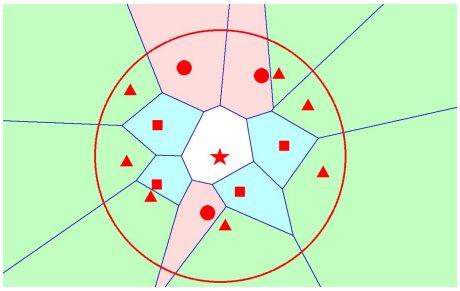


**Figure 2.1:** FLoD: Neighbors in VON (Voronoi-based Overlay Network)

Applied to the VR Multiplayer platforms, a full P2P approach to the problem such as FLoD would lower the load on the Server in the case of a large number of users but, on the other hand, a lonely player without others in the same AoI would eventually request the information of the scene directly to the Server. For this reason and due to the greater complexity of implementing a P2P network, we chose to stick with the client-server implementation, despite knowing its limits in terms of user capacity.

The metrics used in the **FLoD** article, to demonstrate the performance of the proposed solution, are as follows.

- **Bandwidth usage**: a fundamental metric to verify that the bandwidth usage of each component of the system is bounded in a certain range;
- **Base Latency**: the time interval between the initial query that notifies the client about the object to spawn and the instant when a base piece of it becomes available to a client.
- **Fill Ratio**: the ratio of data volumes between the client's obtained data and the visible data (according to the server storage). This metric provides a numerical value for the visual quality perceived by the users;

Those values can be well suited to our work because of the similar goals of the projects. For this reason, we choose to adopt some of them, as can be seen in Chapter 8.

### 2.1.2 RING: a client-server system for multi-user virtual environments

A second project found during research presents a system (called "RING" [13]) whose main focus is to handle the interaction between large numbers of users, maintaining a consistent state between the entities of the scene. The key concept used by the RING system is the *Occlusion Culling*: a Client of the simulation needs only to receive

updates of the other entities with which it visually interacts. The key idea is illustrated in Figure 2.2.

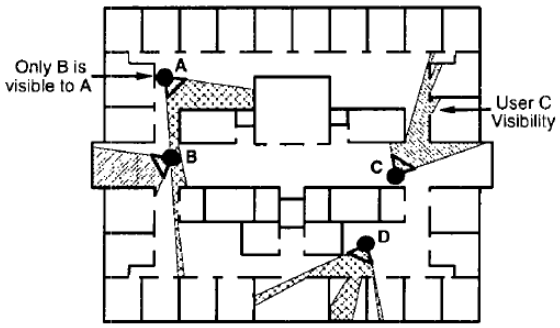



**Figure 2.2:** RING: entity-entity visibility culling

The system is composed by multiple Servers, arranged in a ring topology. When an action is performed by a Client, the update message is sent to the Server to which it is connected. The Server then calculates the occlusion, to anticipate which Servers and Clients serve users that will perceive the update; then it sends the message to the interested Clients, or to the Servers that will eventually propagate it further.

Another possible optimization pointed out by this paper is to use a multi-resolution simulation, so as to reduce network traffic and Client behavioral simulation processing. If, for example, an entity E is further away from the Client A than the entity B, but both visible, B's updates can be sent with a finer resolution. In fact, E may be far enough so that small updates are imperceptible to A.

The results show that the system achieves the goal of greatly reducing the messages exchanged, recording a 40x decrease in the number of messages processed by Clients. A similar approach has been used for the update phase of this thesis work, except for the implementation of a real-time occlusion culling technique, which would require a dedicated deeper investigation and high performance servers. The solution we developed uses the *Frustum Culling*, a strategy that takes in account all the objects in the frustum-view of the Player, regardless of their actual visibility (see Section 5.2.2 for more details).

The main focus of the RING project is to allow a large number of simultaneous entities to move in the scene. For this reason, the experiments consist of an entity that navigates through virtual environments containing 64, 128, 256, 512, and 1024 entities managed by Clients. Each test is then repeated for 25, 05, 100, 200, 400, and 800 rooms. The metrics reported are the average rates of messages received by individual Clients in each test.

### 2.1.3 Benchmarking Open-Source Static 3D Mesh Codecs for Immersive Media Interactive Live Streaming

This third work [14] focuses on the problem of efficiently stream Time-Varying 3D Meshes (TVM), in order to facilitate the concepts of telepresence and teleimmersion. As an evolution of Video Streaming, these technologies need to be supported by a codec that compresses the meshes before they get sent. The article benchmarks the

performance of existing open-source codecs for static 3D mesh compression, in the context of immersive media interactive live streaming. The codecs considered are the following. Google Draco, Corto, MPEG's Open 3D Graphics Compression (O3dge), and OpenCTM.

This article exceeds the purpose of this thesis work, since all the meshes involved in the project are static. However, future iterations of the project could benefit from a better compression strategy, in particular for what concerns mesh and texture streaming. Moreover, the project can also be expanded to include dynamic or Time-Varying meshes, opening the doors to other application fields, such as holoportation or tele-presence meetings and experiences.

### 2.1.4 Distributed Rendering in Vulkan

Finally, we took inspiration for some aspects of the project from the thesis work of Giacomo Parolini [15], Distributed Rendering in Vulkan. The base idea of his project is the same as ours: to create a game-streaming framework, capable to dynamically send the required objects, meshes, and textures from a central Server to the connected Clients. The Server's job is to host the simulation of the game, manage the raw assets, and send to the Client the data it needs with the due priority. The Client's job is to process the data from the Server and render the game world, as well as handle player input. The scheme of the Client-Server tasks and interactions is reported in Figure 2.3

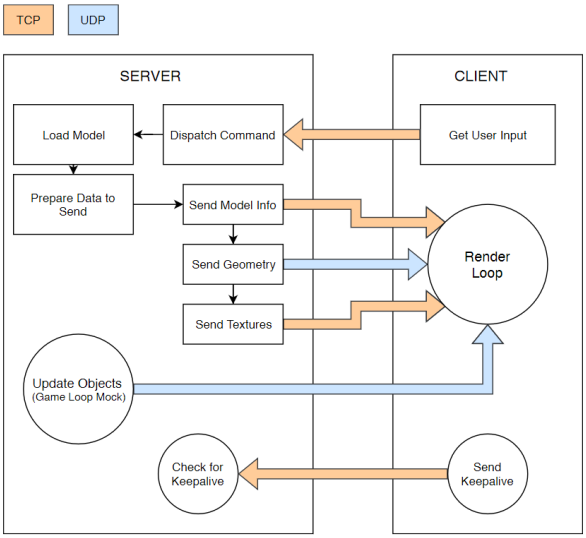



**Figure 2.3:** Distributed Rendering in Vulkan: Client and Server tasks

The Server device makes use of 6 threads, to accomplish each task on the rendering pipeline:

1. the **main thread**, responsible for starting the Server and running the main "game loop";
2. the **UDP active thread**, which sends geometry data, object transforms, and dynamic lights updates to the Client;

3. the **UDP passive thread**, which receives UDP ACKs from the Client and tells the UDP active thread to stop sending the acknowledged packets;
4. the **TCP active thread**, that is, the one pushing resources to the Client;
5. the **TCP passive thread**, which receives all TCP packets coming from the Client and dispatches them to the appropriate threads;
6. the **keepalive listening thread**, which periodically checks for the latest time a KEEPALIVE message was received and drops the Client if that time is beyond a certain threshold.

On the other hand, the Client device performs the following operations.

- it initializes the application;
- it tries to connect to the Server;
- if the handshake is successful, it initializes Vulkan resources;
- it runs the main loop;
- upon disconnections, it cleans up the resources and terminates.

In contrast to what we have done, his project mainly targets MMO (Massive Multiplayer Online) games or other online video games for *Desktop* devices, while we developed a tool that could work on VR applications of any kind. As an additional difference, his framework is built on a lower level, including the building of a custom rendering engine, using C++ in combination with the Vulkan graphics API [16], while we preferred to build our system on top of an existing game engine, to (ideally) make possible the migration of existing projects, the fruition in an environment well known by many game developers, and the integration with other third-party plugins and extensions.

## 2.2 Performance Metrics

Each of the existing works analyzed in Section 2.1 focuses on a different aspect of the 3D objects' streaming pipeline; for this reason, they make use of metrics that can better demonstrate the advantages achieved by their solution, ranging from bandwidth usage to the volume of objects loaded in the scene. The similarity to some aspects of this thesis work allows us to repurpose some of the aforementioned metrics; moreover, some additional metrics have been included with the aim of explaining the subjective perception of the environment being streamed around the player.

### 2.2.1 Network Traffic

When dealing with multiplayer applications, it is essential to test the functioning of the system through network tests, carried out under stress conditions. A first metric that tells us the amount of information shared between two devices is the data traffic outgoing from the Server.

### 2.2.2 Frame Rate

Another fundamental metric in the 3D graphic field is the frame rate, measured in frames per second (FPS). The Meta Quest 2 internal monitors are capable of a total refresh rate of 120Hz, if the experimental settings are enabled; this allows the user to benefit up to 120 FPS in-game. In general, we want to keep the frame rate as high as

possible, ideally never going below the threshold of 60 FPS.

### 2.2.3 CPU and GPU

The last metric related to the Oculus hardware that is interesting to measure, to get a complete overview of the device functioning, is the processor and graphic card usage. For this reason, we measured the average CPU and GPU usage during all of the experiments.

### 2.2.4 Delta E 2000

The following metric describes how human eyes perceive the difference between two colors. The value of $\Delta E$ was proposed for the first time by the International Commission on Illumination in 1976. To better approximate the human vision of the colors, a new color space called CIELAB has also been introduced. In this color space, colors are represented by the combination of the following three values:

- L* represents lightness, with 0 being perfect black and 0 perfect white;
- a* represents red-greeness of the color. Positive values of a* are red, while negative values are green;
- b* denotes yellow-blueness of the color. The positive values of b * are yellow, while the negative values are blue.

These values are based on the fact that colors can not be perceived as green and red, or as yellow and blue simultaneously. The first proposed $\Delta E$ formula is:

$$\Delta E_a^* b = \sqrt{(L_2^* = L_1^*)^2 + (a_2^* - a_1^*)^2 + (b_2^* - b_1^*)^2}$$

It calculates the Euclidean difference between the colors expressed in the CIELAB space. However, the initial version of the metric and its first improvement, the $\Delta E94$, present some inaccuracy when calculating the perceived lightness of two colors. For this reason, we used the current iteration, the $\Delta E2000$, that is the most complex, yet most accurate, CIE color-difference algorithm available. To reduce the errors rate, the $\Delta E_{00}^*$ also applies a conversion from the CIELAB color space to the LCH (Lightness Chroma Hue) space. The proposed $\Delta E_{00}^*$ metric is calculated as follows:

$$\Delta E_{00}^* = \sqrt{\left(\frac{\Delta L'}{k_L S_L}\right)^2 + \left(\frac{\Delta C'}{k_C S_C}\right)^2 + \left(\frac{\Delta H'}{k_H S_H}\right)^2 + R_T \frac{\Delta C'}{k_C S_C} \frac{\Delta H'}{k_H S_H}}$$

The procedure for calculating the values in the LCH space and, therefore, the implementation of the $\Delta E_{00}^*$ metric is described in the article "The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations"[10]. For this thesis project, the mean of the $\Delta E_{00}^*$ pixels difference is used to represent the perceived difference between the scene screenshot (from the perspective view of the user) at an instant in time and the final scene, with all the assets loaded into the Client. The results of the experiment using this metric are shown in Chapter 8.

### 2.2.5 HDR-VDP

The purpose of the HDR-VDP visual metric is to compare a pair of images and predict the following values.

- **Visibility**: what is the probability that the differences are visible for an average observer;
- **Quality**: what is the quality degradation with respect to the reference image, expressed as mean-opinion score.

The model tries to accurately reproduce the human visual system, while limiting the computational complexity. Therefore, the metric HDR-VDP is used in the project's experiments to predict the perceptive visibility difference from the image displayed by the user and the final scene, with all the assets loaded into the Client, as shown in Chapter 8. For the calculation, the MATLAB implementation of **HDR-VDP-3** provided by the authors has been used. In particular, the chosen task parameter is *detection*, the same task as HDR-VDP-2, whose stages are illustrated in Figure 2.4. The perceptibility of the differences between the entire images, has been calculated as mean of the *P_map* (*quality probability* values, calculated for each pixel).

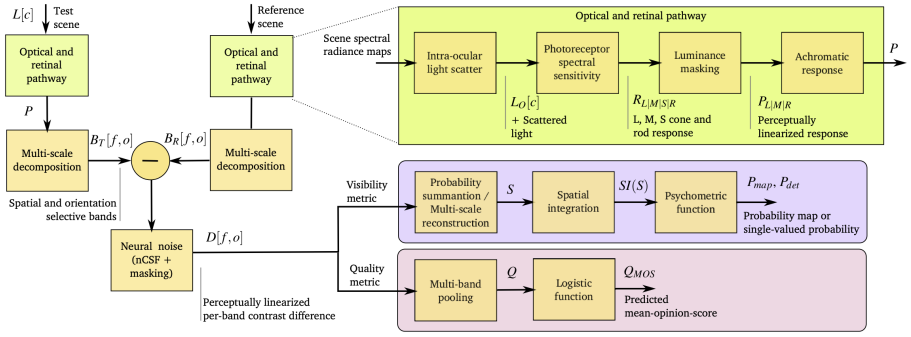



**Figure 2.4:** The processing stages of the HDR-VDP-2 metric

## 2.3 Game Engines

The proposed project aims to change the way in which the games are designed, but, because most of the game developers are already used to popular existing Game Engines, we choose to build our framework on top of one of them. This facilitates the integration with existing projects and allows us to focus on the core idea of our work, rather than in the creation of a brand new game engine.

The 3D Engine scene is dominated by two main actors: Unreal Engine [17] and Unity 3D [8]. Some other possible competitors are Godot [18], CryEngine [19] and Amazon Lumberyard [20], but these will not be taken into account for this work, as they are expensive or less popular.

**Unity** [8] is a Game Engine distributed by Unity Technologies. It is the world's most used engine, mainly because it has a very shallow learning curve. The structure of the components is easy to understand and the script programming in C# is simple and efficient. Moreover, it offers built-in compilation for multiple platforms without the need for any additional plug-in or software. Its graphics quality is slightly worse than that of its competitor, Unreal Engine, but this project will not try to achieve photorealistic graphic quality, given the limitations of the VR technology in the current state and of the network transmission requirements.

**Unreal Engine** [17] is developed by the Epic Game Company, and is currently the

leading engine when it comes to realistic visualization. It is better suited for large projects, worse for small applications or games, even though it supports both the Android and iOS mobile operative systems. The code uses C++, and the framework is less intuitive compared to Unity, but offers more complete tools for post-processing effects and for optimization. It also has a slightly smaller community base that makes it less immediate to perform troubleshooting or bug fixes.

Ultimately, taking these considerations into account, Unity 3D was selected as the Game Engine for this project. Unity also offers a simple, complete and customizable VR input library, the XR Interaction Toolkit [21], which provides a great starting point to handle the input interactions with the VR Headset and the Controllers.

## 2.4 Network Libraries

While the Object Streaming Techniques falls within the innovative part of the thesis work, the Client-Server handshaking and communication in game development has already been widely explored. To avoid spending much time in the implementation of such base communication concepts, we chose to base our work on an existing Network Library for Unity. The following are some of the most popular networking options that can be integrated into the Unity Engine:

- **Mirror (fork of UNet)** [22]: UNet was the official networking solution provided by Unity until it was deprecated in 2018. The library was forked by the Mirror project, which offers nearly the same functionalities and APIs. This includes a Server-Client or Host-Client architecture to build multiplayer applications. It is free and open source. Mirror is stable and has a lot of documentation, also due to its similarity with UNet. UNet was deprecated because it was not compliant with the performance, scalability, and security requirements of the developers, and the same issues are present in Mirror.
- **Photon PUN** [23]: Photon PUN (Photon Unity Networking) is a Unity package for multiplayer games. It offers a free plan that can manage up to 20 concurrent connections, and a pro subscription that offers some advantages, including Photon Cloud and up to 100 users. It is one of the easiest solutions because it offers a High-Level abstraction over the complexity of the Photon Realtime Engine. Although it is a great option to begin with, it is better suited for small toy projects, and it does not offer the same level of customization as the others.
- **Netcode for GameObjects** [24]: Unity Multiplayer Netcode (also called MLAPI) is the new Unity official solution for Multiplayer. It consists of a Mid-Level library that hides the complexity of the Low-Level API and the Transport Layer. The library is still under development, but it offers an extensible solution that is highly customizable and has a lot of community support. We chose this library mainly because it is the first-party solution for Unity projects and so it represents the future standard technology for Unity Multiplayer.

### 2.4.1 Netcode Overview

In order to use the Netcode library, a Unity project must include the `Network Manager` component, connected to a `GameObject` of the scene. This component is the entry-point for the network initialization. Each instance of the game can be launched as Client,

Server or Host (this last type of instance can both act as a Server for the other players and participate in the game as a Client itself). The `NetworkManager` can be equipped with the `Network Player` Prefab, an object that will be instantiated for each Client connection and that represents the Player Character in game. The position of the player is automatically synced between Clients and Server.

If another object of the scene needs to be spawned in all the clients and automatically synced between Clients and Server, it must include the `NetworkObject` component as well. For the majority of the objects in the project this functionality has not been used, because they are handled by our custom streaming method. Indeed, the `Netcode` spawn functionality requires that the objects are already saved in the Client's app memory as well as *Prefabs*, while our aim is to make the two instances of the project as independent as possible by keeping the whole scene only in the Server.

## 2.5 Networking Protocols

The framework subject of this thesis work needs to send serialized assets from the Server to the Clients. We identified two different protocols for sending these assets, by using the technologies already included in the game engine and in the Netcode library, each with its advantages and drawbacks.

### 2.5.1 RPC

The preferred way to send messages and perform remote operations using the Netcode library is the RPC (Remote Procedure Call [25]). It consists of a function or method call, performed remotely from one executable to another, as illustrated in Figure 2.5. To use the RPC primitive, an object must include the component `NetworkObject`, which assigns a shared identity to it, keeping the multiple instances connected. The RPC are created by adding the attributes `[ServerRpc]` or `[ClientRpc]` above the method signature.

There are two main different types of RPC:

- the **ClientRPC**s are functions called by the Server on the Clients, and they trigger all the Clients simultaneously. Since the Server has full authority by default, there are no limits in the operations that it can perform on the Clients' objects through ClientRPCs;
- the **ServerRPC**s are functions called by one of the Clients on the Server. In order to operate on the Network Object where the RPC is called, the Client must be the *Owner*, otherwise the ServerRPC method's attribute has to include the parameter `(RequireOwnership = false)`.

An RPC can be used to send a signal to the remote host, but it also supports function parameters, as long as they are C# or Unity primitives, or they implement the `INetworkSerializable` interface [26]. Setting the RPC to *Reliable*, will ensure the delivery of the message, using a RUDP (Reliable UDP) packet under the hood. The RPCs are not sent immediately by the library; instead, they are enqueued and sent on a periodic basis. This sets a limit to the number of RPCs (or to the size of them) that can be submitted in a period of time: the MTU (Maximum Transmission Unit) of the Unity Transport;

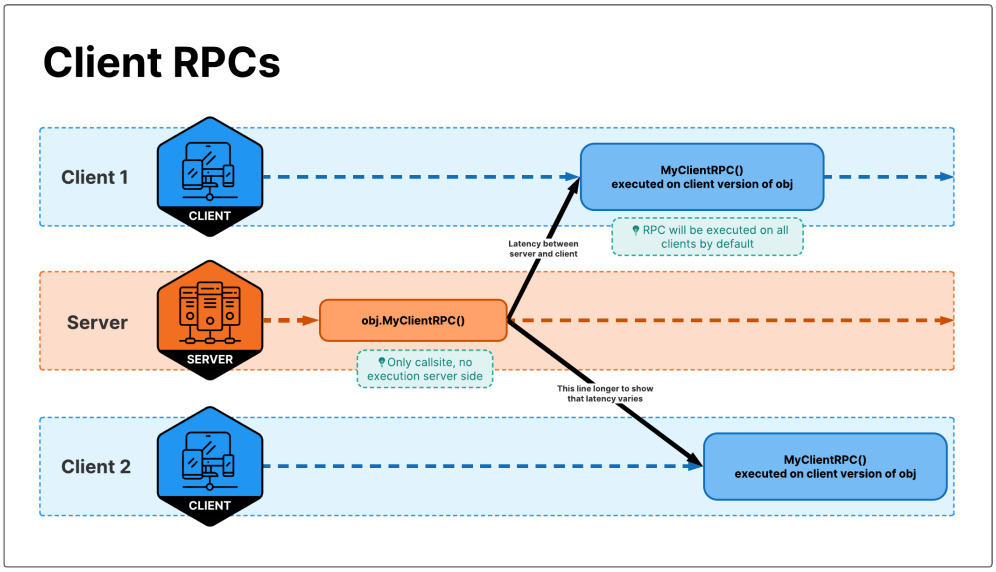



**Figure 2.5:** Client RPC

### 2.5.2 HTTP

Another way to send information, which bypasses the limits imposed by the Unity Transport Layer, is to use the HTTP protocol. **HTTP** [27] is a protocol for fetching resources, often used in the Web for *HTML* documents. The main aspects of the HTTP protocol are as follows:

- it is designed to be human readable, thus providing easier testing for developers, and reduced complexity for newcomers;
- it is extensible, using the HTTP headers;
- it is stateless, meaning that there is no link between two requests being successively carried out on the same connection;

The HTTP request packet is structured as follows (Figure 2.6):

- the HTTP method, usually GET, POST, or a noun like OPTIONS or HEAD that defines the operation the client wants to perform;
- the path of the resource to fetch; the URL of the resource is stripped from elements that are obvious from the context, for example without the protocol (http://), the domain (here, developer.mozilla.org), or the TCP port (here, 80);
- the version of the HTTP protocol;
- optional headers that convey additional information for the servers;
- a body, for some methods like POST, similar to those in responses, which contain the resource sent;

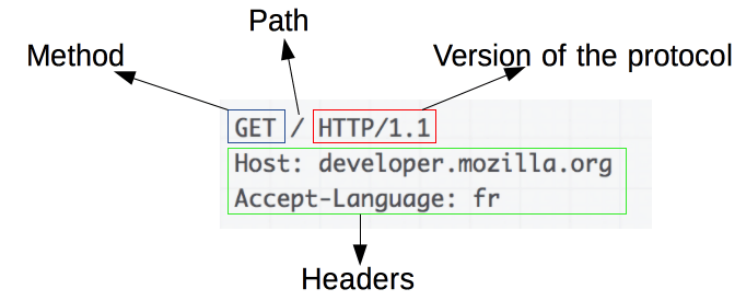



**Figure 2.6:** HTTP Request Packet

An example of an HTTP response packet is shown in Figure 2.7 and consists of the following elements:

- the version of the HTTP protocol they follow;
- a status code, indicating if the request was successful or not, and why;
- a status message, a non-authoritative short description of the status code;
- HTTP headers, like those for requests;
- optionally, a body containing the fetched resource;

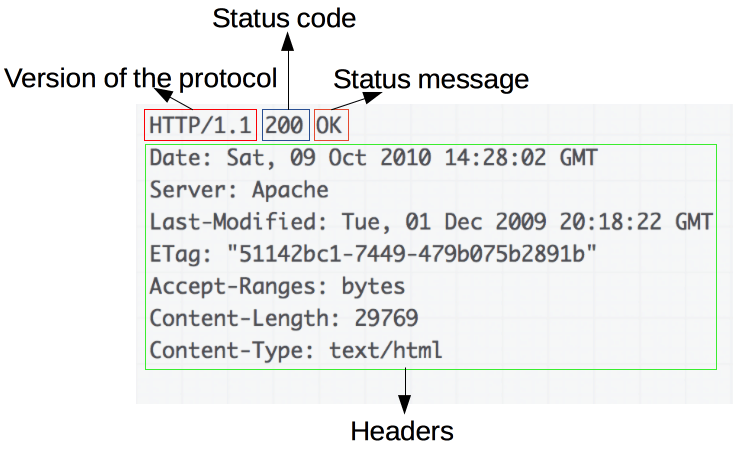



**Figure 2.7:** HTTP Response Packet

In our framework, an HTTP Server launched in the background of the game Server, alongside of the Netcode system, can expose the serialized resources needed by the Clients. Although this method has the drawback of adding some payload to the message, it is the easiest way to share and download a large amount of data, such as *meshes data* and *textures*. Moreover, the HTTP Server functionalities could be hosted in a different machine (as long as it already contains all the serialized GameObjects), this way the computational overload on the Netcode Server device is decreased.

# Chapter 3

# VR Object Streaming Framework

## 3.1 Project Goals

The project purpose is to develop a library that is as complete as possible: in particular, it needs to handle all the communication between the Clients and the Server, including the player movements and actions, and the environment's streaming and update. The software should also be ready to use, integrable, and expandable with custom behaviors. The library should include at least the following functionalities:

- it should be able to handle the initialization and connection phase of the Server and of the Clients;
- it should allow the instantiation of the Client's Player Object into the Server and into the other Clients, keeping track of its movement. In particular, because the project is focused on a VR type of interaction, at least the movements of the head and of the two hands needs to be synced;
- based on the Player movements, the Server should calculate which objects to send to each Client. The objects will have a *priority* value that indicates how crucial it is to show them to the Client, avoiding in this way to send low priority objects that are far away or not visible by the Client. For instance, a big building will have a high priority, while a street sign can be ignored when distant enough.
- the objects need then to be sent to the Client, this involves two distinct phases:
  - sending the initial and most important information of the objects, such as the name, an identificator, the position, rotation, and size;
  - sending additional object data, such as the mesh, textures, and materials.
- After being sent to the Client, some objects also need to be periodically updated in order to reflect the forces and actions that both the Server or the Clients may apply to it. More specifically, the most important properties that will need to be updated are the position and the rotation. Eventually, updates could also include some other secondary properties, such as the size or color of the objects, thus permitting greater diversity in the type of experience provided by the final product.

## 3.2 Workflow

The framework consists in the interoperation of the two types of devices: the Server (a computer that hosts the simulation) and the Client (the Meta Quest device in *standalone* mode that connects to it). The main purpose of the framework is to stream the objects contained in the Server scene to the empty Clients that connects. A secondary goal is to keep the status synced between the different instances of the same object in the two types of device. The streaming operations, that constitute the primary interacion between the two devices of our 3D objects' streaming framework, are illustrated in the diagram in Figure 3.1.

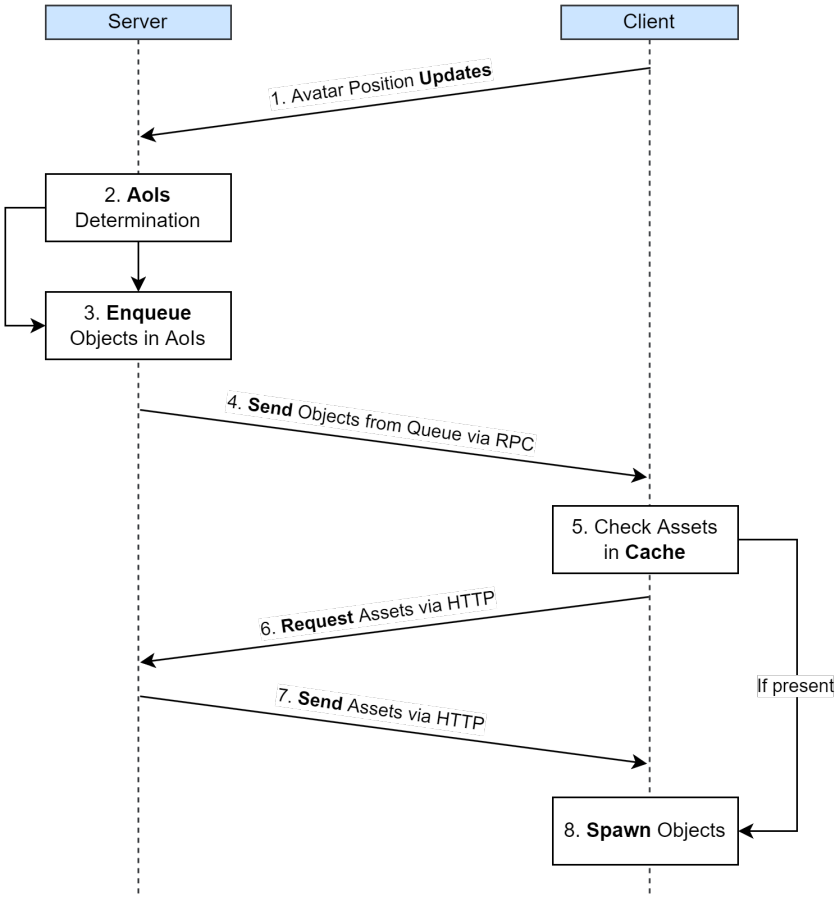



**Figure 3.1:** Framework workflow

1. While the Client is connected to the Server the Client sends periodic updates of the position/rotation of the Player's avatar to the Server;
2. the Server uses the Client position information to determine the AoIs (Areas of Interest): three circular zones around the Player, where the objects of different priorities need to be sent to the Client (for more details about zones, see Section 5.2.1);

3. the Server adds to the Client's specific queue all the objects in its AoIs, assigning them a priority based on the distance from the Player and its relevance;
4. the Server periodically sends the objects base data contained in the queue to the Client, using a *Netcode* RPC method;
5. the Client receives the objects base data and it checks in the local cache for the assets needed to draw it;
6. if the Client's cache does not contain the assets (Mesh and Material) of the object, it requires them via HTTP to the Server;
7. the Server sends the assets to the Client via HTTP;
8. the Client draws the object with the obtained assets to the scene.

## 3.3 General Architecture

The macro-components and the connections with the main libraries are illustrated in Figure 3.2. The components can be further divided into two parts that control different behaviors of the objects.

The **Spawn** part (also referred to as *Objects Streaming*) has the objective to send the objects from the Server to the Clients. It makes use of both Netcode and HTTP libraries to share different parts of the objects. Initially, with the Clients' information provided by the Netcode system, the *Colliders* are created. The *Colliders* establish which objects around each Player need to be sent and assign them a priority. The *SpawnManager* sends the initial information about the object to the Client through the Netcode RPC primitive, meanwhile it makes available to the HTTP thread the components of the object, like Mesh and Materials, saving them to a buffer memory. When the RPC is launched on the Client, it checks in its own local cache for the needed components of the object, and it eventually requires them via HTTP to the Server.

The **Sync** part has the objective of keeping synchronized the changes of the objects' properties between the Server and the Clients and vice versa. They both use the *SyncManager* class to sync the properties changes, using the Netcode Library. The difference between the two actors is that the Client can only update the position and rotation of the objects that it owns, while the Server can send to the Clients any properties change, that includes the modification of object's lighting or material's color.

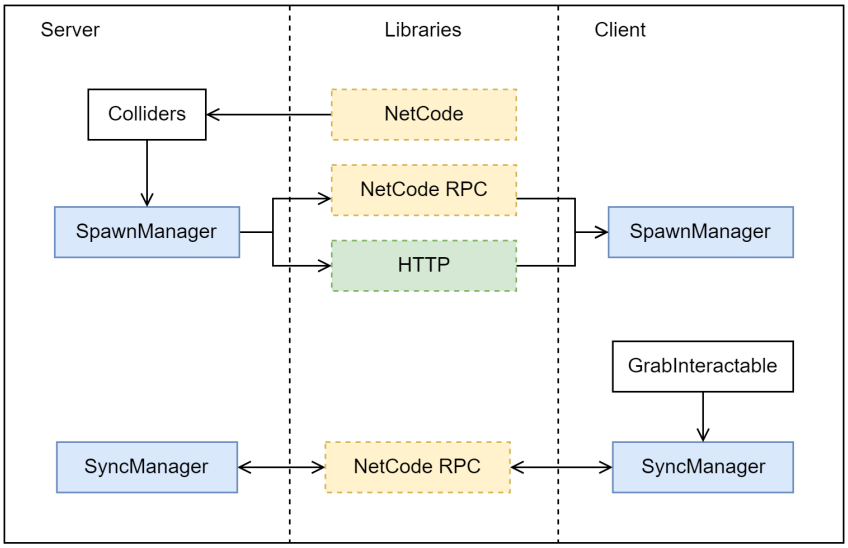



**Figure 3.2:** General Architecture

## 3.4 VR Object Streaming Architecture

The framework we built is composed of two actors: the Server and the Client (the Host option is still feasible with some adjustment, but has not been optimized for the purpose of this work). The code is organized to reflect these two types of devices.

The `Server` namespace contains the components that will be loaded in the Server instance of the project: its purpose is to receive the Clients' position, send them the base information of the objects and their updates. The Server also performs the physics simulation of all the entities and exposes the objects' specific components. The Client, on the other hand, connects to the Server, receives the objects' data, asks for the additional components and attaches them to the local GameObjects. It sends and receives the objects' properties updates, consequently modifying their local copy.

There are also some additional code components, that both the actors make use of: this includes all the serializable object types, and some static utility functions. The fundamental class, `NetObject` is the fundamental class used by nearly all the other components. This class is unique for each GameObject that needs to be spawned into the Clients or synced. It contains, in fact, some important property such as the object identificator, its priority into the scene and the current owner of the object (that can be the Server, or the Client that is using it).

### 3.4.1 Server Architecture

The Server is the actor that rules the whole game infrastructure. For this reason, it is the first one to be started, and it is composed by two main blocks of classes: the *Spawn Classes* and the *Sync Classes*. The *Spawn Classes* (Figure 3.3) are the ones used for the initialization phase: the Server loads the objects of the scene, grabbing them from the local `Resource` folder; when a player connects it creates two *Colliders*

(their purpose is explained in Section 5.2) and handles the sending of *Serialized Objects* to the Clients.

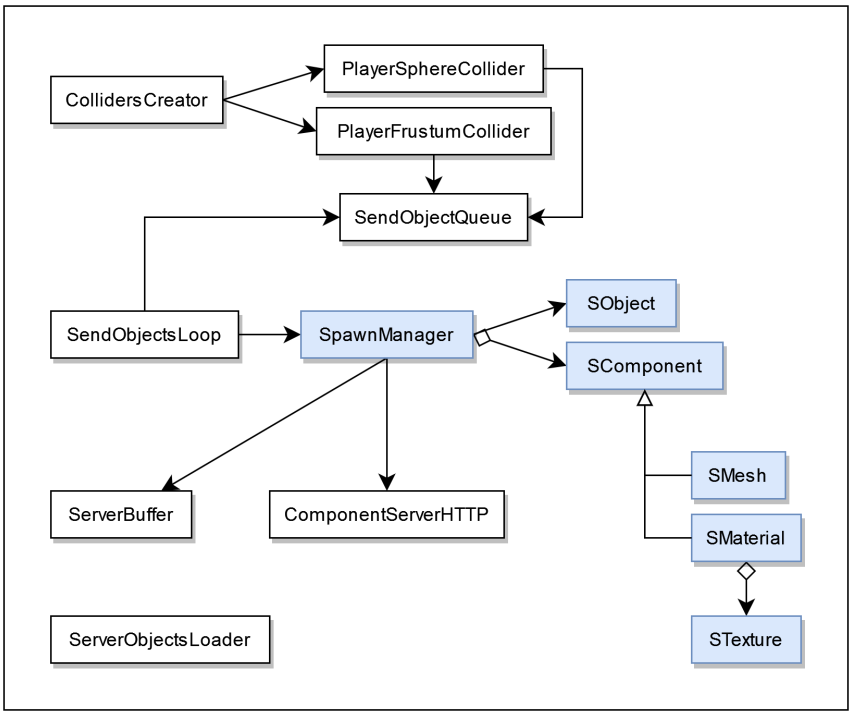



**Figure 3.3:** Server Spawner Class Architecture

For dynamic or movable objects, the framework needs a method to sync the desired properties' changes so as to keep the same status through all the Clients. This is achieved by using the classes in Figure 3.4. The *Sync* Classes have the purpose of sending updates to the clients whenever there is a significant change in the target property (such as position, rotation, color, or light intensity). The *Action* Classes, on the other hand, change the property of the attached GameObject when they receive a corresponding signal from a Client.

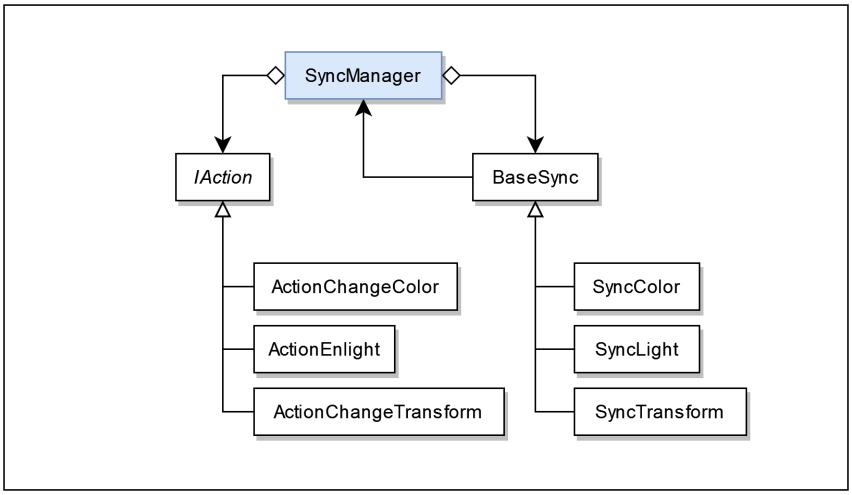



**Figure 3.4:** Server Sync Class Architecture

### 3.4.2 Client Architecture

As previously mentioned, the Client shares some classes with the Server. In particular, it needs to receive the objects and components data, so the *Serializable Components* and *Serializable Object* classes are used by both the Server (which uses them for the serialization) and the Clients devices (which uses them for the deserialization). Moreover, because the Client receives the initial object metadata by Remote Procedure Calls, it needs to share the same `SpawnManager` object of the Server (the RPC functioning is better explained in Section 2.5).

The Figure 3.5 shows the architecture of the Client as concerns the objects' spawn. The difference with the Server one is that here the `SpawnManager` will recreate the objects with the data received from the *ClientRPC* (if not already present in the scene), and then, by iterating through the temporary dictionary, using the component ids, it will download the ones that have not already been downloaded, using the HTTP protocol. The `SpawnManager` in the Client devices uses the `ClientBuffer` class to temporary store the objects and components, and the `ComponentClientHTTP` class to handle the HTTP requests to the Server counterpart.

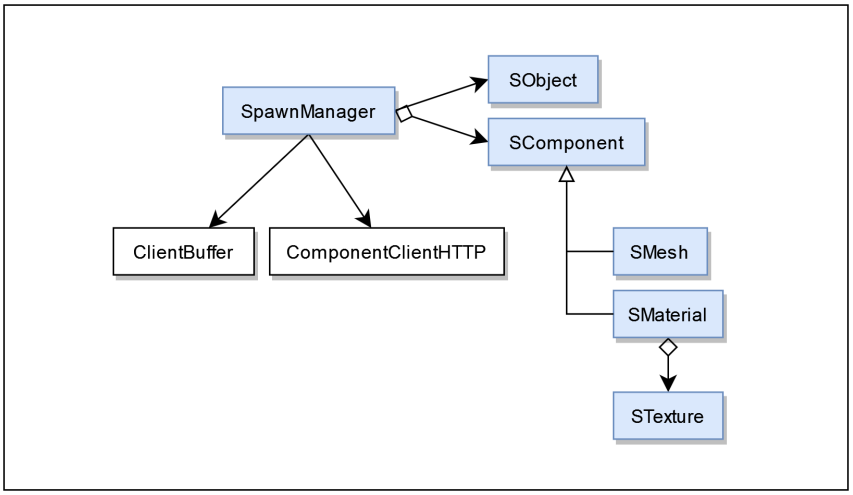



**Figure 3.5:** Client Spawner Class Architecture

In the opposite way of the Server, the Client needs also to synchronize the objects' properties when the `SyncManager` receives update messages from the Server (this is done using the RPC mechanism: Section 2.5). The `SyncClient` components will receive update messages from the Server, and they will update the objects' properties consequently.

The `XRGrabNetworkInteractable` class inherits from the `XRGrabInteractable` (the class of the XR Interaction Toolkit [21] aimed at managing the objects grabbing with XR controllers). The `XRGrabNetworkInteractable` component adds some additional behavior to the object to which it is attached:

- set the object ownership to the User, when it grabs it;
- send updates to the Server for position or rotation changes of the object;
- send a signal if the activation button is triggered while the object is in the Player's hands.

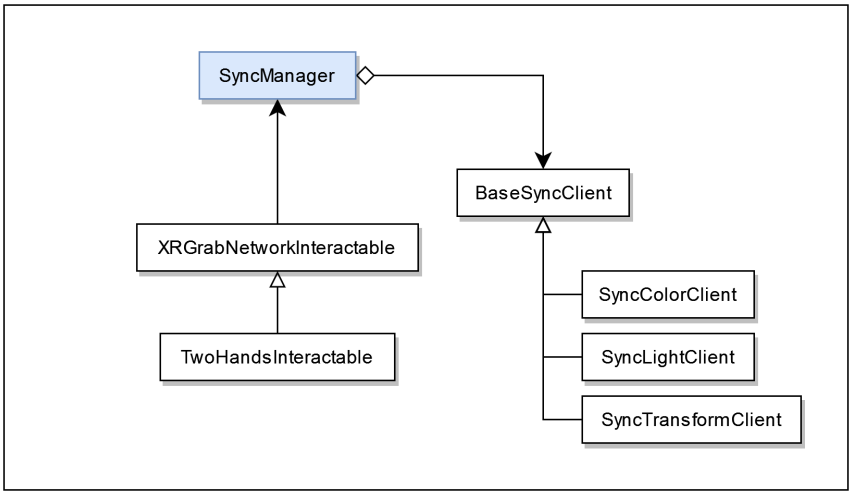



**Figure 3.6:** Client Sync Class Architecture

# Chapter 4

# Objects Serialization

The first and most important challenge in designing this kind of architecture is: how can an object be serialized, in order to be efficiently sent from Server to Clients? By default, the Unity GameObjects are not directly serializable, because they are containers for other components, and do not include many specific data. It is possible to serialize the single components, using the JsonUtility [28] library included in the Framework, or the BinaryFormatter [29], but we chose to implement custom serialization methods, in order to select and minimize which information to send to the Clients. This allowed us to optimize the data flow, and even to choose to apply some compression to the components' data.

## 4.1 Serializable Objects

The first piece of information that our framework needs to send over the network is the `SObject` (Serializable Object): a data structure representing the base information of an object in the scene, which has the purpose to inform the Client of the position and the features of the object that it will need to download and render. Each `SObject` is created by the Server and sent to the Client when it is in proximity of the object (the `SObjects` are sent using the priority mechanism explained in Section 5.2). The `SObject` structure contains the following information:

- an identificator to uniquely reference the object between all the connected devices;
- the name of the object *;
- position, rotation, and scale, all using a custom serializable `Vector3` wrapper;
- the object priority in the scene *;
- the identificator code of the parent;
- the components' identificators: an array containing the ids of the components that the object will require and then attach.

  The **Object ID** is a unique number that identifies the object in the scene. It is calculated using the `GetHashCode` C# method with the object's name as input (or `"[name]>[parent_name]"` if the object has a parent);
- some boolean fields, that indicates which components would need to require periodic updates from the Server.

* *these elements are not mandatory in the Client, so they can be omitted. They have been inserted only as additional debug information.*

To send the `SObject` structure, the framework uses Netcode ClientRpc [25]. There are two main reasons for this choice:

- first of all, we did not implement an HTTP listener on the Client, and since it cannot perceive the objects in the interest area before building them, it cannot actively use the HTTP protocol to require the shared contents from the Server;
- secondly, the initial information of the object is composed by a few bytes (less than 200 B), so it can be fitted in a Netcode RPC call, whose limit is imposed by the UDP protocol (65,535 B).

## 4.2 Serializable Components

After the main object data has been received, the Client needs to attach the additional components to the object. This is done using the *componentIds* field: for each component identificator into the array, the Client tries to download the corresponding resource from the HTTP Listener exposed by the Server. Note that if the resource is already present in the Client Components Cache, it does not need to be downloaded once more: this is particularly useful for recurrent elements in the scene, such as repeated textures, materials, and meshes.

The resources are all the same base type (`SComponent`), thus permitting the Client to use them without taking care of the specific resource type. The resources are serialized to the JSON format by the server using the JsonUtility library [28], shared with the HTTP protocol, and then deserialized back by the Client. An example of serialized *Mesh* component is shown in Listing 4.1. The base abstract class, in addition to providing serialization and deserialization functionalities, contains the overridden method

```
public abstract void AttachTo (SObject object)
```

The purpose of this method is to initialize the component, attach it to the given GameObject, and copy all the serialized fields back into the component just created.

```
HTTP/1.1 200 OK
Server: Mono-HTTPAPI/1.0
Date: Sat, 22 Oct 2022 08:38:35 GMT
Transfer-Encoding: chunked
Keep-Alive: timeout=15,max=99

{
    "$type":"Serializers.SMesh, Assembly-CSharp",
    "_triangles":[ [0, ...] ],
    "_vertex":[
       {
          "$type":"Utils.SVector3, Assembly-CSharp",
          "x":1.0,
          "y":-1.0,
          "z":8.176204
       },
       ...
    ],
    "_tangent":[],
```

```
    "_uv":[
       {
          "$type":"Utils.SVector2, Assembly-CSharp",
          "x":0.6402893,
          "y":0.662574
       },
       ...
    ],
    "_boundsCenter":{
       "$type":"Utils.SVector3, Assembly-CSharp",
       "x":0.0,
       "y":0.0,
       "z":4.379275
    },
    "_boundsSize":{
       "$type":"Utils.SVector3, Assembly-CSharp",
       "x":2.261132,
       "y":2.261132,
       "z":10.89601
    },
    "Id":155368602,
    "Name":"Sword Instance"
}
```


**Listing 4.1:** Serialized Mesh Component HTTP packet

### 4.2.1 Meshes

The *Mesh* is the first and most important element that identifies a 3D-object: it is the structural representation of the model, consisting of a collection of vertices and polygons. The polygons are typically quadrangles or triangles, but triangles are more often used to benefit from current 3D graphic cards hardware acceleration. The polygons can be further broken down into vertices in X, Y, Z coordinates and lines [30]. In addition, some other data structure is needed by Unity in order to properly render the Game Object:

- the **Normals**: a depiction of the orientation of a polygon's surface, essentially a perpendicular line interjecting the plane. It is used by the engine to determine the surface's or corner's orientation toward a light source;
- the **Tangents**: a vector that follows the mesh surface along the horizontal texture direction. Tangents in Unity are represented as `Vector4`, with $X$, $Y$, $Z$ components defining the vector, and $W$ used to flip the binormal if needed;
- the **UV**s: arrays of *Vector2*s that can have values between (0,0) and (1,1). They represent the fractional offsets of the polygon into a texture. In other words, a UV group tells what portion of a texture to render on each specific polygon. A mesh can contain more than one UV group, for example if the object uses more than one material.

Those and some other minor *Mesh* metadata are packed within the object type `SMesh`, in order to be sent to the Clients. Being a `SComponent` sub-class, the `SMesh` type also contains the serializing features and the `AttachTo` method.

One problem that occurred in the early stages of the project was how to "split" the mesh

objects to obtain smaller-sized packets. The Netcode's RPCs that were initially used to send the components, in fact, have a maximum payload size of $2^{16}$ *bytes* = 65536 *B*. We designed some possible strategies to overcome this problem, but in the end none of them was used because of the choice to use the HTTP protocol instead, which automatically partitions the message to fit it into a TCP packet.

- the first option was to reduce the number of polygons and vertices, using a technique called *Mesh Simplification* [31][32]. This method is often used for games and 3D simulations, to reduce the LOD (Level-Of-Detail) for distant objects or to reduce the computational load (an example of mesh simplification can be seen in Figure 4.1). Even though this is a great improvement in terms of packet dimension, it requires a lot of additional computation by the Server, it degrades the user experience and, if the network gets better, it is not possible to "update" the mesh on the player's device to improve the graphic quality without a retransmission of the whole mesh;
- another possible solution, that has not been deeply explored, is to split the mesh in multiple parts, for example taking only a portion of the triangles array, and its relative indexes, normals, tangents, and UVs. By doing so the Client initially receives the first part (for instance, the base part of the object), with this information it can already render that portion, to provide the player and idea of the position of the latter. When the other packets will arrive, the corresponding parts of the object will be added to the rendered mesh. This method has the obvious drawback of requiring a custom stream management solution. Moreover, it requires the duplication of the boundary vertices and data.
- the last option is to build only a custom byte-stream management, splitting the final byte representation of the mesh in different pieces. This solution does not allow the partial reconstruction and rendering of the object by the Client.

**Figure 4.1:** A sequence of approximations generated using Garland and Heckbert's algorithm [33]. The original model on the left has 5,804 faces. The approximations to the right have 994, 532, 248, and 64 faces respectively.

### 4.2.2 Materials

When dealing with 3D Models, the *Material* is another fundamental element to take into account. A material describes how the surface of an object is rendered, so it contains its optical properties like color, *Texture* or if it is matte or shiny. In Unity, a single object can also contain more than one material, if different portions of the mesh need to be displayed with different properties. There are plenty of properties that can be settled into a Unity Material, that also varies depending on the *Shader* used. For the purpose of this work, we choose to use the Unity Standard Shader and to serialize only a restricted set of properties:

- the **rendering mode**, to describe how the material will be rendered, the mode can be: opaque, cutout, fade, or transparent. The models of the test scene use

only the *opaque mode* for the materials with textures without transparency, and the *cutout mode* for the textures with some transparent area, like the tree's leaves;

- the **base color** of the material;
- all the **textures** used by the material (see Section 4.2.3);
- the **emission** field, to indicate if the material is a source of light;
- the **emission color**;
- the **cutoff** amount, used in the *cutout* mode as a threshold for the transparency value.

### 4.2.3 Textures

In computer graphics, a *Texture* is a raster image wrapped around the surface of an object in the scene. The graphics card can quite efficiently perform the wrapping operation, so the textures help to create a more realistic environment, while using fewer polygons and vertices, so as to lower the computational load.

The UV mapping is a 3D-modeling procedure that consists of projecting the 2D image to the surface of the 3D mesh (the letters U and V indicate the image coordinates, because X, Y and Z are already used to denote the axis of the 3D space). The way in which the texture is aligned with the Mesh's polygons is described by the *UV group* arrays of the Mesh (as described in Section 4.2.1).

The Unity Material components contain different types of textures, each of which needs to be serialized and sent to the Clients in order to correctly render the surface of the objects. Some examples of textures of the Unity Standard Shader, which are used by the models of this project, are:

- the **Albedo**: the base color input that defines the color of the surface;
- the **Metallic Map** (or **Metalness**): a black and white texture that specifies where the material reflects the light like metal (white areas) and where it does not (black areas);
- the **Normal Map**: texture used to fake the lighting of bumps and dents on the material surface. It is stored as a regular RGB image, where the RGB values correspond to the X, Y, and Z coordinates of the normal vectors (similarly to the Mesh's *Normal* component, described in Section 4.2.1).


**Figure 4.2:** Texture representation showing respectively: (a) the Albedo, (b) the Metallic Map, (c) the Normal Map and (d) the final rendered result.
Model by Slayer4Demons (CC Attribution)

In our framework, the object's textures are serialized using the `STexture` class, and embedded in the `SMaterial` component that uses them, before being sent to the Client.

As can be seen in Listing 4.2, the `STexture` constructor takes in input a boolean parameter `alpha` that specifies whether the texture contains some transparency (so if it uses the *alpha* channel). In this way, the opaque textures (the majority of the textures in the scene) can be serialized in JPG, thus reducing the size of the packets to send.

```
public STexture(Texture2D texture, bool alpha = false) {
    _format = texture.format;
    _textureData = alpha
        ? texture.EncodeToPNG()
        : texture.EncodeToJPG();
    _width = texture.width;
    _height = texture.height;
}
```

**Listing 4.2:** STexture Constructor

To achieve better results in terms of performance of the streaming, the textures' size should be settled to a maximum of 512×512 px, otherwise their download will be slower, thus blocking the streaming of the meshes and of the other objects of the scene and negatively impacting the user perception.

A possible solution to overcome a poor bandwidth connection could be to send out a lower-resolution version of the texture, in a similar way to what could be done with the Mesh's polygons number (Section 4.2.1), but this approach would have the same main problems: it would require additional computation and retransmission in case of an upgrade of the network connection. A more clever strategy to temporarily address the lack of the texture, could be to embed the average color of it in the `SObject` class, and use that as a material color until the texture is fully downloaded.

### 4.2.4 Other Serializers

The JsonUtility Library [28] is able to convert only *Serializable Types* [34] and that does not include the Unity specific types, such as Vectors. For this reason, it has been necessary to create custom serializable data types to be used by the Serializable Components (Section 4.2) for every non-primitive field they make use of.

The Serializable classes are implicitly convertible, using the .NET *implicit* operator (as shown in Listing 4.3), so they can be used as direct replacement to the original type. The classes that have been created are:

- **SVector2/3/4**: serializable vectors containing, respectively, x,y/x,y,z/x,y,z,w *float* fields. They are mostly used in the SMesh (Section 4.2.1) component to represent vertices' and triangles' properties;
- **SColor32**: a serializable version of Color32, with a,r,g,b *byte* fields, used to represent the Material's color and emission color (Section 4.2.2).

```
[Serializable]
public class SVector2 {
    public float x;
    public float y;
```

```
    public static implicit operator Vector2(SVector2 sv2) =>
        new() {
            x = sv2.x,
            y = sv2.y
        };

    public static implicit operator SVector2(Vector2 v2) =>
        new() {
            x = v2.x,
            y = v2.y
        };
}
```

**Listing 4.3:** SVector2 Serializable Type

# Chapter 5

# Server Side

The Server side of the framework rules the simulation on all the connected devices and it handles the communication with them, including the objects streaming phase. It loads the scene from the Server device memory and it starts listening for new connections (using the *Netcode* library). When a Client connects, the Server starts some loop cycle to constantly monitor the Client's position and consequently sends to it the needed objects. Moreover, the Server sends periodic updates if an object that is visible to one or more Clients, moves or changes the state of one of its properties.

Therefore, to avoid overloading or freezing the main Thread (the single one used by the Unity Engine to simulate the game physics and run the game logic), the Server makes use of additional Threads and Coroutines.

The **Coroutine** primitive [35] in Unity represents a block of code, whose execution can be arbitrarily paused by the engine and then restored in a different frame, with the result that the coroutine operations are spread to multiple frames and ran concurrently with the rest of the game logic. The Coroutine is usually used to run animations or other long operations that would otherwise block the execution of the main thread. A similar goal is achievable by using the Threads. The main difference is that the *Thread* primitive instantiates a new thread, which is handled directly by the Operative System. The threads, however, have an important drawback compared to the Coroutines: due to the non-thread-safety of the Unity API, the Unity objects cannot be accessed from threads different from the main one, where the Unity Engine executes.

For this reason, the only thread that has been used in the Server is the *HTTP Server*: an HTTP Listener that exposes the Objects Components to the Clients that require them. This thread is launched just after the instantiation of the Server from the *Netcode Network Manager*. This does not require accessing the Unity objects because, at this point, all the serialized components have already been stored in a shared *cache* static class.

The Server also uses the following Coroutines:

- **Send Enqueued Objects Loop**: as mentioned in Chapter 3, the objects' data are not sent directly to the Clients when they go close to them; instead, the object is added to a queue, specific for that Client (using the class *SendObjectQueue*) and periodically sent to the corresponding Client through the *Send Enqueued Objects Loop*. This allows to split the network load in a controlled flow, so as to

avoid congestions, especially in the initial loading phase, when a lot of objects need to be shared.

- **Client Position Monitor Loop**: the Server launches a loop cycle for each client connection, in order to track their position and keep them updated by sending the required objects. In the *Client Position Monitor Loop*, the server checks if the Client has come close to some new objects and if so, the object information is sent using the `Objects Spawner` class (Section 5.3). The distance control is performed using the Unity Physics function *Overlap Sphere* as explained in Section 5.2.1. The loop then adds to the *Send Objects Queue* the new objects to be sent to the Client.
- **Update Send Zones Radius Loop**: finally, to prevent network congestions, a zone-resize mechanism has been implemented. At specified time intervals (each 3 seconds in our configuration) the radius of the Overlap Spheres used to detect the objects near the Player (Section 5.2.1) are resized to lighten the transmission load, or to display more of the environment, depending on the network conditions. The resizing is performed ad-hoc for each client, based on the RTT (Round Trip Time) of the connection between that Client. The *Update Send Zones Radius Loop* is responsible for the resize operations. The formulas describing the radius change of each one of the three zones around the Player are the following:

$$Rad_0 = 0.7 * Last + 0.3 * \left( -\ln\left(\frac{RTT}{1000}\right) * 30 \right)$$
$$Rad_1 = Rad_0/2$$
$$Rad_2 = Rad_0/4$$

where *Last* is the previous Level 0 Zone Radius, $Rad_0$.

## 5.1 Server initialization

The first operation performed by the Server when it starts is to load all the current environment objects from the *Resources* folder into the scene. One key difference between the Client and the Server Projects is that the Client one does not contain any information about the objects that will be spawned. This allows to keep the Client application as light as possible, containing just the *scripts* to receive the information from the Server, the base elements of the scene, and the XR Interaction Toolkit used to control the Scene with the VR Headset and controllers.

The Server Project also needs to specify the parameters to be used for the simulation, before the compilation of the executable:

- all the *Netcode* and *Unity Transport* parameters. This includes:
  - the Server address and port;
  - the MTU (Maximum Transmission Unit): the maximum amount of Bytes that can be sent through RPC calls;
- the *HTTP Listener* address and port;
- the *Objects Send Frequency*: the frequency with which the initial information of the objects is sent from the Server to the Clients;

- the *Clients Update Frequency*: the time delay between each check of the Clients' position;
- the *Objects Update Frequency*: the delay between each update of the objects' properties to the Clients;
- the *Clients Zone Update Frequency* and the *Client Zones values* (its purpose is explained in Section 5.2.1).

For the implementation of the HTTP protocol it has been used the standard .NET HTTP Library [36], always running in background, on a parallel Thread. But, because of the thread unsafety of the Unity Engine Objects, we needed to store the Serialized Components into a temporary cache before sending them to the Clients. This operation is performed right before the enqueuing of the base objects' data for the delivery to the Clients, as described in Section 5.3.

## 5.2 Objects Streaming Priority

When a Player connects to the Server, a new *Player Prefab* is spawned. The *Player Prefab* is a GameObject, replicated in the Server and in each Client, that is automatically created and synced by the *Netcode* Library, using the selected prefab object: it represents the Player Avatar and it is controlled by the owner Client. In our case, the *Player Prefab* is composed of a head (which mirrors the head movements of the human player, using the sensors mounted on the HMD – Head Mounted Device) and two hands (controlled by the XR controllers). After the connection of the Player, the Server instantiates and attaches two objects to the *Player Prefab*'s head: the *Objects Detector* (Section 5.2.1), to detect the objects in the AoIs to send; and the *Frustum Collider* (Section 5.2.2), to detect the objects in the field of view to keep synchronized.

The *priority* with which the objects are enqueued is calculated using the importance precedence assigned to each object when the scene is created and the distance from the Player's Avatar. More precisely, the formula used for the *Priority* calculation is the following.

$$P = O_d(O_l + 1)$$

where:

- $O_d$ := Object Distance, the distance between the Player and the object.
- $O_l$ := Object Level, the importance level of the object.

### 5.2.1 Objects Detector

As previously mentioned, the Client receives by the Server only the objects that are more likely to be seen from the Player perspective. To optimize the User Experience but keep the usage of bandwidth as low as possible, the concept of *Draw (or Rendering) Distance*, widely used in the video game field, has been reused. It consists in hiding from the Player's field of view the objects that are far away from it. This technique is usually mitigated by adding some fog to contextualize the poor visibility.

In our project, this translates to the subdivision of the area surrounding the Player into three concentric spheres, as can be seen in Figure 5.1. The farther zone is used to send only the bigger objects, those that would be visible even from afar, such as

big buildings, mountains, and the ground. The middle zone detects the medium-sized objects as well (trees, cars, bushes, etc.). The closest zone represents the area in which every object is sent to the Client. To distinguish between important (or larger) and negligible (or smaller) objects, we use the `priority` field of the class *NetObject* (which is attached to every object that needs to be streamed by the system). At the current state, there are 3 different priorities: from 0 (most important) to 2 (less important).

We initially implemented a Collider component to detect the objects inside the zones, but in the end we choose to use the Unity Physics method *Overlap Sphere*. In Unity, a *Collider* is an invisible component that handles the collision between GameObjects through trigger events. A *Collider* has a shape that can be of a primitive type (box, sphere, and capsule), or it can precisely match a 3D mesh: this guarantees a higher fidelity, but it causes high processor overhead. On the other hand, the *Overlap Sphere* method returns a list of the objects in the spherical area of the specified radius, and there exist variants of the same primitive shapes.

Our framework can benefit from the use of the *Overlap Sphere*, given that it do not need the object detection to be performed at every frame of the simulation. For this purpose, it has been created a *PlayerObjectDetector* class in the Server, to represent each of the three zones around the Player. When the component is instantiated onto the Player's Avatar, it launches one *Objects Detection Cycle*: a coroutine that performs in loop the object detection operations, with a customizable time interval that is set at 2 seconds. Inside the loop, it is called the `DetectObjects` method (Listing 5.1) that, using the *Overlap Sphere* method, detects the changes of the objects in the zone and adds them to the *Send Queue*.

```
private void DetectObjects() {
    var t = transform;

    // Get colliding objects with right priority
    var colliders = Physics.OverlapSphere(t.position, radius)
        .Where(c =>
            c.TryGetComponent<NetObject>(out var o)
            && o.priority == _level)
        .Select(c =>
            c.GetComponent<NetObject>())
        .ToArray();

    var selected = colliders.Except(_previous).ToList();
    SendNewObjects(selected);
    var removed = _previous.Except(colliders).ToList();
    DeleteOldObjects(removed);

    // Update ‘previous‘ list
    _previous = colliders.ToList();
}
```

**Listing 5.1:** Objects Detection Cycle

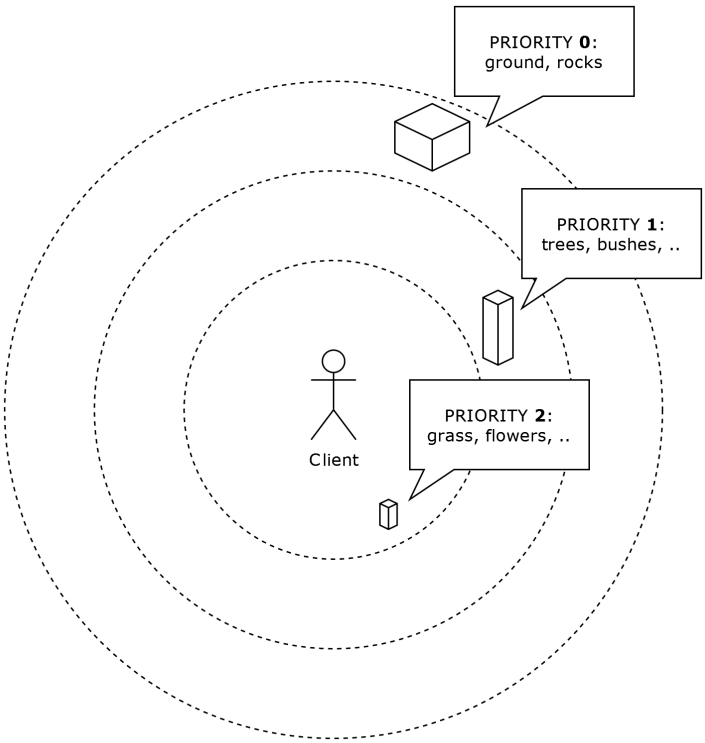



**Figure 5.1:** Client Sync Class Architecture

### 5.2.2 Frustum Collider

In geometry, the *frustum* is the basal part of a solid cone or pyramid, formed by cutting off the top by a plane parallel to the base. In 3D computer graphics, this shape represents the space region that may appear on the screen of the Player, it is the *field of view* of the *virtual camera*. We choose to use this concept to optimize the update of the objects' properties: when an object is not inside any of the clients' *frustum view*, it does not need to be updated, because no one will see its changes.

Just after the creation of the *Objects Detector* (Section 5.2.1), the Server creates a new GameObject as a child of the Player Prefab's head. It adds a *PlayerFrustumCollider* - that handles the operations to perform when an object enters or exits from the Client's *field of view* - and a new *MeshCollider*, with the shape of a frustum corresponding to the view, that detects the collisions with the objects. In order to detect the frustum corresponding to the camera view, it has been used an *extension method* of the `Camera` class, which makes use of the Unity Engine's `ViewportToWorldPoint()` function to detect the edges of the mesh.

The `PlayerFrustumCollider` class in Listing 5.2 shows that when an object comes in collision with *FrustumCollider*, its `NetObject.facing` property is increased by 1. In the same way, when it exits, the variable is decreased so as to keep in memory for each object how many Clients are looking at it. This information is then checked when an object changes some of its properties, in order to update only the visible ones, as described in Section 5.4.

The decreasing of the number of viewers has been delayed by 1 second, to avoid a bug in which a falling object was freezing in midair, from the Client's perspective, because it was exiting from the *frustum-view* area too early.

```
public class PlayerFrustumCollider : NetworkBehaviour {
    private void OnTriggerEnter(Collider other) {
        other.GetComponent<NetObject>().facing += 1;
    }

    private void OnTriggerExit(Collider other) {
        StartCoroutine(DelayedChangeViewers(other));
    }

    private static IEnumerator DelayedChangeViewers(Component
        other) {
        yield return new WaitForSeconds(1);
        other.GetComponent<NetObject>().facing -= 1;
    }
}
```

**Listing 5.2:** Player Frustum Collider Class

## 5.3 Server Objects Spawner

The Class `SpawnManager` is the heart of the Framework: it contains both the Server and the Client logic for, respectively, sending and receiving the *GameObjects* and attaching the corresponding components to them. The class must be shared by both the actors in order to communicate through the *Netcode's* RPCs. Indeed, the *Netcode* library allows only to send RPCs between two or more instances of the same shared object. The *SpawnManager* class is thus used as a "bridge" between the initial objects loaded on the Server and their replica that is progressively sent and built on the Client.

The main method, used by the Server side of the class is:

```
SendObject(ulong clientId, GameObject obj)
```

The `SendObject` method is called by the *Send Enqueued Objects Loop* if there are some objects left in the queue. Initially, the method creates a new `SObject` instance, starting from the given GameObject. As described in Chapter 4, `SObject` is the serializable object representation that is initially sent to the Client. The `SObject` is then inserted into a *Server Cache*, where it will be available for future re-send or updates of the same entity. The next step is to add the object's components to the Serializable Object. This is done by the `AddComponent` method of the *SObject class*: the AddComponent method takes as input a generic `SComponent` (the base class of all the Serializable Components) and adds it to the array `componentIds`, an array in the SObject that keeps track of all the components that will need to be linked to the object. More in detail, the components that are saved by the *SendObject* method, and the most important when dealing with 3D objects are:

- the **Mesh**: if the object contains a *MeshFilter* component associated to it, we can suppose that it uses a *Mesh* to render the shape of the object. The method uses the mesh to build a `SMesh` object, which is the serializable object that contains the mesh information. The SMesh is added to a Dictionary, where

the components to send via HTTP are stored. The temporary storage of the components is mandatory in order to allow the different Threads – like the HTTP one – to access the Unity Objects when required. In the end, the SMesh is attached to the `SObject` we just created;

- the **Materials**: a physical object is usually composed of one or more *Materials*, so the method iterates through all the materials connected to the *MeshRenderer* component and creates the corresponding `SMaterial` objects. As for the Mesh, the material components are saved in the HTTP Dictionary, and then attached to the `SObject`.

```
public void SendObject(ulong client, GameObject obj) {
    // Object
    var sObj = new SObject(obj);
    _serverCache.TryAdd(sObj.Id, sObj);

    // Mesh
    if (obj.TryGetComponent<MeshFilter>(out var mf)) {
        var sMesh = new SMesh(mf.mesh);
        componentServerHttp.AddComponentToServerCache(sMesh);
        sObj.AddComponent(sMesh);
    }

    // Material & Texture
    if (obj.TryGetComponent<MeshRenderer>(out var mr)) {
        for (var i = 0; i < mr.materials.Length; i++) {
            var material = mr.materials[i];
            var sMaterial = new SMaterial(material, i);
            componentServerHttp.AddComponentToServerCache(
                sMaterial);
            sObj.AddComponent(sMaterial);
        }
    }

    // Send
    SendClientRpc(client, sObj);
}
```

**Listing 5.3:** SendObject Method

## 5.4 Server Sync

To make the user experience more immersive, the VR Streaming Framework also includes objects sync functionality. The first and most important property that should be synchronized between the Clients is the position and rotation of the objects: in fact, if one Player handles or makes an object move, one would expect that the movement is also reflected to the other Players of the VR simulation. This should concern only specific objects, since the majority of the objects composing the environment, such as trees, cars, or buildings, are static and cannot be moved by the Players.

There are different strategies that allow to achieve a similar behavior, the most important distinction that can be made is between Server and Client Authoritative movements:

**Server Authoritative** indicates that the Server owns and controls the whole simulation, so it is the only process that can operate on the objects to make them move or to change some others properties; moreover the gravity or forces that affect the objects are also simulated by the Server application. With this type of authority setting, a Client that wants to move an object needs to send before a request to the Server, specifying the direction and amount of the movement;

**Client Authoritative** in contrast, means that the responsibility of the movements can be held also by the Clients. This strategy is usually in combination with the concept of *ownership* of the objects, permitting only the Client who is the owner to control it. If the owner Client wants to move an object, it can directly move it in its own local environment, then it sends an update message to the Server that will take care of syncing the movement back to the other Clients.

The *Server Authoritative* is more safe, especially for competitive multiplayer games, since it can prevent many cheating techniques by filtering the Clients' request. If a Player is trying to perform an operation that is unnatural or that is identified by the Server as impossible, it means that the Client has been compromised and the Server could eventually proceed to banish or penalize it. On the other hand, the *Client Authoritative* strategy does not offer any guarantee that the Clients are performing allowed operations, but the visual perception of the object's movement on the owner Client is smoother and more accurate: in fact, the Client can immediately process the local movement performed by the user, without having to await the Server confirmation.

Attempting to obtain the best of the two methods, the VR Streaming Framework uses a hybrid strategy that we call *Mixed Authoritative*: when the Player grabs an object, the Client autonomously acquires its ownership of it, then when it moves the object, the position and rotation updates are sent to the Server that will update the other Clients of the simulation as well. In contrast to the Client Authoritative strategy, the physics of the game is ruled entirely by the Server, so as to keep it consistent through the various Clients and make the objects' movements affect also the ones indirectly touched that are not owned by the Client, as can be seen in Figure 5.2.

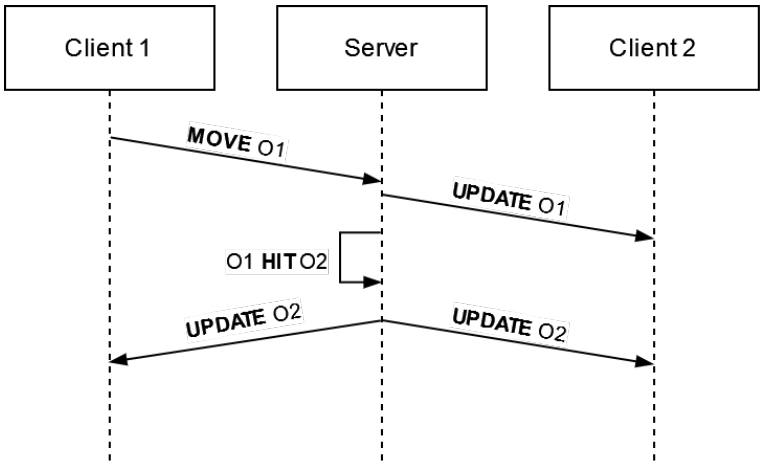



**Figure 5.2:** Object movement workflow (O1= Object1, O= Object2)

The other objects' properties, like the material color or the light emission, can instead

be only synced from the Server to the Clients. However, clients are able to send *signals* that trigger certain Server-side behaviors on the object, as explained in Section 5.4.3 . The activation workflow is illustrated in Figure 5.3.

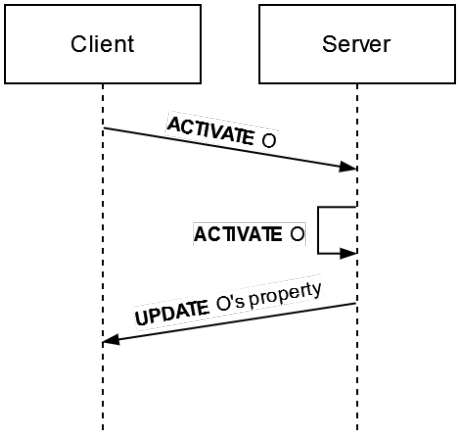



**Figure 5.3:** Object activation workflow (O= Object)

### 5.4.1 Properties Sync

The synchronization of the properties between the Server and the Clients has been standardized by using the same super-class for every property type. This allows to reduce the redundancy of the code that links the multiple instances of the same object between the devices, and most importantly, it makes it easier to further extend the sync behavior including other properties. Since update messages contain only a small amount of data, they need to be frequently sent and do not require a high level of reliability; the best protocol choice is to use the UDP protocol; in particular, the *RPC* primitive offered by the *Netcode* library has been used.

The classes `BaseSync` and `BaseSyncClient` represent, respectively, a synchronized property on the Server and on the Client that can be attached to a GameObject of the scene (the `BaseSyncClient` class is created and attached by the Client when it first receives the streamed object). The class `SyncManager`, instead, is unique and is used by both the Client and the Server. As happens with the `SpawnManager` class, the `SyncManager` acts as a "bridge" between the two devices and associates changes in properties with the corresponding objects, by using their `object-id`s. The workflow for updating the property of an object from Server to Clients is shown in Figure 5.4.

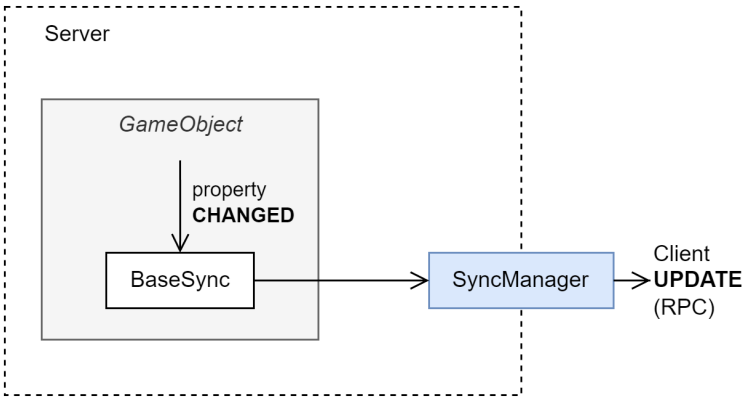



**Figure 5.4:** Object sync classes workflow, Server side

The `SyncManager` class (signature in Listing 5.4) is composed of:

- `RequireOwnershipServerRpc`: a method that the Clients use to acquire the ownership of an object when they grab it. Internally, the Server uses the method `LockGravity` to stop its gravity to affect the object while the Client is moving it;
- `SyncClientRpc`: the method used by the Server to send object updates to the Clients. The parameters in input to the method are as follows:
  - `int id`: the identificator of the object;
  - `SyncData data`: the updated information of the property that has been changed. The class `SyncData` is a serializable wrapper class, which contains only the functionalities for the json serialization of the sync data structure and that is used to send to the Clients the updated properties for each of the components managed by the framework;
- `SendSignalServerRpc`: the method that the Clients use to send activation signals relative to an object to the Server (see Section 6.2.3);
- `UpdateTransformServerRpc`: a special method to let the Client move the objects and consequently send update messages to the Server. The *Transform* updates of the movable objects are treated differently from the other properties: as explained in Section 5.4, the Clients can directly move the objects as for the *Client Authoritative* strategy and then inform the Server of the changes;
- `AskForUpdateServerRpc`: with this method, a Client can actively require the update of some object. This may happen if a Client connects to the simulation after some object's property has changed: in this case, the Server does not perceive any property change to notify, but the Client actively requires an initial sync of the objects as soon as it receives them;
- `SearchObject`: the utility function that searches for the required object in the scene, using the given `object-id` parameter.

```
public class SyncManager : NetworkBehaviour {
    // ---------------------- Sync Owner
    [ServerRpc(RequireOwnership = false)]
```

```
    public void RequireOwnershipServerRpc(int id, ulong owner);

    private static void LockGravity(Rigidbody rigidbody, bool
        locking);

    // --------------------- Sync Server to Clients

    [ClientRpc(Delivery = RpcDelivery.Unreliable)]
    public void SyncClientRpc(int id, SyncData data);

    // --------------------- Sync Client to Server

    [ServerRpc(RequireOwnership = false)]
    public void SendSignalServerRpc(int id);

    [ServerRpc(RequireOwnership = false)]
    public void UpdateTransformServerRpc(int id, SyncData data);

    [ServerRpc(RequireOwnership = false)]
    public void AskForUpdateServerRpc(int id);

    // --------------------- Other

    private static SObject SearchObject(int id, bool isServer);
}
```

**Listing 5.4:** Sync Manager Class

The `BaseSync` class is inherited by three classes, specific for each property synchronization:

1. `SyncColor`: synchronizes a material color of the GameObject's `MeshRenderer`;
2. `SyncLight`: synchronizes a `Light` component connected to the GameObject;
3. `SyncTransform`: synchronizes the `Transform` component of GameObject, composed of *position*, *rotation* and *local scale*.

When a `BaseSync` is launched, it initially creates a coroutine with the purpose of monitoring the target property, through the `UpdateConditions()` function. When the property changes, the Coroutine runs the overridden function `UpdateSync()`: each sync class creates a specific `SyncData` object to encapsulate the properties' changes, and it sends the update message to the Clients using the `SyncManager`.

### 5.4.2 Transform Sync

A different approach is used to synchronize the position and rotation of objects. In fact, the `SyncTransform` class can operate in two different ways:

- the synchronization from the Server to the Clients is the same as the other components. Using the same mechanism explained above, in Section 5.4.1, the *Transform* values are sent to the Clients through the `SyncManager` class;
- on the other hand, the synchronization of the transform properties in the opposite direction, from the Clients to the Server, is also allowed. To allow the Clients to move the objects, the Server performs the following operations, as illustrated in Figure 5.5:

1. the Server grants the ownership of the object to the Client, through the method `RequireOwnershipServerRpc()` of the `SyncManager`;
2. disables the physics of that object, as to avoid multiple concurrent forces to be applied on it;
3. when the RPC method `UpdateTransformServerRpc` on the class `SyncManager` is called by the Client, the Server calls the `UpdateServerTransform` on the object's `SyncTransform`;
4. the `UpdateServerTransform` method uses the given `TransformData` to update the object's position and rotation through its *Rigidbody*;
5. the Server's `SyncTransform` loop will detect the new movement (as described in Section 5.4.1) and send updates to the other Clients as well.

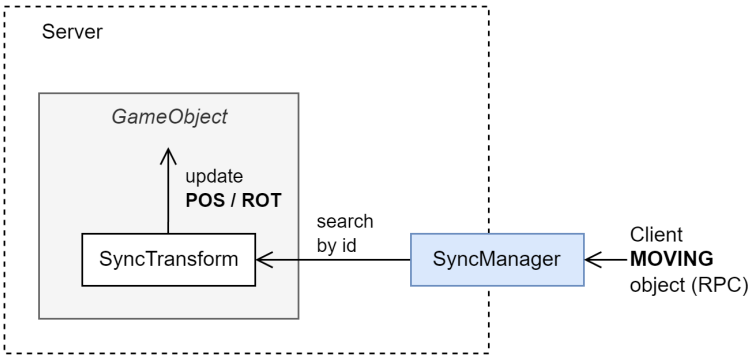



**Figure 5.5:** Transform sync workflow, Server side

### 5.4.3 Objects Actions

To make the simulation even more interactive, we added the option for the Client to activate some objects of the scene and consequently trigger some changes in the objects state and that may be visible by the other connected Clients. A typical example of object activation is the switching on/off of a torch.

To obtain the desired behavior, the framework needs to include some sort of active interaction mechanism from the Client. This requires the creation of a bidirectional communication channel so that the Client can send a signal message to the Server whenever it requires the activation of an object. This method is not optimal in terms of perceived responsiveness for the user that triggers the action because before the action is actually performed on the Client's scene, the signal needs to be received by the Server and sent back to all the connected Clients. However, in this way it is guaranteed the Server authority, making possible to apply some additional verification before activating the object's behaviors required by the Clients.

A similar mechanism to the one that allows the properties to be synced has been used to let the Clients activate the objects of the scene.

A Client can inform the Server of object activation with a *Server Action RPC*, containing the *id* of the object to activate. When the Server receives the call to the RPC method, it searches the current scene for the given `objectId`; then, if any `IAction` is connected

to the GameObject, it calls its `Activate()` method. The concrete implementations of the actions will finally trigger the property change on the object (in the aforementioned example, the `ActionLight` - Listing 5.5 - will turn on or off the light of the torch). The `Action` components are classes on the Server, attached to the objects, which produce a behavior related to the object when they receive an *activation signal*. The server activation workflow is illustrated in Figure 5.6.

```
public class ActionLight : IAction {
    public new Light light;
    public float intensityOn;

    private bool _state;
    private float _intensityOff;

    private void Start() {
        if (light == null) {
            light = GetComponent<Light>();
        }

        _intensityOff = light.intensity;
    }

    public override void Activate() {
        light.intensity = _state ? _intensityOff : intensityOn;
        _state = !_state;
    }
}
```

**Listing 5.5:** ActionLight Class

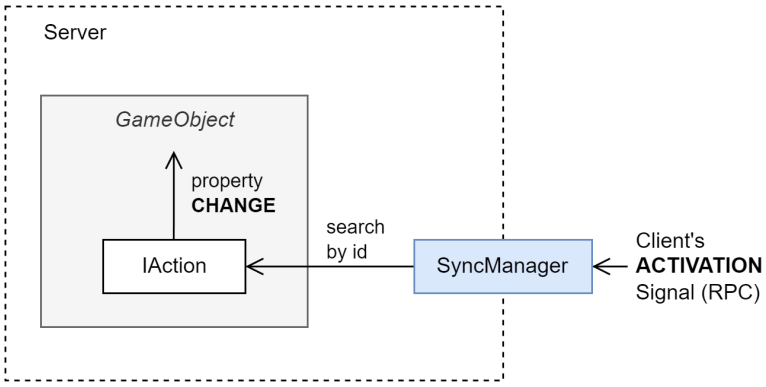



**Figure 5.6:** Object Activation workflow, Server side

# Chapter 6

# Client Side

The purpose of the Client devices is to run the simulation from the users' perspective, so as to allow the players to move and interact with the environment hosted by the Server. In other words, the Client is an application on the user device (in our case, the Meta Quest 2) that acts as an interface between the player and the Server.

The Client-side application initially is empty of every scene object; it only contains some base elements and the components to connect to the Server. The most important elements are:

- the main directional source of light (simulating the Sun). For the purpose of this work, the concept of day-night cycle has not been included, so the main light is static and not controlled by the Server;
- the in-game GUI (Graphic User Interface), in the form of an interface control panel used to connect to the Server;
- the *NetworkManager* entity, to initialize the connection through the Netcode library;
- the *SpawnManager*, to receive the objects and components from the Server;
- the *SyncManager*, to receive the objects properties updates from the Server;
- the *ComponentClientHttp* object, to handle the http connections;
- all the objects and components required to use the *XR interaction toolkit*, better described in Chapter 7.

The tasks that the Client performs during its lifespan can be summarized in four points:

1. receiving the initial object description from the Server;
2. downloading from it the components required for the object;
3. receiving and applying objects' properties updates from the Server;
4. sending position and rotation updates to the Server when the user grabs and moves an object.

After the user has pressed the corresponding button on the control panel, the Client device connects to the Server, if it is running. The *Netcode* library handles the initial connection phase to the Server, and provides automatic synchronization of the Player's

Avatar. More in detail, the position and rotation of the head and the hands are synced, as well as the animation of the hands' fingers when grabbing or operating. The avatar's head position (which corresponds to the user's HMD) is used by the Server as a central point to initialize the three *send zones* (Section 5.2.1) that the Server uses to create the streaming priority queue.

## 6.1 Client Objects Spawner

The `SpawnManager` class is used by the Client to retrieve and build the scene that has been streamed from the Server. The `SObject` structure is used to encapsulate the base information of the object and is sent to the Clients through Netcode RPC with the method `SendClientRPC()`.

The first operation that the Client performs, after receiving from the Server the RPC method call containing the initial information of an object (Section 4.1), is to check for the presence of the object in the Client's objects cache that corresponds to the objects in the local scene. It may, indeed, happen that the User is returning to an area that he has already visited and which objects have already been downloaded: in this circumstance, we do not want to download the object's components again, so as to spare bandwidth.

If the object is not present in the cache, the Client instantiates a new one, through the `BuildObject()` method of the `SObject` class. As can be seen in Listing 6.1, the method creates a new object using the name, tag, and *Transform* parameters saved within the structure, and if some of the *Sync Options* is enabled, it creates and adds to the object the appropriate *ClientSync* component classes.

Once the object is created, the Client adds the other needed Unity Components, namely the *MeshRenderer*, the *MeshFilter* and the *MeshCollider*. In addition, it adds the *NetObject* component and, if the object can be moved by the user, the *Rigidbody* and *HandsInteractable* components.

Finally, the Client starts the procedure to download (if not already present in cache) each component of the object, using its *Component IDs* list. The component cache is a dictionary, indexed by the *id* of the component that contains the `SComponent` object itself, that is, the serialized object downloaded via HTTP. If the *component id* is not found in the cache, it is downloaded using the *async* method `GetComponentFromHttpServer()` of the `ComponentClientHttp` class. The asynchronous method guarantees that the downloading of the component does not interfere with the main thread, allowing the user to continue moving or looking around without freezing. Inevitably, the initial download phase still produces some lag in the Client device, because it needs to simultaneously download, build, and render all the components of the objects composing the initial AoI (more details about the solution's performance are discussed in Chapter 8).

```
[ClientRpc(Delivery = RpcDelivery.Reliable)]
public void SendClientRpc(ulong Client, SObject sObj) {
    if (NetworkManager.Singleton.LocalClientId != Client) return;

    // Get or build Object
    if (TryGetFromObjectCache(ref sObj)) return;
    var obj = sObj.Obj;
```

```
    // Add NetObject (and eventually Sync)
    var netObj = obj.AddComponent<NetObject>();
    netObj.Setup(sObj.Priority, sObj.IsMovable, sObj.Id);

    obj.AddComponent<MeshFilter>();
    obj.AddComponent<MeshCollider>();
    obj.AddComponent<MeshRenderer>();

    if (sObj.IsMovable) {
        obj.AddComponent<Rigidbody>().isKinematic = true;
        obj.AddComponent<TwoHandsInteractable>();
    }

    // Attach Components
    foreach (var id in sObj.GetComponentIds()
                .Where(id => id != 0)) {
        AttachComponentToObject(id, sObj);
    }

    sObj.Obj.SetActive(true);
}
```

**Listing 6.1:** SendClientRPC Method

For this thesis work it has been implemented a restricted set of components, the Mesh and the Material: they are the least pieces of information necessary to create and display the objects, but the flexibility of the framework allows to easily add more of them in future iterations of the project. With regard to reconstruction and attachment of components to the object, the `AttachTo()` method of the `SComponents` classes is implemented as follows, using the data saved in the serialized component itself:

**Mesh:** a new Unity Mesh component is created using the name and the rectangular bounds of the object. After that, the *vertices*, *normals*, *tangents*, *uvs*, and *triangles* arrays are configured, parsing the serialized arrays used to send them. The newly created mesh is applied to both the *MeshFilter* and *MeshCollider* components of the object, yet the collider is made *convex* if the object is movable, because of Unity physics limitations due to performance reasons. If the object is movable, the collider is also added to the list of colliders in the `XRGrabInteractable` object.

**Material:** a new Unity Material component is created using the standard shader and the name of the component. The material *mode*, the *cutoff*, the *color*, the *emission* and the *emission color* are configured. All the textures saved in the structure are deserialized and applied to the material. Finally, because *MeshRenderer* can contain multiple materials, the newly created material is inserted in the correct position of the material array of the renderer.

As soon as the components are downloaded, they are gradually applied to the object, without awaiting the delivery of all of them. This choice allows the user to have an early sense of the object's presence even though, for example, the texture's download has not yet been completed. This may produce odd artifacts, like objects with a magenta flat texture (the default color for objects without texture); their impact could be limited, for example, by sending an average color alongside the mesh, to temporarily fill the object while waiting for the actual texture to be downloaded.

## 6.2 Client Sync

As explained in Section 5.4, the second most important functionality of our framework is to synchronize certain component's properties from the Server to all the Clients, with the only exception of the `SyncTransform` component, which can synchronize its properties in both directions.

### 6.2.1 Properties Sync

Each set of properties of an object that can be synchronized through the devices is encapsulated in structure classes, which are themselves handled by the `SyncData` class. The purpose of `SyncData` is to serialize and deserialize the data structures when required by the framework, parsing them into JSON strings that can be sent through Netcode RPC. An example of a data structure is `LightData` (Listing 6.2), where the fields represent the corresponding new values of the light component of the object.

```
[Serializable]
public struct LightData {
    public float intensity;
    public SColor32 color;
    public LightType type;
}
```

**Listing 6.2:** LightData struct

The Client device receives updates of the properties from the Server using the *Netcode* RPC method `SyncClientRpc` of the `SyncManager` class (Listing 5.4), in the form of `SyncData`. Then it searches for all the `BaseSyncClient` components attached to the object, and it calls the `TryApplySync` method (Listing 6.3) on each of them, passing as input the received data. The `TryApplySync` method is overridden by each Client's sync method, which checks if the data are formatted using the correct data structure. If so, the methods apply the parameters contained in the structure to the object's component. The object update workflow is illustrated in Figure 6.1.

```
public override bool TryApplySync(SyncData data) {
    if (data.Data is not LightData d) return false;

    var l = GetComponent<Light>();
    l.intensity = d.intensity;
    l.color = (Color32) d.color;
    l.type = d.type;
    return true;
}
```

**Listing 6.3:** TryApplySync method

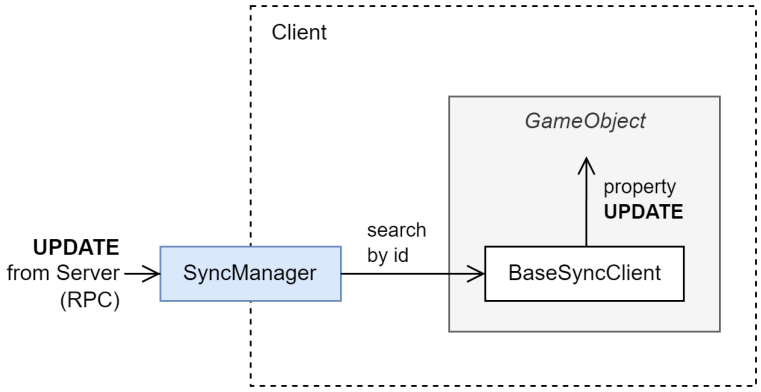



**Figure 6.1:** Object sync classes workflow, Client side

### 6.2.2 Transform Sync

This section describes the Client side of the *position* and *rotation* updates of an object, which are performed by Clients to the Server, when they interact with a movable object.

The `SyncTransformClient` class, other than updating the *Transform* properties when it receives new updates from the Server, starts a loop with the purpose of observing the movements of the local object, if the Client owns it. The *Transform* update workflow, summarized in Figure 6.2, is the following:

1. when the Player grabs an object that is movable, the `XRNetworkGrabInteractable` class requires to the Server the ownership of that object;
2. from then on, the movements of the object are observed by the Coroutine in the `SyncTransformClient` class of the Client;
3. when the *position* or *rotation* values of the object's *Transform* change, a `SyncData` object is created, containing a new `TransformData` data object;
4. then the `SendSignal` method of the same class calls the `UpdateTransformServerRpc` RPC method on the `SyncManager` class, which will be executed on the Server, updating the object's properties as described in Section 5.4.2.

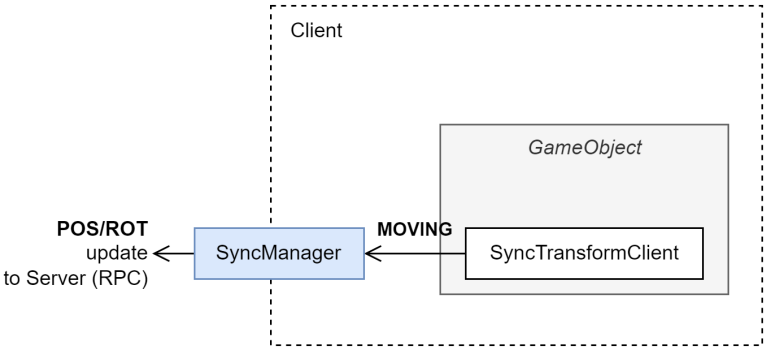



**Figure 6.2:** Transform sync workflow, Client side

### 6.2.3 Objects Actions

Clients who want to activate an object can do so by pressing the trigger button on the XR controller while grabbing the object. The button press is intercepted by the `XRGrabNetworkInteractable` class (explained in Section 7.2), which sends an *activation signal* to the server, calling the method `SendSignalServerRpc(id)` of the class `SyncManager` (Listing 5.4), with the object id as an input parameter. The activation workflow on the Client side of the framework is illustrated in Figure 6.3.

The *Object Actions* can be launched from the Client while holding any movable object of the scene. In fact, any of them has a `XRGrabNetworkInteractable` component connected that synchronized the object's movement with the Server. However, not every movable object will trigger an action, but only those with a `IAction` implementation connected to the object on the Server side of the framework.

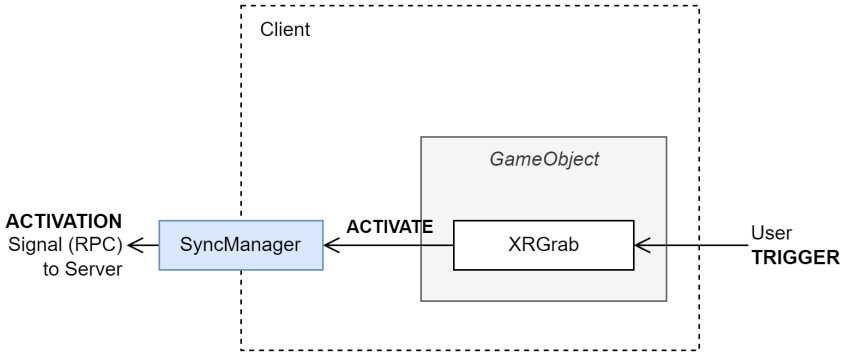



**Figure 6.3:** Object Activation workflow, Client side

# Chapter 7

# VR Interactions

This chapter covers a fundamental aspect of this thesis project that has been left aside until now: the integration with the Meta Quest device that makes it possible to interact with the environment using HMD and motion controllers. Unlike desktop games or 3D applications, developing a VR Client app involves some additional challenges.

- The Avatar synchronization, in our case limited to the hands and head of the player, must reflect one-to-one the movements performed by the user, while in a classical 3D game, only the major Avatar animation triggers need to be shared (e.g. "Start walking");
- the character movement must be controller through teleportation or continuous movement;
- the Client application must take into account the higher field of view of the Player;
- the framework must handle the physical interactions with the objects and synchronize back the movements with the other Players.

## 7.1 XR Interaction Toolkit and Controls

The Unity Engine [8] offers different powerful tools to easily integrate virtual and augmented reality capabilities into Unity projects. The most important one is **XR Interaction Toolkit** [21]: a high-level interaction system that allows 3D and UI interactions from Unity input events. XR Interaction Toolkit contains a set of components that support the following interaction tasks:

- Cross-platform XR controller input;
- basic object hover, select and grab;
- haptic feedback through XR controllers;
- visual feedback (tint/line rendering) to indicate possible and active interactions;
- basic canvas UI interaction with XR controllers;
- a VR camera rig for handling stationary and room-scale VR experiences.

Another useful plugin provided by Meta, specifically for Oculus devices, is the **Oculus Integration SDK** for Unity [37]. The Oculus Integration contains the following:

- **Audio Manager**: contains scripts to manage all the audio and sound effects in the applications.
- **Avatar**: contains the scripts and prefabs to add Oculus Avatars to the apps.
- **Interaction SDK**: a library of modular, composable components that allows developers to implement a range of robust, standardized interactions.
- **LipSync**: contains a set of plugins and scripts that can be used to sync Avatar lip movements to speech sounds.
- **Platform**: contains the components to add the Oculus Platform Solutions functionality to the apps. Platform Solutions include Leaderboards, Achievements, Cloud, P2P Connection, and many other individual components that can be independently integrated in projects.
- **Sample Framework**: this folder contains samples that demonstrate the core concepts of the Unity Integration.
- **Spatializer**: contains the scrips and plugins to add sound sources and make them sound as though they originate from a specific desired direction.
- **VoiceMod**: contains a plugin and set of scripts used to modify incoming audio signals.
- **VR**: this folder contains the Oculus VR utilities, a set of scripts, and prefabs to enable VR development.

For this thesis project, only the general *XR Interaction Toolkit* has been used, because the simplicity of the VR part of it did not require all the advanced functionalities offered by the *Oculus Integration* plugin. Future iterations of the project that will aim to be more interactive from the virtual reality perspective may also include this powerful tool offered for the Meta Quest devices.

### 7.1.1 XR Origin

The *XR Origin* is the fundamental component for controlling the camera and the position and rotation of the Avatar throughout the scene. The component is included in *XR Interaction Toolkit* and can be inserted into the scene from the Unity Editor. Its structure is composed as follows:

- **Camera Offset**: is the main component which represents the whole Player's body; in fact, its position can be regulated to match the height of the Player.
  - **Main Camera**: this object contains the main camera of the scene from the user point of view and includes the *Tracked Pose Driver* component that links the camera with the HMD and allows the Player to look around the virtual scene by moving and rotating its head through the head mounted visor.
  - **Left/Right Hand Controller**: as the name suggests, the controllers allow the Player to use the virtual hands in the simulation to grab or interact with objects, controlling them with the motion controller. By default the *Hand Controllers* do not include any 3D model that makes them visible, but only a colored ray to interact with the graphic interface of the games. To make it more entertaining and precise to interact with 3D objects, we added to our project the *hands presence* (Section 7.1.2).
- **Locomotion System**: this component is not included by default, but is available

in *XR Toolkit*. The *Locomotion System* has been added to let the Player walk in the virtual environment, controlling the movements through the controllers' joysticks.

### 7.1.2 Hands Presence

The framework includes a virtual representation of the Player's hands. The hand models have been introduced to provide a visual clue when interacting with the objects on the scene. The hands assets, so the 3D models and the animations, are taken from *Oculus Interaction SDK*, while the script to link the hands behavior to Player movements through the controllers is inspired by the Valem tutorials on YouTube. The script that controls the presence of the hands is composed of an *initialization phase*, in which the hands models are instantiated into the scene, as child component of the **Left/Right Hand Controllers**, so as to follow the position and rotation of the motion controllers; and an *update phase*, where the values of the *Trigger* and the *Grip* animations are settled, using the pressure values of the physical buttons (the model and the animations are shown in Figure 7.1).

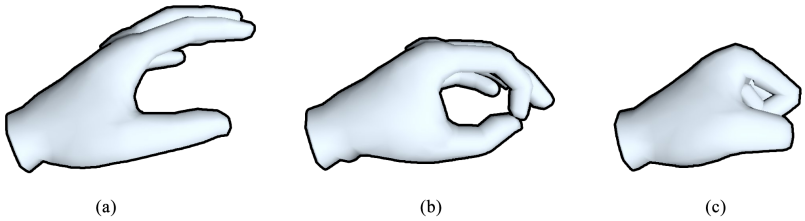



**Figure 7.1:** Hand Presence Model. Respectively: (a) Idle hand animation; (b) Trigger hand animation; (c) Grab hand animation.

Alongside the `HandPresence` script, it has been created `NetworkPlayerMap`: a class with the purpose of mirroring the movements and animations of the hands to the network Avatar objects, in order to keep them synced with the Server and the other Clients. The `Start` call verifies that the controllers are connected and hides the local copy of the hands, to avoid displaying two overlapped pairs of hands. Then, for each frame of the simulation, the position of the head and the hands are copied to the network object, while the hands' animations are synced only if the values exceed the `updateTolerance` parameter.

The animation synchronization functions are performed using a *Server RPC* method of the `NetworkPlayerMap` component itself, which is attached to the Player's Avatar. Being children of the *Player Prefab* object, the hands and head objects are automatically owned by the Client, so it can freely control their properties.

## 7.2 VR Object Grab

The Unity *XR Interaction Toolkit* provides a default functionality to let the user interact with the objects in the scene by adding the component `XRGrabInteractable` to the objects. The component allows basic interactions with the object and offers some settings to control the physics behavior when grabbing or leaving it. However,

our framework needs to perform additional operations in order to keep the objects synchronized; for this reason, we created the `XRGrabNetworkInteractable` class, extending the `XRGrabInteractable` one. The class performs the following additional operations:

- In the `OnSelectEntered` method, in addition to grabbing the object, it is called the `SetOwnership` method for each `BaseSyncClient` subclass found connected to the object. This triggers a request sent to the Server for assigning the ownership of the object to the Client. When the ownership is obtained, the object will automatically update its *position* and *rotation* from the `TransformSyncClient` component, as described in Section 6.2.2.
- In the `OnSelectExit` method, the ownership is released, and the Server reassigns it to itself.
- In the `OnActivated` method, which corresponds to the *trigger* button pression while holding an object, the *activation signal* (explained in Section 5.4.3) is sent to the Server.

## 7.3 Two Hands Interactable

The *XR Interaction Toolkit* has limited functionalities with respect to the *Grab* interactions . An example of such limitations is the possibility to grab objects only with one hand at a time, making it impossible to perform a series of natural interactions with some of them, like using a golf club, a two-handed weapon, a steering wheel, etc. To solve this questionable problem, we further extended the `XRGrabInteractable` class, making a new subclass of `XRGrabNetworkInteractable`, because we needed to maintain the network capabilities of the objects. The new class is called `TwoHandsInteractable` and, in summary, offers the functionality of grabbing the object with a second hand, while holding it with the first and rotating the object as if it were anchored to both hands.

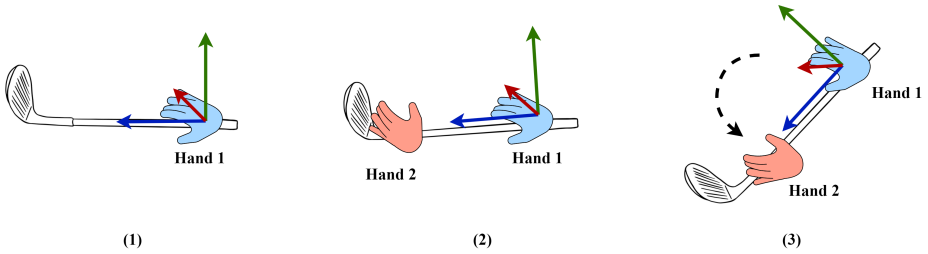



**Figure 7.2:** Two Hands object grab

The additional operations performed by `TwoHandsInteractable` to achieve the aforementioned result are summarized in Figure 7.2.

First, we need to introduce the concept of *Pivot Object*, also called *Attach Transform* in the context of *XR Interaction Toolkit*: it is an empty object (without any component other than the *Transform* component) that the main object uses to correctly move

and rotate, mirroring its movements.

1. When the first hand approaches the object and selects it (thus when the `OnSelectEntered()` method is called):
   - the *Pivot Object* position and rotation are set to the hand's position and rotation values;
   - the *Pivot Object* is set as parent of the object, so as to make it follow the *Pivot* rotation without distortions. If the *Pivot* is set as child component, the dynamic rotations of an object with the *Local Scale* value different from (1,1,1) will produce a mesh distortion;
   - in every successive frame, the *Pivot Object Transform* is updated to reflect the hand's rotation and position: in this way, the object moves naturally, following the hand's movements and using the point where the hand grabbed it as *Attach Point*.
2. When a second hand enters and selects the object, the *Pivot Object*'s rotation is set to the angle between the first hand's position and the second hand's position.
3. In the successive frames, while both hands are grabbing the object, the rotation of the *Pivot Object* is updated with the angle between the first and the second hand's positions: this ensures that the object does not follow the initial hand's rotation, but it stays aligned with both hands.
4. When one of the two hands releases the object, the scenario is restored to the initial state:
   - the object is set back as parent of the *Pivot Object*;
   - the only hand left is set as *fist hand*, even if it was the last to grab the object;
   - the *Pivot Object*'s *transform* is copied from the new first hand;
   - the *Pivot Object* is set again as parent of the object.
5. When the last hand leaves the object as well, the *Pivot Object* goes back to being a child of the main object.

# Chapter 8

# Experiments

This chapter contains a description of the experiments performed to test the performance and advantages of using our solution. In particular, the framework as described in this thesis work has been compared to a version of the same application where the Server does not perform any kind of selection or prioritization before streaming the objects.

Section 8.1 describes the configuration of the tests, Section 8.2 outlines the major problems encountered in the design and execution of the experiments, and Section 8.3 illustrates and discusses the results of the experiment.

## 8.1 Testbed

The testbed uses the following hardware components:

- **The Server**: in our specific case, the Server instance of the framework executes on a desktop computer with:
  - 16GB of Installed Physical Memory (RAM);
  - Intel(R) Core(TM) i5-10400 CPU, 2.90GHz, 6 Cores, 12 Logical Processors;
  - NVIDIA GeForce RTX 3060;
  - Samsung 980 NVMe M.2 SSD, 500 GB;
  - Microsoft Windows 11 Pro.
- **The Client** device, a Meta Quest 2 HMD, with 128GB internal storage;
- **The Access Point**: the Server device is connected to the AP through an Ethernet cable, while the Meta Quest 2 headset is connected to it through WiFi.

On the software side, while most of the experiment logs are embedded within the project application, some metrics require external tools to be measured:

- **NetLimiter** [38]: a software-only traffic shaping tool, monitoring, and firewall software for the Windows operating system. It is installed on the Server device, and has the purpose of limiting the bandwidth that the Server-side of the application can use;
- **NetBalancer** [39]: a Windows application for local network traffic control and monitoring. It is used to monitor and save network traffic logs for the Server application;

- **OVR Metrics Tool** [40]: provides performance metrics for Quest applications running on Android devices, including frame rate, heat, GPU, and CPU throttling.

The configuration parameters of the desktop version of the application can be completely configured, launching the executable with the appropriate command-line arguments.

`-mode <server/client>`
: Allows to select whether the application will be launched in Server or Client mode.

`-no-priority`
: Disables the priority features of the framework.

`-delay <delay value>`
: Allows to set a custom time delay for the objects streaming to the Clients.

`-time <time value>`
: Allows to set a timeout to automatically close the simulation.

`-show`
: Displays objects of different precedence indexes in different colors.

We performed two different kinds of experiments: the first one, illustrated in Figure 8.1, includes the actual Meta Quest 2, but it requires to manually transfer the logs from the device after the conclusion of the test time. The second experiment is fully automated, but it executes both the Client and the Server instances of the framework on the same desktop device, simulating the VR application through a functionality offered by *XR Interaction Toolkit*; with this strategy, however, it is not possible to register accurate measures of network traffic and the performance of the VR device.

- The *Quest* tests use the complete setup described above, which includes both the Desktop Server and the Meta Quest 2 device. This test is used to measure the traffic on the network between the two devices and the performance metrics of the Quest (*Frame Rate* of the app, *CPU* usage, *GPU* usage).
- The *Local* tests use only the desktop device to run both instances of the framework. These tests are used to save screenshots of the Client's view.

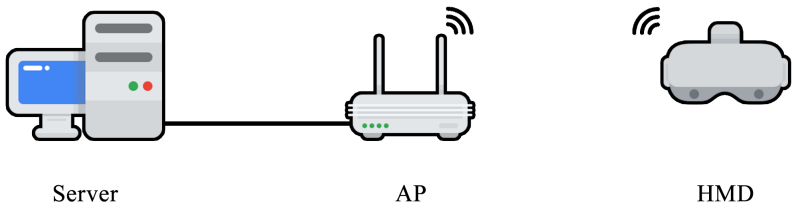



**Figure 8.1:** Testbed configuration (icons by anywatt, Flaticon.com)

Each set of tests is performed both in a *Standard* version - which sends the whole scene as soon as the Client connects to the Server, and in a *Priority* version - so using our solution described in Section 5.2. The tests are then repeated three times, one for each preset bandwidth limit (Table 8.1):

1. The first bandwidth limitation simulates an old DSL domestic connection;
2. the second one is the most common intermediate solution, composed of a mixture

of optic fiber connection and copper cable;

3. the third configuration does not make use of any limitation, so the full bandwidth available in the internal LAN, between the Server and the Client can be used.

For the first two experiments (*DSL* and *FTTC*), the *Objects Send Frequency*, defined in Section 5.1, is set to 60 ms; whereas, to exploit most of the available bandwidth, the *Unrestricted* setup does not use any delay, leading to the transmission of one object of the queue for every frame that the Server manages to render.

| Test | Bandwidth | Send Frequency |
|---|---|---|
| DSL | 4 Mbps | 60 ms |
| FTTC | 20 Mbps | 60 ms |
| Unrestricted | ∼200 Mbps | 0 ms |

**Table 8.1:** Tests Bandwidth Limits

## 8.2 Problems Encountered

### 8.2.1 Metrics Choice

One of the first issues encountered in the design of the experiment set is how to numerically measure the perceptive benefits that the user would experience by using our priority solution instead of a more standard approach, in which the elements of the scene are all sent sequentially to the Clients.

The chosen *perspective* metrics aim to calculate the difference between the Client's viewpoint of the scene and the complete scene that a Player would display if the assets were contained into the Client device itself; that, in our framework, corresponds to the last viewpoint of the Client, when the scene is fully downloaded and rendered. The three metric used are: *HDR-VDP* [11], *Delta E 2000* [10] and *Pixel Ratio*. The first two metrics have been described in Section 2.2, the *Pixel Ratio*, instead, is a metric that we originally designed to achieve the same result, which we then decided to support with more documented metrics.

### 8.2.2 Client Disconnections

The *Objects Send Frequency* is the time interval the Server uses to send the initial pieces of information of the objects contained in the queue to the Clients. The delay has been introduced as a safety measure to avoid overloading the Client devices when deserializing a large number of objects in a short amount of time. Moreover, it prevents congestions and, therefore, premature disconnection of the Clients. In fact, the *Netcode* library is not able to keep the connection active, probably due to the clogging of the network caused by the excessive HTTP flow.

Since the *Objects Send Frequency* is set to 0 ms for the *Unrestricted* experiments, meaning that an object is sent every frame calculated by the Server, it has been necessary to increase the delay to 60 ms for the *DSL* and *FTTC* experiments to avoid constant Client disconnections.

### 8.2.3 Client RTT Freeze

Another problem faced when designing the experiments is related to the *Round Trip Time* (RTT) value. The RTT is essential for the functioning of the framework, because it is used to dynamically resize the *AoI zones* around the Players. As described in Chapter 5, the three circular zones identify which objects need to be enqueued and sent to the Player and which ones must be ignored because too far away from it.

Due to internal design choices of the *Netcode* networking library, if the Client's avatar stands completely still for more than a few seconds, the RTT values measured by the Server (relative to the connection with that particular Client) cease to be updated, keeping the RTT to the last valid measured value. This would not be a problem in a realistic simulation scenario, where the user would look around and move their head or hands, but it represents an issue for the performed test, where the avatar is left steady to capture the screenshots from the exact same point of view at each instant.

The problem has been easily overcome by adding a check function on the Server. When it detects a repetition of the Client's RTT value, it automatically sends a command to make the Client move its avatar in an imperceptible way to the human eyes, but sufficient to trigger an update on the Server side of the framework.

## 8.3 Experiment Results

### 8.3.1 Network Traffic

The network traffic transmission rate has been measured as described in Section 8.1. In Figure 8.2 and Figure 8.3 it can be observed that both configurations with or without our priority system use approximately the same amount of total bandwidth to send the required data, since the overall information sent of the scene is the same. In fact, the priority send method starts the simulation using AoIs of small radius, but it dynamically adjusts the size of the zones whenever the RTT value becomes lower (as discussed in Section 5.2.1), leading at the end of the test simulation to zones that cover almost the entire area of the scene.

Note that only the first time a new asset is sent to the Client, a considerable amount of bandwidth is used, while the successive objects that need those assets will use the already-cached components. This particular behavior causes peaks of higher bandwidth usage, delayed compared to the first peaks, in the experiments using the *Standard Setup* (Figure 8.2):

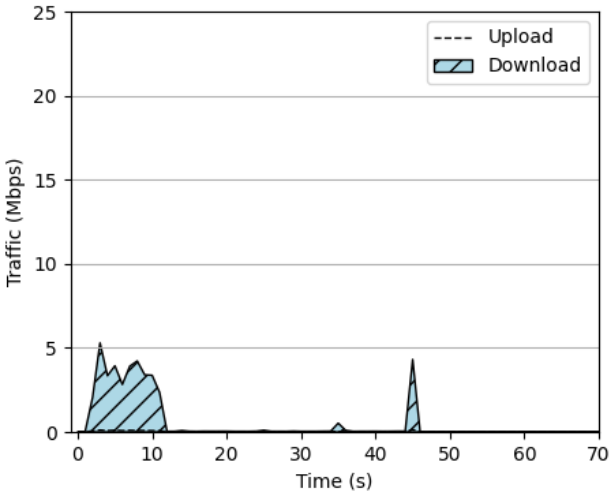
- At the beginning, the Server sends via HTTP most of the required assets, as soon as objects that use them need to be streamed to the Client;
- then there is an intermediate phase in which only objects with assets already in cache are sent (approximately between 10 s and 45 s, in the DSL and FTTC experiments);
- in the end, the Client needs to render a group of objects of which it has no cached mesh or materials, and this produces another HTTP streaming flow.

The fact that the same trend is not encountered in the *Priority Setup* (reported in Figure 8.3) is unintended and depends solely on the specific arrangement of the objects in the scene. The traffic in upload, so from the Client to the Server, is much less noticeable, given that it only consists of Client's avatar movement updates or *keepalive* messages exchanged by the *Netcode* library.

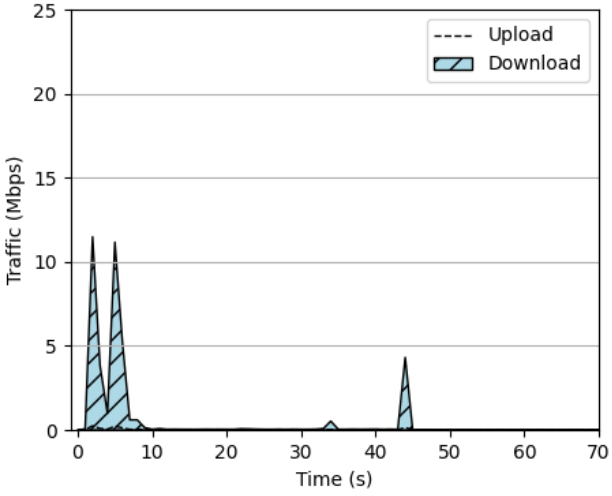
The Table 8.2 shows the cumulative amount of data downloaded by the Client during the simulation and the average data transmission rate.

| Test | Cumulative (S) | Average (S) | Cumulative (P) | Average (P) |
|---|---|---|---|---|
| **DSL** | 5.14 MB | 0.58 Mbps | 5.45 MB | 0.61 Mbps |
| **FTTC** | 5.07 MB | 0.57 Mbps | 5.00 MB | 0.56 Mbps |
| **Unrestricted** | 6.05 MB | 0.67 Mbps | 5.51 MB | 0.59 Mbps |

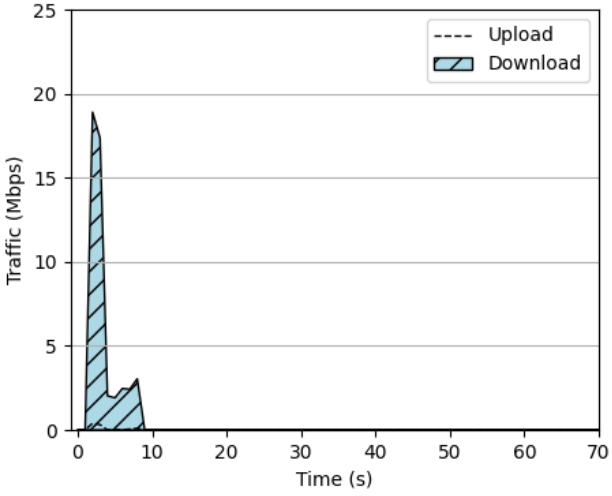
**Table 8.2:** Download Cumulative Traffic and Average Transmission Rate. (S)=Standard, (P)=Priority




*DSL*

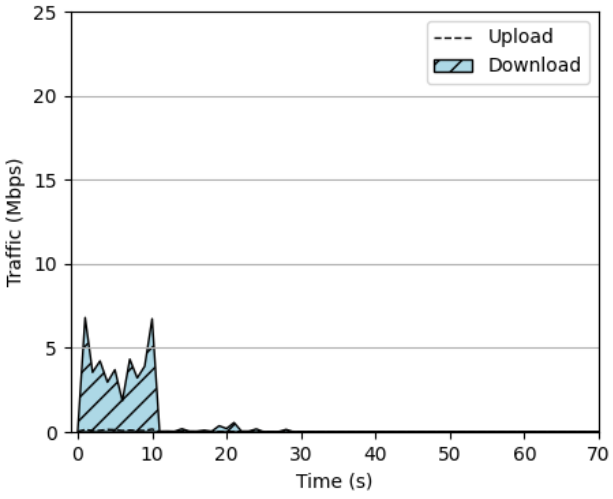



*FTTC*

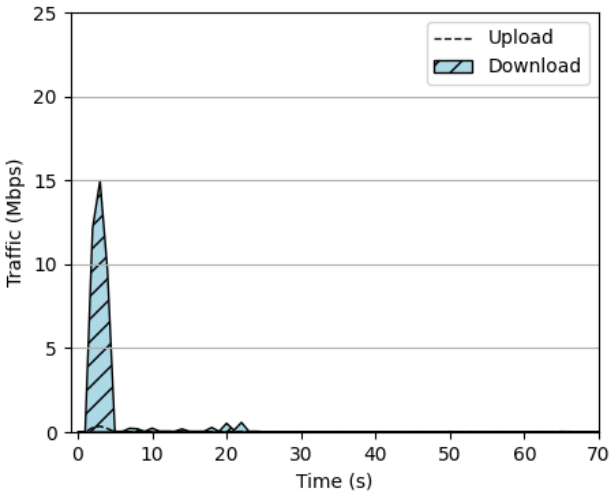



*Unrestricted*

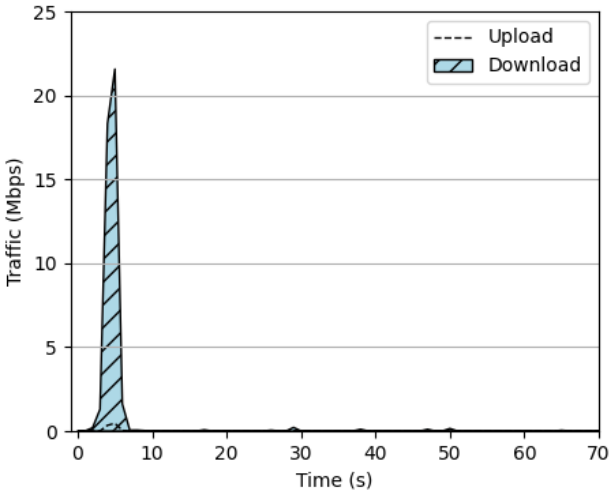
**Figure 8.2:** Network Traffic Measurements, Standard Setup




*DSL*




*FTTC*




*Unrestricted*

**Figure 8.3:** Network Traffic Measurements, Priority Setup

### 8.3.2 Frame Rate

The *Frame Rate*, expressed in *frames per second* (FPS), is the frequency at which consecutive images are displayed by the device, as illustrated in Figure 8.4. The value is measured on the *Meta Quest 2* using the *OVR Metric Tools* [40] for the duration of the experiments. The *Meta Quest 2* has been set to the experimental value of 120 Hz of screen refresh rate, so it can theoretically handle up to 120 FPS. The Figure 8.5 shows the *Frame Rate* of the two displays of the device during the *Standard Setup* experiments, and the Figure 8.6 shows the *Frame Rate* of the displays of the device during the *Priority Setup* experiments.

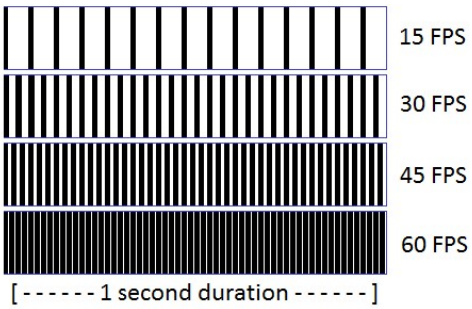



**Figure 8.4:** Frame Rate Description

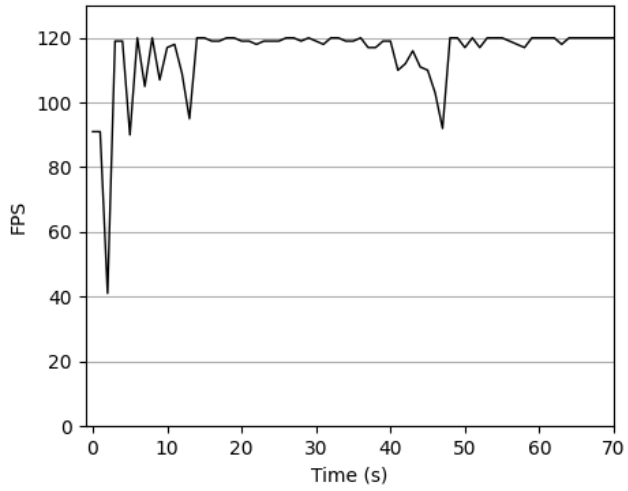
All the results in Figure 8.5 and Figure 8.6 show an initial phase in which the *Frame Rate* drastically decreases, mainly caused by the assets reception and deserialization that the Client must perform, and by the objects rendering pipeline of the local 3D scene. These operations are computationally heavier and produce a slowdown in the refresh rate of the application.

Whilst for the *DSL* and *FTTC* bandwidth limits, the *Frame Rate* value remains around 40 FPS at its lowest peaks, the *Unrestricted* experiment shows an initial drop that reaches nearly 0 FPS. This is caused not only by the higher bandwidth available, but especially by the higher *Objects Send Frequency* (Table 8.1): the high amount of data sent by the Server in a short period of time, when most of the assets of the scene are encountered for the first time, cause the Client device to clog up. It is important, however, to note that in the *DSL* and *FTTC* experiments the scene is loaded more slowly around the player, providing an overall poorer user experience, especially in the *Standard Setup* tests. On the other hand, the *Unrestricted* tests offer a faster download of almost the entire scene, on both the *Standard* and the *Priority setup*, at the cost of an initial freeze of the simulation for a few seconds.

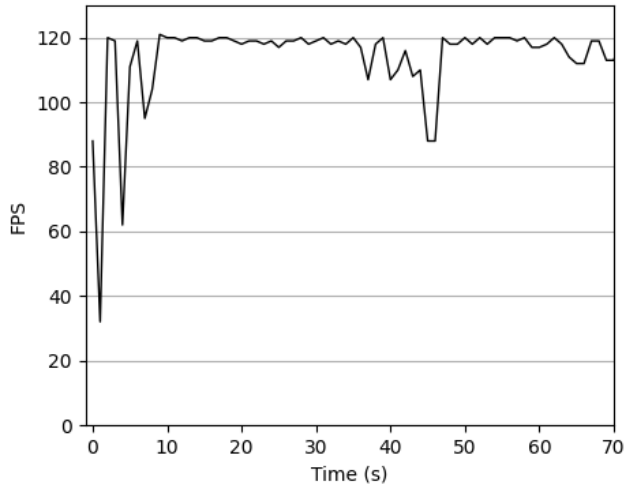
The Table 8.3 shows the average Client's FPS and the FPS Standard Deviation.

| Test | Average (S) | SD (S) | Average (P) | SD (P) |
|---|---|---|---|---|
| **DSL** | 115.15 FPS | 11.39 FPS | 116.29 FPS | 11.67 FPS |
| **FTTC** | 114.03 FPS | 13.45 FPS | 116.20 FPS | 14.16 FPS |
| **Unrestricted** | 111.57 FPS | 24.76 FPS | 116.16 FPS | 18.01 FPS |

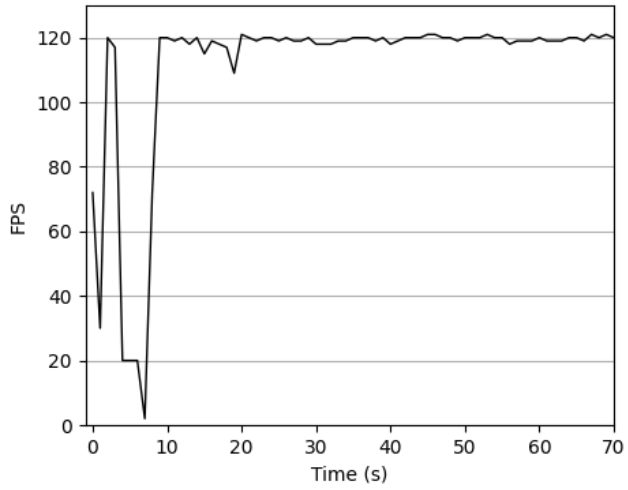
**Table 8.3:** Frame Rate Average and Standard Deviation. (S)=Standard, (P)=Priority




*DSL*

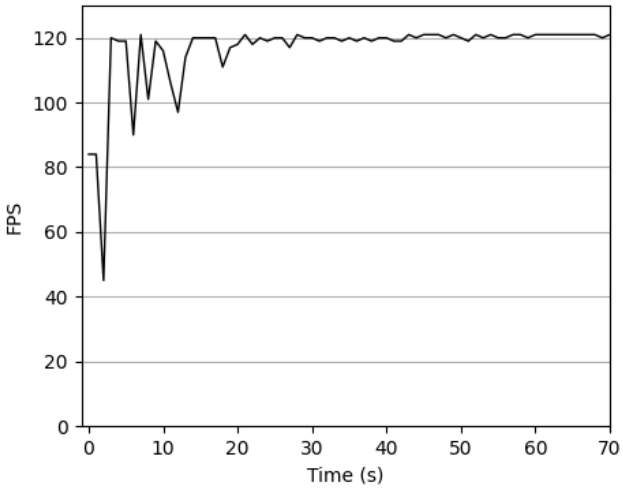



*FTTC*

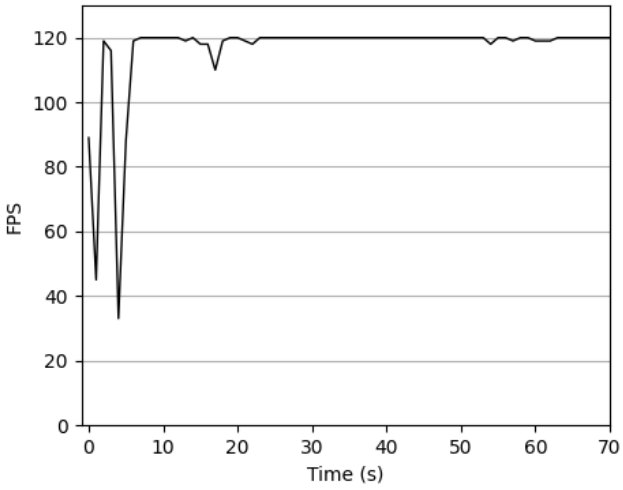



*Unrestricted*

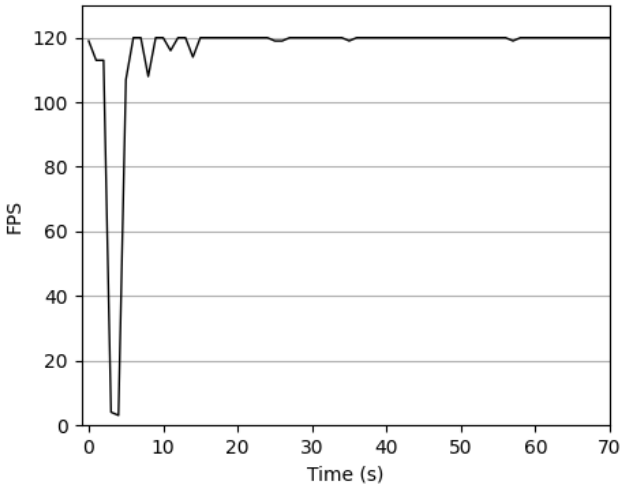
**Figure 8.5:** Frame rate Measurements, Standard Setup




*DSL*




*FTTC*




*Unrestricted*

**Figure 8.6:** Frame rate Measurements, Priority Setup

### 8.3.3 CPU and GPU usage

The *Quest 2* Central Processing Unit (CPU) and Graphic Processing Unit (GPU) usage during the experiments is reported in Figure 8.7 for the *Standard Setup* and in Figure 8.8 for the *Priority Setup*. The device is equipped with the Qualcomm Snapdragon XR2 SoC, a derivative of the Snapdragon 865 designed for VR and augmented reality devices, which makes use of the Adreno 650 graphic chip.

The results reflect the same considerations as those made for the *Frame Rate* measurements. In particular, the CPU usage increases when the framework performs operations of deserialization of the assets and decreases when the assets have been cached and the Client can reutilize those assets to draw the objects received by the Server. On the other hand, the usage of the GPU is low when few objects need to be rendered and displayed in front of the user camera, and increases proportionally with the number of objects loaded and shown in the environment in front of the Player.

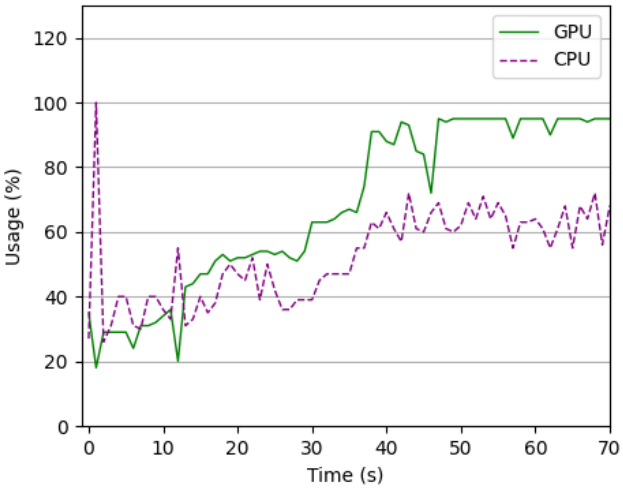
The Table 8.4 shows the average GPU usage and the standard deviation of GPU usage. The Table 8.5 shows the same data about the CPU.

| Test | Average (S) | SD (S) | Average (P) | SD (P) |
|---|---|---|---|---|
| **DSL** | 69.09 % | 25.34 % | 70.03 % | 18.91 % |
| **FTTC** | 70.47 % | 24.65 % | 75.32 % | 14.11 % |
| **Unrestricted** | 87.17 % | 16.02 % | 82.49 % | 11.45 % |

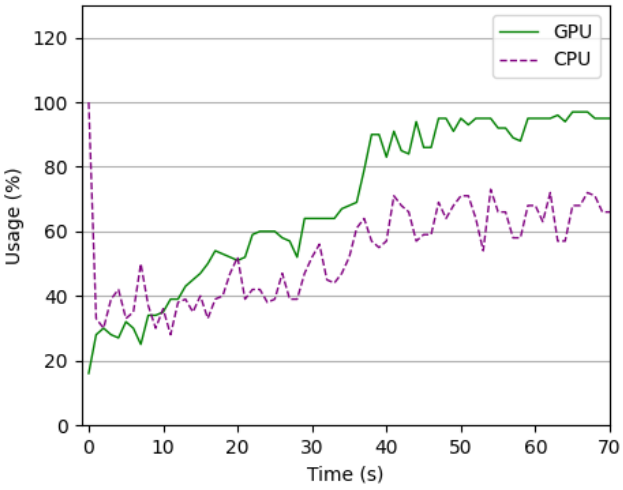
**Table 8.4:** GPU usage Average and Standard Deviation. (S)=Standard, (P)=Priority

| Test | Average (S) | SD (S) | Average (P) | SD (P) |
|---|---|---|---|---|
| **DSL** | 52.53% | 13.91% | 37.24% | 8.19% |
| **FTTC** | 53.51% | 14.33% | 40.14% | 5.51% |
| **Unrestricted** | 53.38% | 8.63% | 53.38% | 8.63% |

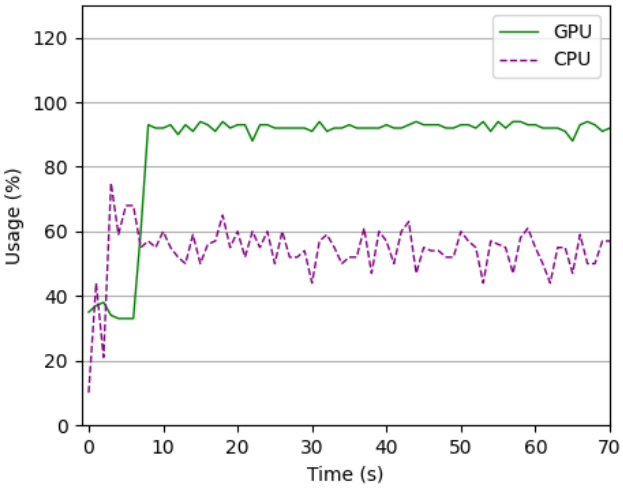
**Table 8.5:** CPU usage Average and Standard Deviation. (S)=Standard, (P)=Priority




*DSL*

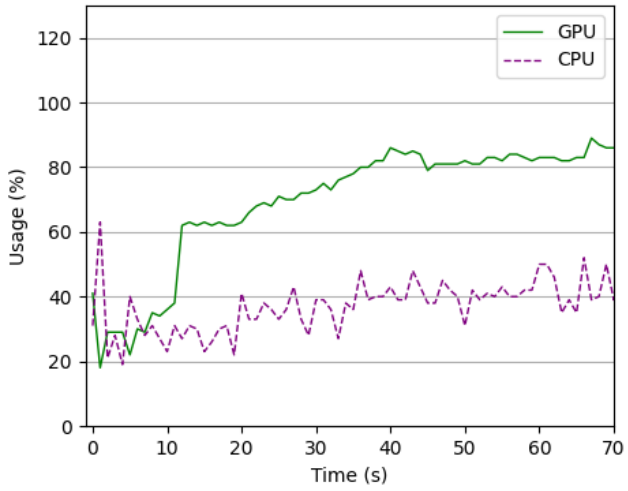



*FTTC*

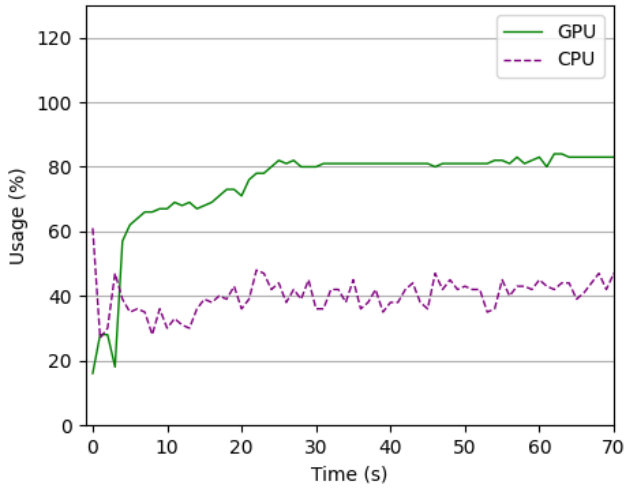



*Unrestricted*

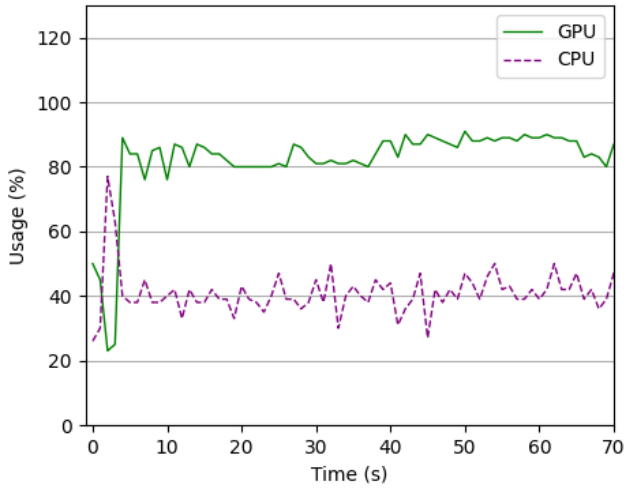
**Figure 8.7:** GPU and CPU usage Measurements, Standard Setup




*DSL*




*FTTC*




*Unrestricted*

**Figure 8.8:** GPU and CPU usage Measurements, Priority Setup

### 8.3.4 Pixel Ratio

The *Pixel Ratio* is a measure that we have introduced to solve in the first instance the problem described in Section 8.2. We define *Pixel Ratio* of two images as the number of pixels that, with the same X and Y coordinates, contain different color values. In particular, we calculated this metric using two pairs of images:

- **Target images**: Screenshots from the perspective of the user camera, taken every half second by the Client;
- **Reference image**: the last screenshot of the *Unrescricted* experiment that, being the fastest test, has all the required elements already loaded into the local scene. This image represents the ideal scenario in which all the objects of the scene are contained in the local memory of the device, and thus there is a negligible delay in loading them.

In Figure 8.9 and Figure 8.10 are shown the view screenshots of the scene at around 0 s, 14 s, 30 s and 47 s after the start, respectively, for the *Standard* and *Priority* setup. In particular, the *Pixel Ratio* value of 0% corresponds to the first scene on the left, without elements on the scene other than the terrain and the skybox, while the value of 100% represents the fully loaded scene, on the bottom-right of the Figure 8.10.

The comparison in Figure 8.11 shows the values of the measure calculated for each screenshot for the duration of the experiments. It can be seen that in each experiment, the *Priority* setup manages to reach a higher similarity to the reference image faster than the *Standard* counterpart. This allows the user to see most of the scene near them loaded in a shorter amount of time, giving the impression of a more immediate immersion into the game world.

The Table 8.6 shows the average Pixel Ratio of each experiment and the standard deviation of the Pixel Ratio.

| Pixel Ratio | Average (S) | SD (S) | Average (P) | SD (P) |
|---|---|---|---|---|
| **DSL** | 51.63 % | 45.14 % | 83.48 % | 33.47 % |
| **FTTC** | 52.13 % | 44.99 % | 92.91 % | 21.54 % |
| **Unrestricted** | 92.61 % | 25.86 % | 94.29 % | 21.93 % |

**Table 8.6:** Pixel Ratio Average and Standard Deviation. (S)=Standard, (P)=Priority

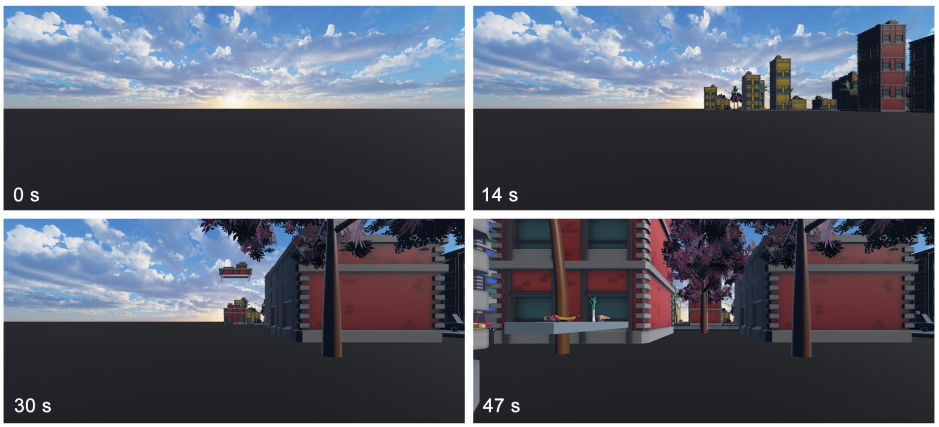



**Figure 8.9:** Scene Screenshots, Standard Setup

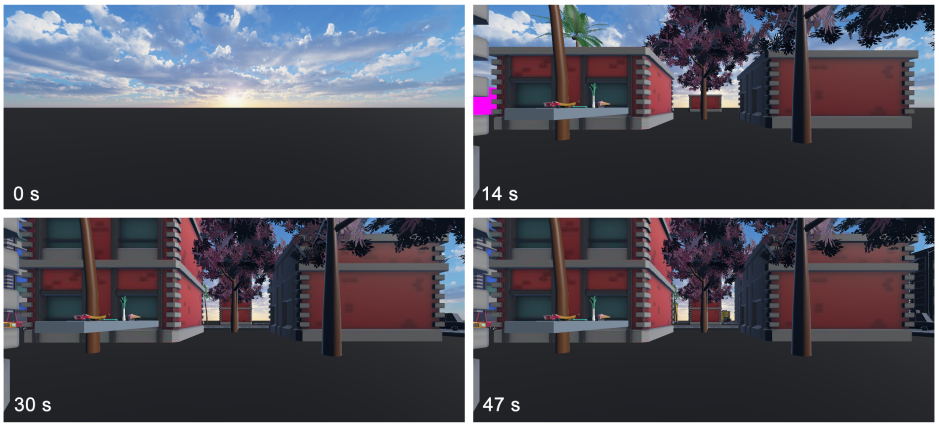
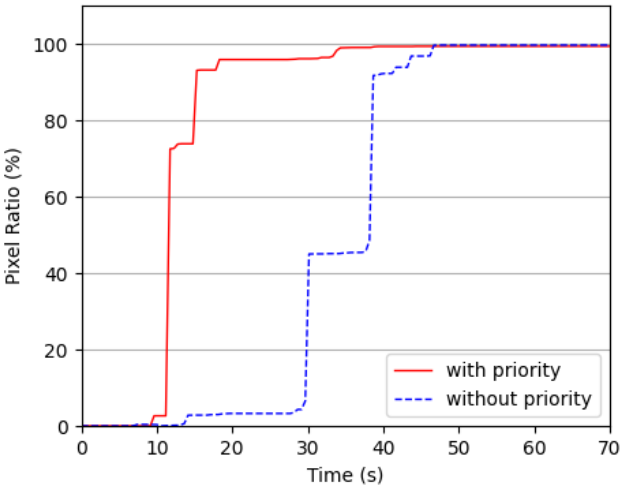
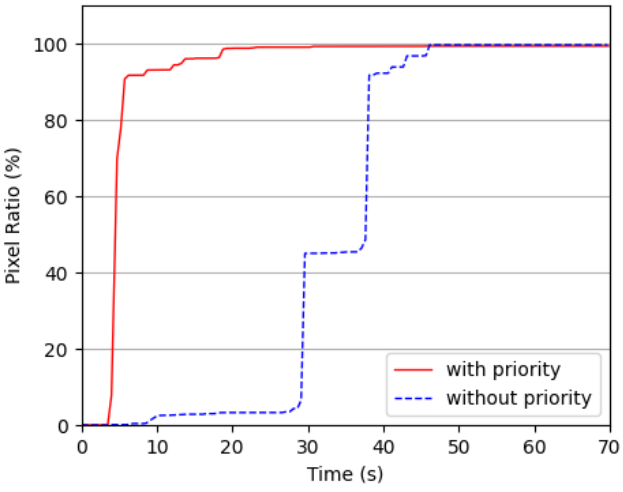


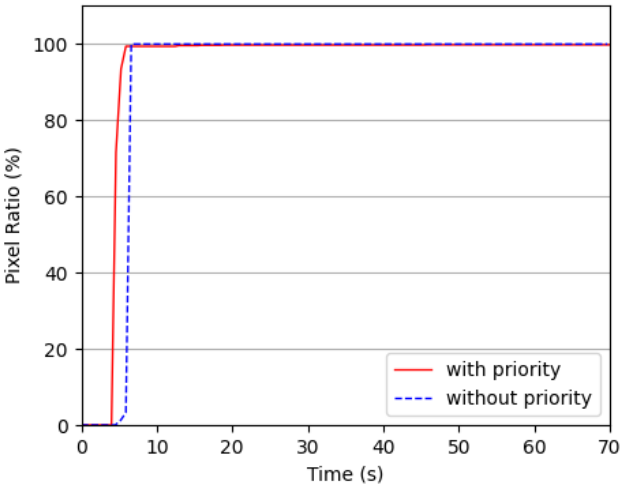

**Figure 8.10:** Scene Screenshots, Priority Setup




*DSL*




*FTTC*




*Unrestricted*

**Figure 8.11:** Pixel Ration Measurements

### 8.3.5 HDR-VDP

As described in Section 2.2, *HDR-VDP* [11] is a metric designed for High Dynamic Range (HDR) multimedia contents, to test the fidelity (e.g. how distracting are image compression distortions) or visibility (is the information sufficiently visible) of HDR images.

In our context, the metric is used to quantify the difference between two images from the perspective of human eyes. Similarly to the *Pixel Ratio* measure (Section 8.3.4), the *HDR-VDP* has been calculated between the screenshots captured during the simulation from the Client's perspective and the reference image represented by the last screenshot of the *Unrestricted* simulation. We utilize the *detection* task parameter, also used by the *HDR-VDP-2* version, and calculate the mean of the *P_map* values, representing the quality probability of each pixel.

In Figure 8.12 and Figure 8.13 are reported the heat maps of a sample of the comparison results between the screenshots (the shown target screenshots are the same of the Section 8.3.4, taken at 0 s, 14 s, 30 s and 47 s) for the *Standard* and the *Priority* setup, representing in white the higher intensity values, so the most different from the reference image, and in black the areas more similar to it.

In Figure 8.14 are reported the simulation plots using the limitations of *DSL*, *FTTC* and *Unrestricted* bandwidth. In general, we can notice, similarly to the *Pixel Ratio*, that the *Priority* solution approaches the final scene screen with a steeper trend, confirming the consideration that the user benefits from the streaming of the objects following the chosen priority algorithm.

The Table 8.7 shows the average HDR-VDP percentage of the tests and the HDR-VDP standard deviation as a percentage.

| HDR-VDP | Average (S) | SD (S) | Average (P) | SD (P) |
|---|---|---|---|---|
| **DSL** | 54 % | 47 % | 19 % | 34 % |
| **FTTC** | 54 % | 47 % | 09 % | 23 % |
| **Unrestricted** | 08 % | 26 % | 06 % | 23 % |

**Table 8.7:** HDR-VDP Average and Standard Deviation. (S)=Standard, (P)=Priority

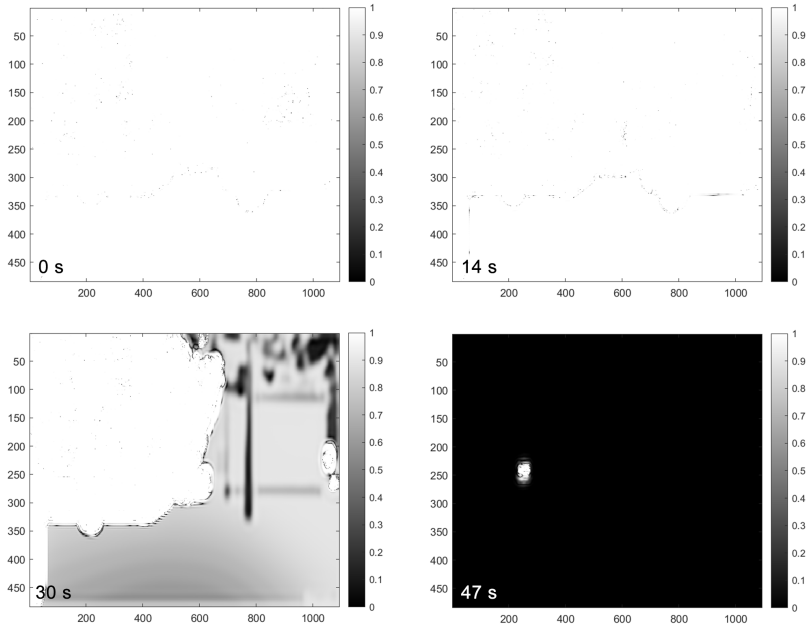



**Figure 8.12:** HDR-VDP Screenshots, Standard Setup (white: different from ref. image, black: equal to ref. image)

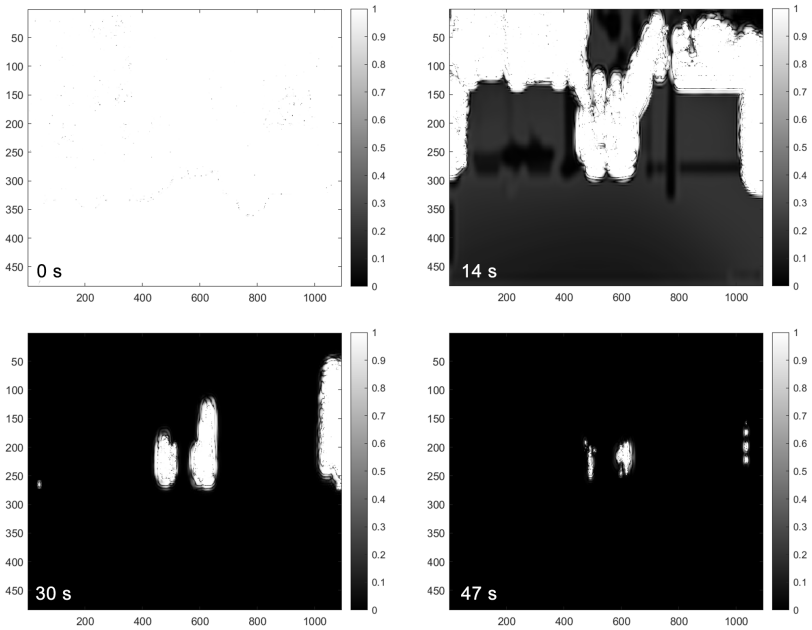
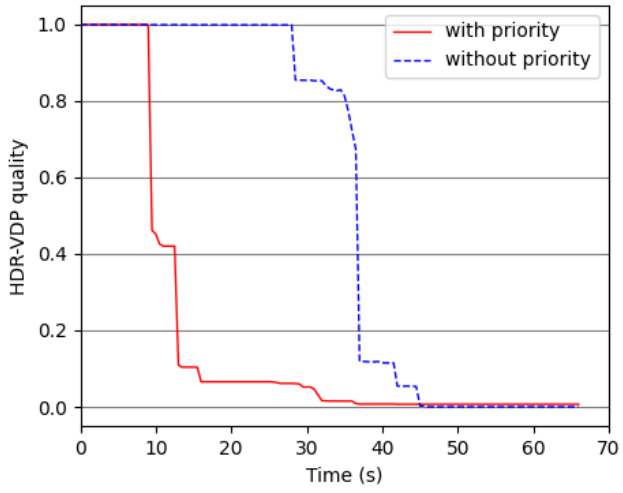
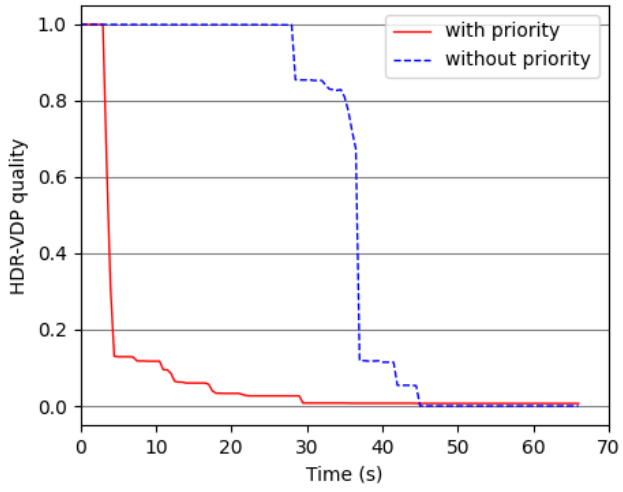


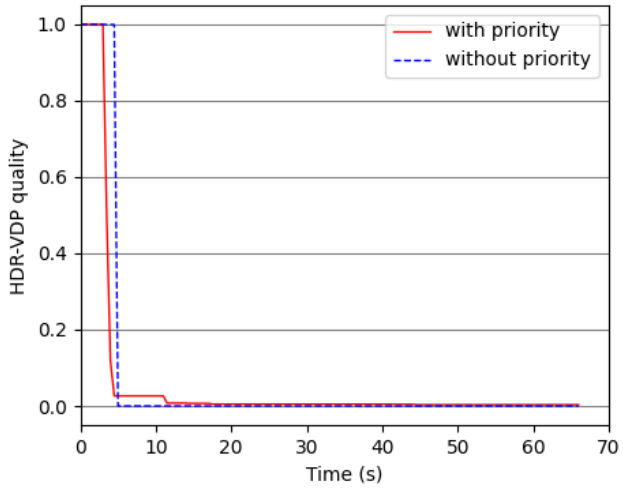

**Figure 8.13:** HDR-VDP Screenshots, Priority Setup (white: different from ref. image, black: equal to ref. image)




*DSL*




*FTTC*




*Unrestricted*

**Figure 8.14:** HDR-VDP Measurements

### 8.3.6 Delta E 2000

The value of $\Delta E$, explained in Section 2.2 is used for the same purpose as *HDR-VDP* (Section 8.3.5) and *Pixel Ratio* (Section 8.3.4).

The heat maps in Figure 8.15 and Figure 8.16 show the pixel-by-pixel color difference, calculated with the most recent $\Delta E$ algorithm, version 2000, which is an approximation of the difference of colors perceived by human eyes. The white color represents the most different pixels between the *target* and the *reference* images, while the black color represents the most similar ones.

The mean of the color differences for the *DSL*, *FTTC* and *Unrestricted* bandwidth, plotted in Figure 8.14, shows the same evolution as the *HDR-VDP* measurement.

An important difference to note is that, being calculated on a pixel basis, the *Delta E* measure does not take into account some portions of the scene that remain unchanged during the experiments. In fact, the ground and some portions of the skybox are visible in both the initial screenshot and the final one, causing the values on the charts to start from an average pixel difference of around 15% rather than 100%.

The Table 8.8 shows the $\Delta E$ metric average values and the $\Delta E$ standard deviation of the results.

| DeltaE | Average (S) | SD (S) | Average (P) | SD (P) |
|---|---|---|---|---|
| **DSL** | 6.74 $\Delta E$ | 6.40 $\Delta E$ | 2.42 $\Delta E$ | 5.00 $\Delta E$ |
| **FTTC** | 6.58 $\Delta E$ | 6.28 $\Delta E$ | 1.01 $\Delta E$ | 3.23 $\Delta E$ |
| **Unrestricted** | 1.07 $\Delta E$ | 3.80 $\Delta E$ | 0.82 $\Delta E$ | 3.28 $\Delta E$ |

**Table 8.8:** Delta E 2000 Average and Standard Deviation. (S)=Standard, (P)=Priority

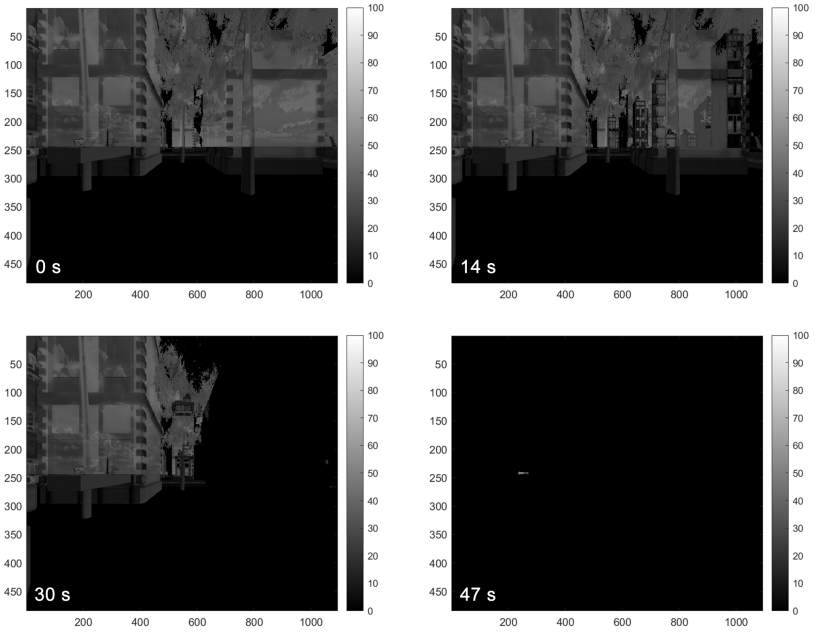



**Figure 8.15:** Delta E 2000 Screenshots, Standard Setup (white: opposite of ref. color, black: equal to ref. color)

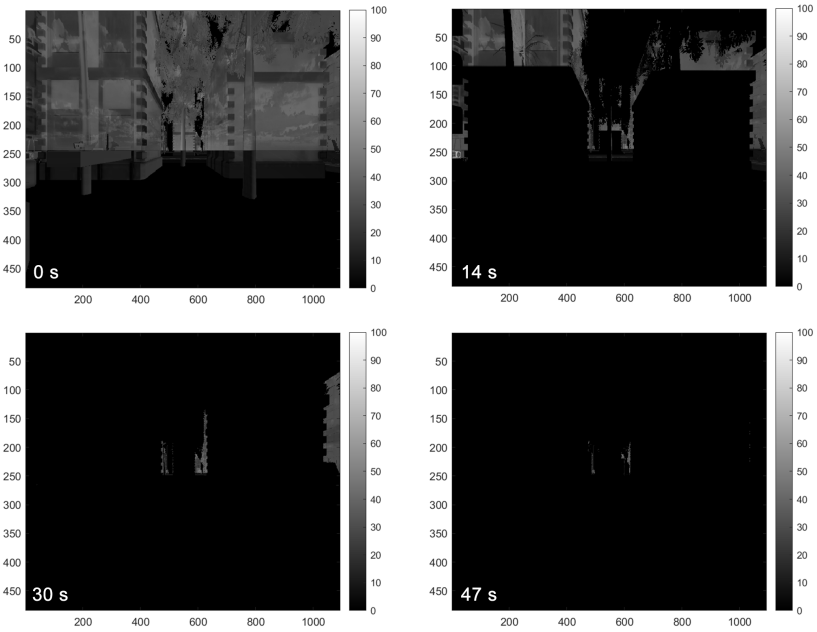
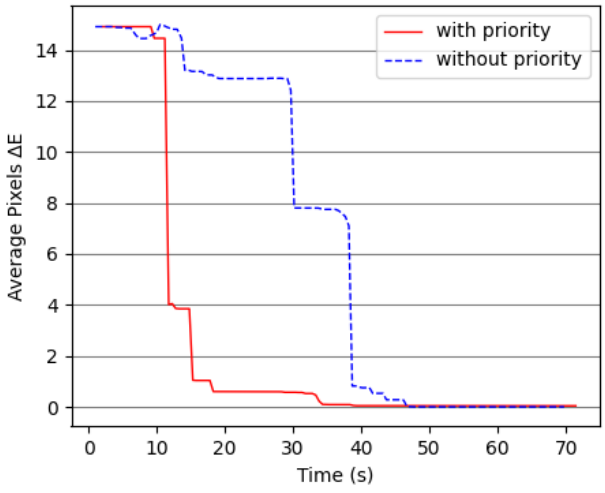
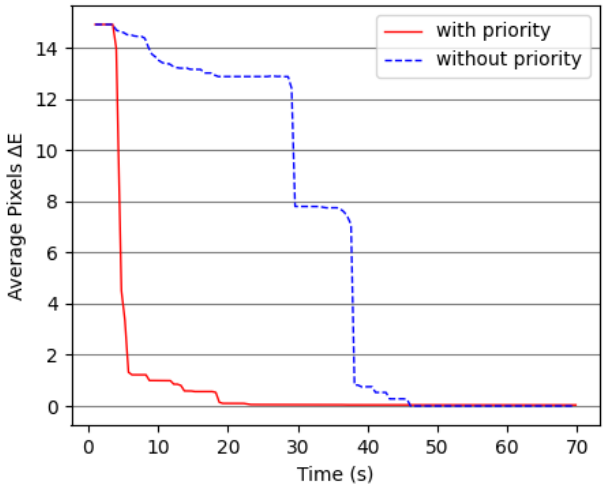


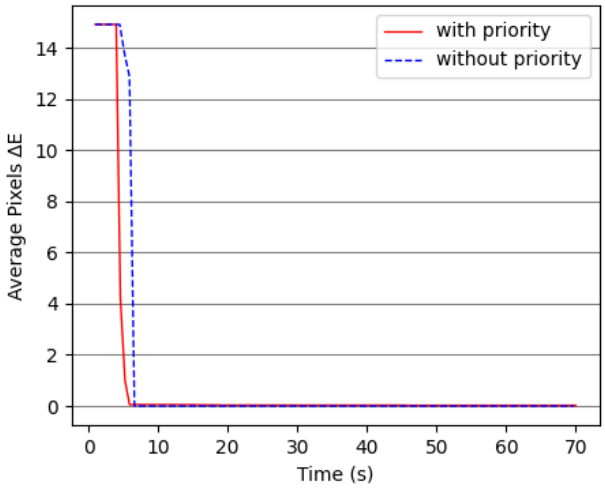

**Figure 8.16:** Delta E 2000 Screenshots, Priority Setup (white: opposite of ref. color, black: equal to ref. color)




*DSL*




*FTTC*




*Unrestricted*

**Figure 8.17:** Delta E 2000 Measurements

# Chapter 9

# Conclusions

In conclusion, throughout this thesis work, we demonstrated that it is possible to create a virtual reality game, experience, or environment arbitrarily extended that the users do not need to completely download before entering and starting to play, but that can be progressively streamed to the local device.

The key idea is that, as happens for other multimedia content, like music or videos, a 3D environment could also be streamed as is through the network. As opposed to *Cloud Gaming*, which receives the input commands by a Client, renders the images of the game on the Servers, compresses the resulting frame's images, and streams the game back to the Client as a video flow, we explored a solution that works by exploiting the characteristics of 3D objects, sending them, and delegating the rendering process directly to the Clients. In this way, it is possible to achieve better visual quality and at the same time better use of the available bandwidth, at the cost of some additional computational overhead for the Client devices.

This work investigates a possible implementation of such technology, which uses the *Unity Engine* and the *Netcode* library to stream the elements of the scene from the Server to the Clients. In particular, after the Client's connection, the Server of our framework creates multiple Areas of Interest around the Players, used as threshold to send objects of different importance; enqueues such objects with a priority that depends on the distance from the Player and from the objects' importance; sends them and exposes the assets that they use, to be downloaded via HTTP by the Clients. We also made the framework usable with VR devices; implemented functionalities of objects' movements and properties update, to make a more complete and interactive experience for the users; and allowed the Clients to activate different behaviors that can affect the state of some objects.

In a standard *metaverse-like* social platform for VR devices, the game world would be completely downloaded before the immersion of the Player into the scene, causing longer waiting times, higher storage memory occupation, and the need to use a smaller number of objects or with fewer graphical complexity. As seen in the results in Section 8.3, using our priority system allows to obtain a faster visualization of the scene employing the same network traffic and computational power, as well as to start interacting while the environment download gets completed around the Player.

A common defining concept of *metaverse* still does not exist, despite the efforts of some of the most important tech companies to develop the new standard for a more

immersive kind of communication between distant people. The software and hardware required to achieve this are used by only a fraction of the Internet users; the remaining majority, on the other hand, still prefer existing and well-established social networks. However, the supportive technology we designed and prototyped may help the building of a more lightweight and dynamic infrastructure on which virtual reality and real-time social applications run. This kind of approach is suitable not only to social platforms, but also to multiplayer or singleplayer video games, and to any 3D virtual reality experience that includes extended explorable areas.

## 9.1 Future Work

Different kinds of test and experiment have been left for the future due to the breadth of the possible variables involved. First of all, a test where the player moves in the scene would better reflect the user experience, but it requires a redesign of the set of metrics to support the dynamic change of view. Secondly, tests involving the synchronization and activation of objects' properties would help to evaluate the performance and responsiveness of the system. Lastly, user tests have also not been included in the thesis work due to the lack of time, equipment, and a large enough pool of people to try out the developed product. This could have offered different points of view on the effective benefits of priority solutions over standard ones, helping to fix issues or enhancing the framework.

Given that most of the effort has been put into developing from scratch a working prototype, including all the required functionality, some of the more specific aspects of the implementation would benefit from deeper optimization. Two main examples are the following:

- The *priority system*, although it obtained better results in our tests than the standard solution, has been kept very simple, calculating the priority of the objects on the basis of only two variables. For example, the system could automatically assign an importance level to the objects based on size, visibility from the user's perspective, or other environmental variables.
- The objects and assets *serialization* methods may be optimized (i.e., making use of compression algorithms) to further reduce the impact on the network and allowing the streaming of more complex objects and textures.

  A different approach that could be explored for the streaming of the objects is to subdivide their mesh into smaller subsets of triangles that could then be sent in smaller packages, in order to dynamically render the objects as soon as the pieces are received.

  There is also the option to create (beforehand or at run-time) different versions of the meshes with different LOD (Level of Detail) and select which mesh to send based on the network conditions or the distance from the Player, as described in Section 4.2.1. The same approach can be applied to the textures, for example, making them appear more detailed as the Player gets closer.

Another interesting enhancement from which the framework would benefit, from a marketable product perspective, is the addition of different synchronization and activation modules, so as to allow the use of more object properties, such as sound emission, shaders, or animations. The process is facilitated by the modularity of such functionalities. Moreover, the support for many different other entity features may

allow the building of better and more authentic worlds, for example, dynamic meshes and textures, wind sources, water and other fluids.

Finally, the ease of use of Unity and its ready-to-use networking and virtual reality toolkits has directed our choice towards this game engine, but the same idea could be redesigned for other engines, with focus on different aspects of the framework. To mention one, the project may be re-implemented using the Unreal Engine, taking advantage of the astonishing graphics that this engine offers. This new version would eventually allow the creation of a more immersive and realistic experience for the final users.

# Bibliography

## References


[1] Sega Corporation. *Sega*. URL: https://www.sega.com/ (cit. on p. 1).

[2] Meta. *Ray-Ban Stories*. 2021. URL: https://about.fb.com/news/2021/09/introducing-ray-ban-stories-smart-glasses/ (cit. on p. 1).

[3] Mojo Vision Inc. *Mojo Lens*. 2022. URL: https://www.mojo.vision/mojo-lens (cit. on p. 1).

[4] *Virtuix Inc.* URL: https://omni.virtuix.com/ (cit. on p. 2).

[5] *Cybershoes Inc.* URL: https://www.cybershoes.com/ (cit. on p. 2).

[6] Stylianos Mystakidis. *Metaverse*. Vol. 2. 1. 2022, pp. 486–497. DOI: 10.3390/encyclopedia2010031. URL: https://www.mdpi.com/2673-8392/2/1/31 (cit. on p. 2).

[7] VRChat Inc. *VR Chat*. URL: https://vrchat.com (cit. on p. 2).

[8] Unity Software Inc. *Unity Game Engine*. URL: https://unity.com/ (cit. on pp. 2, 11, 53).

[9] Ryan Shea et al. *Cloud gaming: architecture and performance*. Vol. 27. 4. 2013, pp. 16–21. DOI: 10.1109/MNET.2013.6574660 (cit. on p. 2).

[10] Gaurav Sharma, Wencheng Wu, and Edul N. Dalal. *The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations*. Vol. 30. 2005, pp. 21–30 (cit. on pp. 3, 10, 61).

[11] Rafał K. Mantiuk, Karol Myszkowski, and Hans-Peter Seidel. *High Dynamic Range Imaging*. 2015 (cit. on pp. 3, 61, 75).

[12] S.-Y. Hu et al. *FLoD: A Framework for Peer-to-Peer 3D Streaming*. 2008, pp. 1373–1381. DOI: 10.1109/INFOCOM.2008.195 (cit. on p. 5).

[13] Thomas A. Funkhouser. *RING: A Client-Server System for Multi-User Virtual Environments*. I3D '95. Monterey, California, USA: Association for Computing Machinery, 1995, 85–ff. ISBN: 0897917367. DOI: 10.1145/199404.199418. URL: https://doi.org/10.1145/199404.199418 (cit. on p. 6).

[14] Alexandros Doumanoglou et al. *Benchmarking open-source static 3D mesh codecs for immersive media interactive live streaming*. Vol. 9. 1. IEEE, 2019, pp. 190–203 (cit. on p. 7).

[15] Giacomo Parolini. *Distributed Rendering in Vulkan*. 2018. URL: https://silverweed.github.io/assets/docs/distributed_rendering_in_vulkan.pdf (cit. on p. 8).

[16] The Khronos® Group Inc. 2022. *Vulkan API*. URL: https://www.vulkan.org/ (cit. on p. 9).

[17] Epic Games Inc. *Unreal Engine*. URL: https://www.unrealengine.com/en-US (cit. on p. 11).

[18] Ariel Manzur Juan Linietsky. *Godot*. URL: https://godotengine.org/ (cit. on p. 11).
[19] Crytek GmbH. *CryEngine*. URL: https://www.cryengine.com/ (cit. on p. 11).
[20] Amazon Games. *Amazon Lumberyard*. URL: https://aws.amazon.com/lumberyard/ (cit. on p. 11).
[21] Unity Technologies. *XR Interaction Toolkit*. URL: https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@0.9/manual/index.html (cit. on pp. 12, 23, 53).
[22] vis2k. *Mirror*. URL: https://mirror-networking.com/ (cit. on p. 12).
[23] Exit Games Inc. *Photon Pun*. URL: https://www.photonengine.com/pun (cit. on p. 12).
[24] Unity Software Inc. *Unity Netcode for GameObjects*. URL: https://mirror-networking.com/ (cit. on p. 12).
[25] Unity Technologies Christopher Pope. *Netcode RPC*. URL: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/messaging-system/index.html (cit. on pp. 13, 26).
[26] Unity Technologies Jaedyn Draper. *INetworkSerializable*. URL: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/serialization/inetworkserializable/index.html (cit. on p. 13).
[27] MDN contributors Mozzilla. *An overview of HTTP*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (cit. on p. 14).
[28] Unity Technologies. *Json Utility*. URL: https://docs.unity3d.com/ScriptReference/JsonUtility.html (cit. on pp. 25, 26, 30).
[29] Microsoft Corporation. *Binary Formatter*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=net-6.0 (cit. on p. 25).
[30] J. Rossignac. *Edgebreaker: connectivity compression for triangle meshes*. Vol. 5. 1. 1999, pp. 47–61. DOI: 10.1109/2945.764870 (cit. on p. 27).
[31] Hugues Hoppe. *Progressive meshes*. 1996, pp. 99–108 (cit. on p. 28).
[32] P. Cignoni, C. Montani, and R. Scopigno. *A comparison of mesh simplification algorithms*. Vol. 22. 1. 1998, pp. 37–54. DOI: https://doi.org/10.1016/S0097-8493(97)00082-4 (cit. on p. 28).
[33] Michael Garland and Paul S Heckbert. *Surface simplification using quadric error metrics*. 1997, pp. 209–216 (cit. on p. 28).
[34] *Unity Serializable A.P.I.* URL: https://docs.unity3d.com/2022.2/Documentation/ScriptReference/Serializable.html (cit. on p. 30).
[35] Unity Technologies. *Unity Coroutines*. 2022. URL: https://docs.unity3d.com/Manual/Coroutines.html (cit. on p. 33).
[36] Microsoft Corporation. *DOT-NET HTTP Listener*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.net.httplistener (cit. on p. 35).
[37] Meta. *Oculus Integration SDK*. URL: https://developer.oculus.com/downloads/package/unity-integration/ (cit. on p. 53).
[38] Locktime Software s.r.o. *NetLimiter*. URL: https://www.netlimiter.com (cit. on p. 59).
[39] SeriousBit Srl. *NetBalancer*. URL: https://netbalancer.com/ (cit. on p. 59).
[40] Meta. *OVR Metrics Tool*. URL: https://developer.oculus.com/documentation/unity/ts-ovrmetricstool/ (cit. on pp. 60, 66).