Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Department of Information Engineering
Master Thesis in Computer Engineering

Development of a virtual reality application for the training of cardiac surgeons

Master Candidate

Boscolo Cegion Nicola

Student ID 2074285

Supervisor

Prof. Savino Sandro

University of Padua

Academic Year
2024/2025
Graduation Date
12/12/2024

To my family and friends,
for their unwavering support,
and encouragement throughout this journey.

Abstract

This thesis describes the design and development of a virtual reality application for the Meta Quest 2 head mounted display for the training of doctors and nurses in the field of pediatric cardiac surgery. The purpose of the application is the presentation of case studies through the immersive visualization of 3D models derived from imaging techniques. The application allows the creation of virtual classrooms where students can interact with 3D models of physiological or pathological hearts under the supervision of the teacher or independently.

Sommario

Questa tesi descrive la progettazione e lo sviluppo di un'applicazione di realtà virtuale per il visore Meta Quest 2 per la formazione di medici ed infermieri nell'ambito della cardiochirurgia pediatrica. Lo scopo della applicazione è la presentazione di casi studio tramite la visualizzazione immersiva di modelli 3D derivati da tecniche di imaging. L'applicazione permette di creare classi virtuali dove gli studenti possono interagire con modelli 3D di cuori fisiologici o patologici con la supervisione del docente o in autonomia.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Snippets

List of Acronyms

AOUPD Azienda Ospedale Università Padova

API Application Programming Interface

APK Android Package

CT Computed Tomography

FFR Fixed Foveated Rendering

FPS Frame Per Second

HMD Head Mounted Display

HTTP HyperText Transfer Protocol

IMU Inertial Measurement Unit

IP Internet Protocol

JSON JavaScript Object Notation

LOD Level Of Detail

MRI Magnetic Resonance Imaging

NDK Native Development Kit

OS Operative system

PC Personal Computer

REST Representational state transfer

RPCs Remote Procedure Calls

SDK Software Development Kit

UE Unreal Engine

UI User Interface

URL Uniform Resource Locator

VR Virtual Reality

VRS Variable Rate Shading

XR Extended Reality

1 Introduction

The pediatric cardiac surgery unit of the University hospital of Padova is an important center, handling more than 300[1] cases per year. The center specializes in surgical treatment of simple and complex congenital heart disease, at any age. The need to teach and visualize the case of operation is very important.
Thanks to Magnetic Resonance Imaging (MRI) and Computed Tomography (CT) scans, surgeons can not only view detailed images of the heart, but also create 3D models Fig.[1.1], as explained in this thesis Bib.[3]. These images are essential for pre-operative planning and understanding the patient's condition. In special cases, 3D models can further enhance understanding of patients with particularly complex heart problems. 3D models can also be used to create molds for producing silicone replicas, allowing surgeons to practice the procedure before performing it on the patient, as explained in Bib.[2]. This approach is typically reserved for special cases, but it can be extremely valuable for future procedures and educational purposes.
In recent years, the pediatric cardiac surgery unit has set up a complete pipeline for the production of 3D printed models. The pipeline covers all the steps from the imaging to the segmentation and preparation of the models to the 3D printing. The models are currently used as a tactile medium to better visualize the morphology of the heart as a support for study, pre-operatory preparation but also to explain to the parents of the patients the pathology of the child.
A collection of the 3D models have been published in a book Bib.[8], with a com-

[1] `https://www.aopd.veneto.it/Cardiochirurgia-Pediatrica`

panion app that allows the readers to visualize the 3D model on their mobile[2]. A stand alone app has also been developed: Cardiology AR is an augmented reality app that allows the user to visualize in the space a selection of 3D models of hearts with congenital diseases.
Both the text book and the AR app however are meant to be used by a single person and, due to the limited selection of models and the inherent technology limitations, lack the functionalities needed to be used successfully as a learning tool during a lesson.
This thesis will discuss the design, development and testing of a software solution that overcomes these limitations and allows the visualization of these 3D models in a classroom. The software exploits Virtual Reality to offer an immersive visualization, and implements functions specifically designed for the use in a teacher to students learning environment.

1.1 Case introduction

As previously mentioned, the Azienda Ospedale Università Padova (AOUPD) pediatric cardiac surgery unit is equipped with a facility for printing cardiac models, which serve as a valuable teaching tool for studying cardiac anomalies. However, printing models is a time-consuming process and, while suitable for examining individual cases, it is impractical for studying multiple cases, as this would require printing numerous models. Additionally, although physical models offer the advantage of immediate usability, they have significant limitations. These limitations include the inability to support remote study, as students must be physically present with the model, and their inherent static nature, which makes them impractical for certain use cases (e.g., resizing the model requires reprinting it).
A solution to these limitations can be the use of virtual reality to view the 3D model. This idea was tested using existing technology and existing software. The experimental setting and the outcomes are described in the following section. The lessons learnt from this experiment made the case for the development of a bespoke app.

[2] https://www.uqido.com/progetti/cardiology-ar/

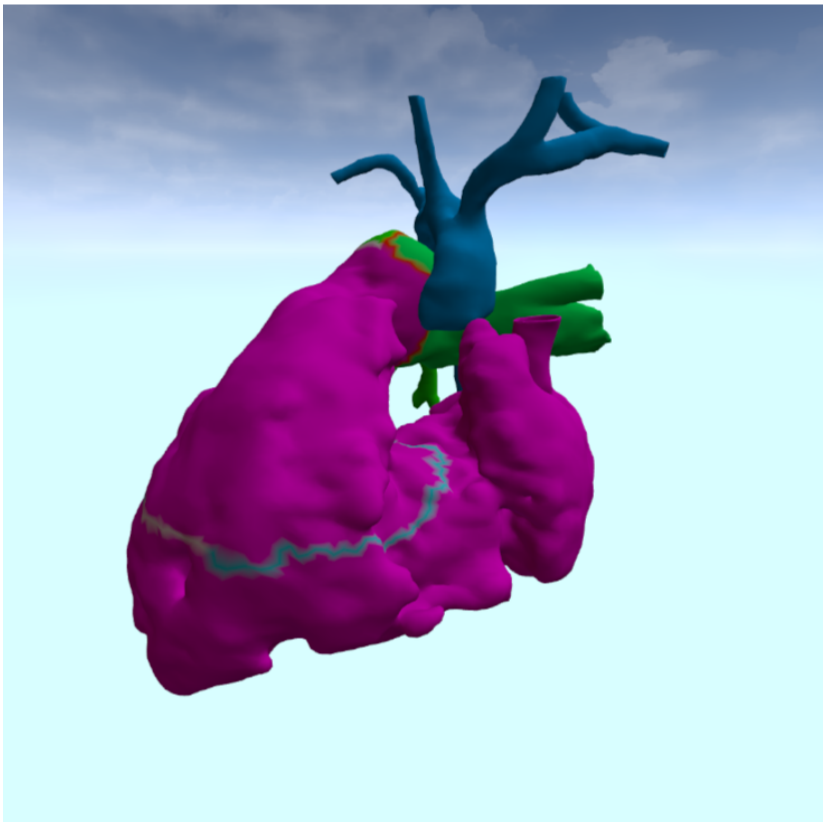


Figure 1.1: 3D model of an heart

1.2 Current technologies tested

As part of this study, a group of surgeons tested an application called ShapeXR using a Meta Quest 2 headset to conduct a virtual reality-based lesson. The surgeons engaged with the application to evaluate its functionality, usability, and effectiveness in a realistic educational context. The outcomes of this experience, and the hardware and software used will be explained in the following sections.

1.2.1 VR hardware

The University of Padua has some Meta quest 2 Fig.[1.2] a Head Mounted Display (HMD) for Virtual Reality (VR).
The Meta Quest 2 is a standalone HMD, which means that it doesn't need other peripherals like an external console or Personal Computer (PC) for working.
The Meta Quest 2 utilizes an Inertial Measurement Unit (IMU) to track the user's head movements and a camera array to determine their position in space, eliminat-

ing the need for external beacons. The cameras also support a pass-through feature; however, this is limited by low resolution and a black-and-white display. For user interaction, the Meta Quest 2 uses two wireless controllers, but it also has the possibility to use hand-tracking, to let the user navigate the interfaces by using his/her own hands. The tracking of the user in the real-world environment is achieved using 4 infrared cameras and other sensors like multi axes gyroscopes and accelerometers. They use a custom version of the Android Operative system (OS), this can give a certain degree of liberty in creating APPs for the device.
The Meta Quest 2, successor to the Oculus Quest, has established itself as one of the best-selling devices on the market due to its extensive feature set and competitive price point, which positions it among the most affordable options available. Recently, it has been succeeded by two newer models, the Meta Quest 3 and Meta Quest 3s. Both models maintain compatibility with their predecessor while introducing new features, such as color pass-through, which effectively classifies them as Extended Reality (XR) devices. Their competitive pricing is likely to make them the new standard in the industry.

Figure 1.2: Meta quest 2

1.2.2 VR software

After a search on the market for readily available free VR applications to use, the staff of the cardiac unit selected as the app to use for the test an app called ShapesXR.

Originally created as a collaboration platform for the design and creation of 3D models, this app allows users to upload models on a website and then to visualize them on the HMD and interact with them in a multi user environment. The app has a multiplayer functionality so that multiple people can look at the 3D models in the environment, even if the developers recommend at max 8 people, initial tests with 14 users connected did not any problem.
ShapesXR lets you create rooms, accessible via a code, where multiple people can create 3D models with basic tools like 3D brushes and standard shapes like cubes, pyramids and so on. It also lets you upload a 3D model file on their own website, so that in the home you can download it and start to work on it. It lets you also create your own avatar. This is the main feature that the surgeons are using for show the 3D Models

1.2.3 Experiment results

Experimenting with the hardware and the software highlighted some problems with this technology, mostly linked to the usability aspects of the app. Testing the solution during a lesson allowed us to collect feedback from the users, both from the teacher and the students point of view.

Software problems: ShapesXR is principally used for 3D modeling, so the app has a lot of features like changing the scale of the world or brushes for modeling the objects that aren't useful for the surgeons, and quite distracting especially when for the user it is the first time using a HMD.
The user experience is extremely important in VR because it is difficult to tutoring the user while using the HMD. Then ShapesXR positions the user in an empty 3D plane with little to none point of reference so If a user accidentally uses the teleport function, they may find themselves somewhere far away from the scene they are supposed to be watching.

1.2.4 Feedbacks from surgeons and nurses:

There was a lesson on 12/12/2023 with the integration of the Quest 2 and ShapesXR. First it was pretty chaotic, a lot of people did not even know how to use the controllers, and they had problems even putting in the code for entering the session. Unfortunately we did not have time to make a nice lesson for teaching how to use the HMD, the tutorial made by Meta approximates takes 15 min to complete, even more if the user wants to try the mini-games, so we did not have the time to show it. The main critical points were:

- Inadequate tutoring for teaching how to use the HMD
- Difficulty for accessing the multiplayer room
- Difficulty at moving in the room
- Some people avatar were blocking the visual of some people

1.2.5 Conclusions

The tests conducted using the Oculus Quest 2 and the Shapes XR app confirmed that visualizing 3D models in virtual reality provides a highly valuable educational experience. However, they also revealed critical issues within the app that render it ineffective as a teaching tool.
The experience gained has resulted in identifying a series of essential features that a VR app must possess to be effectively used in this educational context. These features will be discussed in Chp.[2]. The challenge of finding an existing app with these capabilities highlighted the need to develop a custom software, which will be described in Chp.[3] and its development in Chp.[4].

2

Requirements

This chapter outlines the software features and constraints, specifying what the system must accomplish and the characteristics it must meet to fulfill users needs.

2.1 Functional Requirements

The main features of the software must be:

- **Show custom 3D model at runtime:** The software must be able to download 3D models in OBJ format and render them at runtime, which will also show coloring made by a surgeon. The 3D model should also be movable and have the possibility to change its size.

- **Multiplayer functionality:** The software must recreate a 3D virtual classroom where a professor can show the 3D model to the students, all students and professor must be synchronized.

- **Upload of 3D models:** There will be a wen app where a surgeon can upload and preview 3D models.

- **Tutoring functionality:** The software will have some tools for learning such as laser pointers, and the possibility to change the position and dimension of the 3D model.

- **Compatibility with Meta quest 2:** The system will need to run on Meta quest 2,as these are the HMD available at present in AOUPD.

2.2 Data requirements

The system necessitates the following data (with the following characteristics) to fulfill its objectives:

- **Required Data:**
  - 3D models in OBJ format.
  - Local Internet Protocol (IP) address for multiplayer.
  - Generating and managing codes for multiplayer sessions.

- **Data Format:** All data must respect its standard, other communication between clients and server must use JavaScript Object Notation (JSON) format file. These formats facilitate data exchange with external systems.

2.3 Non-functional requirements

To ensure the development of an effective VR software, the following non-functional requirements must be considered:

- **Use of Game engine:** A game engine is a software made by another company that is capable of creating a video game, by giving the developers some tools for making the development experience quicker and easier.
- **Simple Back end:** The backend must do the least things effectively because the IT department does not need to update the server that will host the services.
- **Compliance with Internet standards:** The system will be compliant with HyperText Transfer Protocol (HTTP) for all communication between server and client, the HMD will use the game engine multiplayer system.
- **Maintaining performance:** VR applications need to perform extremely well for having a good VR experience, Meta allows a minimum of 72 Frame Per Second (FPS), but our target for a better experience will be 90 FPS.

3 Design

The software is made by three main components: VR APP, backend, web app. This chapter will talk about the design and implementation choices driven by the requirements described in the previous chapter.

3.1 The user experience

VR app The user experience in the VR app will be designed to be intuitive, there will be two types of users: professors and students. Professors will have the ability to create a virtual classroom. Students can join these rooms by entering a unique code provided to them. Once inside, the professor will guide the lecture by showcasing 3D models of different hearts. They can manipulate these models by moving, resizing, and using a laser pointer to highlight specific areas of interest.

Web app The web app will let the professor upload any OBJ models needed for the lecture by a simple form, and the web app will allow users to view 3D models.

3.2 The VR app

This section will explain the main choices behind the VR app

The development environment: There are three main way to develop on the Meta Quest 2:

- **Native:** By using native Application Programming Interface (API) that Meta provides for the HMD.
- **Unity:** Is a famous game engine principally used for lightweight video games, It is pretty functional and easy to use, its programming language is `C#`.
- **Unreal Engine:** Is a famous game engine used for big games, its performance is the best in the market, it uses `C++` and a graphical programming language called Blueprint.

As we said in the non-functional requirements we will use a game engine, I have opted for Unreal Engine (UE) because of its performance, the 3D models that will use are complex (~1-10MBytes in size), so it will be used for its performance, also it has a lot of tools for multiplayer and a good documentation other than a big community of developers.

3.3 Unreal Engine

Made by Epic Games, the software was born for the video game Unreal, now it is one of the best engines that power a lot of important video games and 3D animation. For this software it will use the 5.2.1 version because at the time of writing this thesis for the best compatibility with the HMD. The main reason for this upgrade was for faster building time and more advanced API of Meta.

The fundamentals: UE has a 3D preview mode that lets the user set the various 3D objects in the scene. In unreal a scene is called "Level", each element in a level is called "Actor", Actors are made by multiple components.
Each element of unreal can be built with a proprietary system called Blueprint, a visual programming interface, or via `C++`, or by combining them. Principally Blueprint can make the development of the software faster and easier at the cost of performance, `C++` performs better, and it has a bigger range of tools. So It is important to combine both languages for having the best performance and flexibility in the software.
Like a lot of game engines, UE tries to render as many frames as possible, each cycle of rendering is called tick.
It's important that long actions must be asynchronous with respect to the game ticks, and if something must be a runner in the so-called "game thread", it must be as fast as possible so that the frame rate does not drop to a certain level.

Another important component is the player controller, this is the instance of a player inside a level, a player controller can possess a pawn actor or a character actor.

External library: One challenge for this application is having a correctly multiplayer session between HMD, unfortunately UE does not provide a `VR Character` that has multiplayer functionality. For having a faster developing experience, a library called VRexpansion[1] will help the development, the library is open source under MIT licensing. The main useful components are:

- **VR Character:** a character for VR games, all its components can be replicated in a multiplayer session.
- **Grippable Interface:** interface that can enable actors or static meshes to be gripped by the HMD components. It also can be replicated in a multiplayer session.

Events: Events are what start the execution of code inside a Blueprint. Each blueprint have its own standard event, they depend on the object, the basic ones are:

- **Begin Play:** runs one time when the actor is spawned (this is not a constructor).
- **On tick:** runs on every game tick.

We can also create custom events with Blueprint or `C++`, they can be triggered whenever we want, they can also be replicated in clients in multiplayer sessions. Like functions, events can have inputs but not outputs.

Multiplayer: UE just supports a client-server configuration for managing multiplayer sessions, it has two main implementations: Stand-Alone server and listen server.
A Stand-Alone server consists in having a server that emulates the gaming session, it requires a server powerful enough to run the basic function of the game event if it does not need to render it.
A listen Server does not need a Stand-Alone server, simply one device acts not only as a client but also as a server. Normally client-server is the best in performance and

[1] `https://vreue4.com`

latency, but because this software is simple, the listen server is the right implementation.
For making a good multiplayer we have some blueprints that can help with the synchronization of the data:

- **Game Mode:** runned only inside the server, as it is called it provides the rules how the game should work, it is important for the login and log out of the players.
- **Game state:** it represents the state of the game, all HMD has one, but it is always replicated by the server.
- **Player controller:** runned in every HMD, it represents the player using the HMD.
- **Player state:** it has all the data of a player like the username, it's replicated.

An important feature for synchronize events between client and server are Remote Procedure Calls (RPCs), RPCs are events of actors that can per reproduce in multiple device at once by following one of these rules:

- **Client:** The RPCs are executed on the owning client connection for this actor.
- **Server:** The RPCs are executed on the server, it must be called from the client that owns this actor.
- **Multicast:** The RPCs are executed on the server and all currently connected clients the actor is relevant for, Multicast RPCs are designed to be called from the server, but can be called from clients. A Multicast RPCs called from a client only executes locally.

Multiplayer Since multiplayer is one of the fundamentals of modern gaming, UE provides a suite of tools to help achieve it. Unfortunately, some of these tools will be unusable because, at this time, we cannot publish the app to the Meta Quest store. This limitation will restrict the multiplayer capabilities, allowing us to create only a LAN-based multiplayer mode, meaning that the HMDs must be connected to the same local network.

3.4 Network Infrastructure

The Network Infrastructure Fig.[3.1] it is simple, a server will be hosted in the IT department of the hospital that will host the Representational state transfer (REST) server for managing 3D Models and multiplayer sessions, and in the same server will host via NodeJS the wen app for managing the 3DModels. The multiplayer itself will be manage by the host HMD

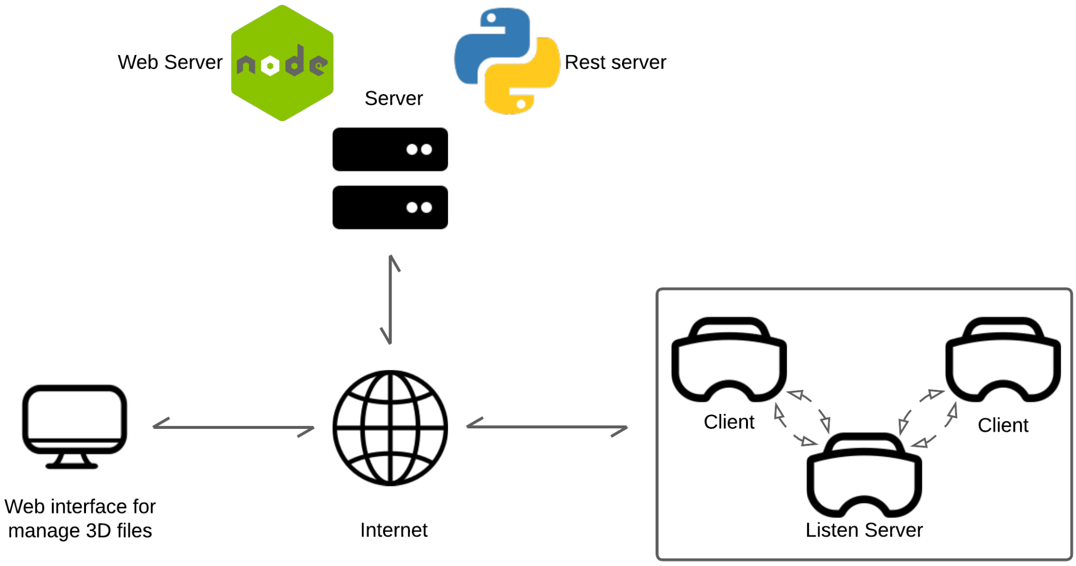


Figure 3.1: Network Schema

The backend: The backend is a simple Python program that functions as a REST server, where each 3D model is a resource. It also manages the multiplayer session by creating the session code, and saves the IP address of the listen server. The library used is called Flask[2], it simply lets you run a function with respect to a HTTP response received.

The web app: The web app is made with ReactJS, a popular framework for website, the main reason of this choice is for faster development because its community is massive, in fact will be using a User Interface (UI) library called MUI[3] and a 3D library for rendering 3D models called React Three Fiber[4].
The main functionality of the web app will be:

- previewing 3D models with the right colors

[2] https://flask.palletsprojects.com/en/3.0.x
[3] https://mui.com
[4] https://r3f.docs.pmnd.rs/getting-started/introduction

- upload 3D models
- delete 3D models

3.5 Development environment

It is important to set the right development environment for this software, even if for the web app and backend it is a standard NodeJS and Python environment, it is not that simple for UE
First the choosing the right version is very important because Meta does not always test all UE version, at the start of the developing of the software, the last one was 5.2.1, we also need Visual Studio with the right components for compile in C++, we could use other text editor, but Visual Studio is the most completed one for unreal. For other information look at: Bib.[5] After we set up the basic UE environment, we have to pick the right Android studio and Software Development Kit (SDK) for building for Android
Another important component is the Android Native Development Kit (NDK) that enables UE to compile native C++ code. But for installing there is a command line tool that UE shipped with that can install the right version. Because the requirements differ with respect to the version of UE for more information look at Epic Games documentation: Bib.[4].
Other than the configuration for Android, we also need to add the right components for building for the Meta Quest 2, There are two other components useful for developing:

- **MetaXRPlugin:** plugin for: sideload, publish and setting the software for the HMD, this component is already implemented in the UE fork of META.
- **MetaXRSimulator:** simulator for the HMD, this is extremely important because the compile time of the Android Package (APK) is slow. The simulator uses a PC instance of the game, and it overlays it.

For more information look at: Bib.[7]

4 Implementation

This chapter will explain the implemented software solution.

4.1 The 3D models

In this section will talk about how 3D models are saved and rendered on UE

4.1.1 The OBJ file format

For understanding how UE will show 3D models and how they will be stored, we must talk about their characteristics.

- **Vertices:** points that describe the geometry
- **Faces** indicated were there is a polygon by grouping 3 or more vertices
- **Normal:** there is one for each vertex that is in a face, indicates the direction to which the face is exposed, and for calculating how light is reflected
- **UV:** vectors that helps how a texture should be applied to the model
- **Vertex colors:** RGB vector that indicates a color for each vertex

As we talked about in chapter 2, we will use the OBJ file format for storing files. The React Three Fiber has a native support for OBJ, and the backend server does not need to read the file but just to manage by saving, deleting and sending it via HTTP. Unfortunately UE does not have a OBJ file reader usable at runtime but just an importer for what it calls static meshes (3D models that don't have moving parts). So there is the need to build a parser Alg.[1] OBJ to UE custom types.

First we need to understand how the OBJ file format is compose of, here a general example code:4.1

```
# List of geometric vertices and colors, with (x, y, z, R, G, B)
v 0.123 0.234 0.345 0.294 0.960 0.258
v ...
...
# List of texture coordinates, in (u, v) coordinates
vt 0.500 1
vt ...
...
# List of vertex normals in (x,y,z) form
vn 0.707 0.000 0.707
vn ...
...
# Faces
f 1 2 3             # simple
f 3/1 4/2 5/3       # with UV
f 6/4/1 3/5/3 7/6/5 # with UV and normal
f 7//1 8//2 9//3    # with normal
f ...
...
```

Code 4.1: OBJ file example

The vertices, UV and normals are simply written, instead the faces have different formats, first not always they use triangles, but also quads, this depends on how the file was exported. For ease of use the parser will support both. The numbers of the face are the indices of the vertex. Indices start from 1. A face can also have the UV and normals corresponding for the vertex. For our use cases UV aren't needed, but for future-proof they are still being parsed correctly.

Sometimes it is useful to divide the 3D model into multiple objects the OBJ format represents by dividing the 3D model with a "o". OBJ can also divide the faces in groups by dividing them with a "g". Here an example of how the division works: code:4.2

The software that the surgeons are using for exporting 3D models just support groups, so we will implement those, and they will become useful for rendering the model in parts.

4.1.2 To Unreal types

UE has some custom classes for managing things like vectors, colors... These classes also have useful methods that also interface with the blueprint system, we can for

```
o object1_name # declaration of the object name
v...
vn...
g group1_1_name # declaration of a group
f ...
g gorup2_1_name
f...

o object2_1_name
v...
vn...
g group1_2_name
f ...
g gorup2_2_name
f...
...
```

Code 4.2: OBJ grouping

example expose variables or functions, so we can call them at blueprint level. This is very important so that we can interconnect the C++ components with blueprints.

Unreal has a component called `procedural mesh`, this component has the possibility to render a 3D model given vertices and triangles, it also has more data that you can feed to the rendered mesh such as: normals, tangents, UV, vertex colors. It can also have collisions and a material.

The `procedural mesh` also has the possibility to load the mesh in parts, so the parser will save the different triangles in the various groups that are defined in the file. This will be important later for loading time.

Vertices are directly read and saved in an array of `FVector` and normals will be saved in the same way. Vertex colors just need to be read and put in a `LinearColor` array, the object itself can be initialized with the data retrieved in the file. UV because they are 2D vectors will be saved in an array of `Vector2D`. Unfortunately there is a mismatch between how UE manages correlation between vertices and normals with respect to the OBJ file standard. Unreal needs two arrays that contain vertices and normals, so that the vertex in the array at the position `i` must have its normal in the normal array at position `i`. This is still a trivial problem, because there's just the need to load all the normals in memory and then save them again in the correct order decided by the faces.

UVs are being managed in the same way. Another problem is that unreal just accepts triangles and not quads, and because it is a common practice to use quads when exporting 3D models the program will convert quads in triangles, this is pretty trivial,

for each quad we can divide it in two triangles.
Unreal also works in `Z`-up coordinates that means that the `Z` axis points up, there is another standard called `Y`-up were the `Y` axis points up, unfortunately the OBJ file format does not have any ways to reference scale or if the file is saved in `Z`-up or `Y`-up, so it is important to export the file in `Z`-up.

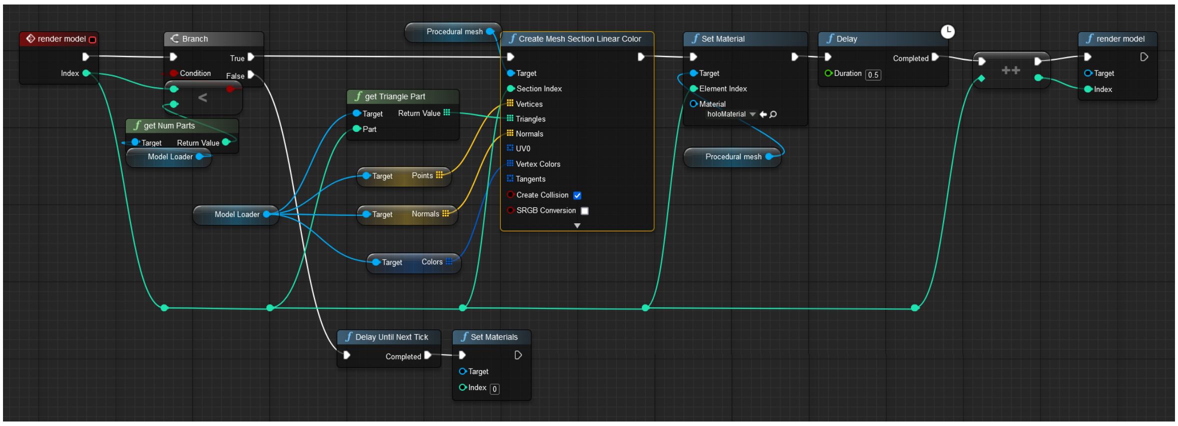


Figure 4.1: Loading mesh funcion

To load the mesh, I created a recursive function Fig.[4.1] that loads each mesh group every 0.5 ms. A delay is necessary because some 3D models are large and can significantly impact performance during loading, and this helps to mitigate that. The function needs to be recursive because UE loops do not allow delays within them. After all the mesh groups are loaded, another similar function is called to change the material and display the 3D mesh colors.
The final result can be seen in the first chapter in Fig.[1.1].

4.1.3 3DModelVewer actor

The `3DModelActor` is a custom actor designed to facilitate the visualization of 3D heart models that is an extension of `Grippable Actor`. This actor comprises four primary components. The first component is the Procedural Mesh, which utilizes Unreal Engine's capabilities to generate and display a heart model in real-time. The second component is a Loading Indicator `Sphere` that provides visual feedback during model loading processes informing users about the ongoing operations. In addition, the `3DModelActor` features a `Text Component` that displays error messages when issues arise during the loading of the 3D models.
The core functionality of the actor is driven by a custom `Model Loader`, This custom

---

Algorithm 1 Parser pseudo code

Require: input file, $vertices$ $colors$ $normals$ UVs vectors, $triangles$ matrix

$normalsTemp \leftarrow []$
$UVsTemp \leftarrow []$
$P \leftarrow -1$ ▷ part index

while file **in** lines **do**
  $L \leftarrow$ file.nextLine
  if L **is** `vertex` **then**
    $vertices$ **add** `vertex`(L)
    $colors$ **add** `color`(L)
  end if
  if L **is** `normal` **then**
    $normalsTemp$ **add** `normal`(L)
  end if
  if L **is** `UV` **then**
    $UVsTemp$ **add** `UV`(L)
  end if
  if L **is** `group` **then**
    $P \leftarrow P + 1$
  end if
  if L **is** `triangle` **then**
    $I_1, I_2, I_3 \leftarrow \textsc{GetVertexIndeces}(L)$
    $triangles[P]$ **add** I_1, I_2, I_3
    $normals[I_1, I_2, I_3] \leftarrow normalsTemp[\textsc{GetNormalsIndeces}(L)]$
    $UVs[I_1, I_2, I_3] \leftarrow UVTemp[\textsc{GetUVsIndeces}(L)]$
  end if
  if L **is** `quad` **then**
    for T **in** $\textsc{Split}(L)$ **do** ▷ Splits the quad in two triangles
      $I_1, I_2, I_3 \leftarrow \textsc{GetVertexIndeces}(T)$
      $triangles[P]$ **add** I_1, I_2, I_3
      $normals[I_1, I_2, I_3] \leftarrow normalsTemp[\textsc{GetNormalsIndeces}(T)]$
      $UVs[I_1, I_2, I_3] \leftarrow UVTemp[\textsc{GetUVsIndeces}(T)]$
    end for
  end if
end while
return vertices, colors, normals, UVs, triangles

---

component is responsible for fetching 3D models from the server, parsing the model data, and integrating it into the procedural mesh and it implements the OBJ parser. The `Model Loader component` can download the 3D Model from the backend thanks to the function httpFileDownload, that thanks to a delegate, can be runned in an asynchronous way thanks to UE threads, and then notify when the execution is complete. Another functionality of `3DModelActor` is that it can be gripped thanks to its parent Blueprint `Grippable Actor`.

```
DECLARE_DYNAMIC_DELEGATE_OneParam(FonFinishLoading, bool, isComplete);

UCLASS(ClassGroup = (Custom), meta = (BlueprintSpawnableComponent))
class VR_API UmodelLoader3D : public UActorComponent {
  GENERATED_BODY()

public:
  // Sets default values for this component's properties
  UmodelLoader3D();
  ~UmodelLoader3D();

protected:
  // Called when the game starts
  virtual void BeginPlay() override;

public:
  // Called every frame
  virtual void TickComponent(float DeltaTime, ELevelTick TickType,
   FActorComponentTickFunction* ThisTickFunction) override;

  TArray<TArray<int32>> trianglesParts;

  UPROPERTY(BlueprintRead, VisibleAnywhere)
  TArray<FVector> points;
  UPROPERTY(BlueprintRead, VisibleAnywhere)
  TArray<FVector> normals;
  UPROPERTY(BlueprintRead, VisibleAnywhere)
  TArray<FVector> tangents;
  UPROPERTY(BlueprintRead, VisibleAnywhere)
  TArray<int32> triangles;
  UPROPERTY(BlueprintRead, VisibleAnywhere)
  TArray<FVector2D> uv;

  UPROPERTY(BlueprintRead, VisibleAnywhere)
  TArray<FLinearColor> colors;
```

```

    UFUNCTION(BlueprintCallable, Category = "loader")
    TArray<FVector>& getPoints();
    UFUNCTION(BlueprintCallable, Category = "loader")
    TArray<int32>& getTriangles();
    UFUNCTION(BlueprintCallable, Category = "loader")
    TArray<FVector>& getNormals();
    UFUNCTION(BlueprintCallable, Category = "loader")
    TArray<FVector2D>& getUv();
    UFUNCTION(BlueprintCallable, Category = "loader")
    TArray<FLinearColor>& getColors();

    UFUNCTION(BlueprintCallable, Category = "loader", BlueprintPure)
    TArray<int32>& getTrianglePart(int32 part);

    UFUNCTION(BlueprintCallable, Category = "loader", BlueprintPure)
    void getAll(TArray<FVector>& pointsVector, TArray<FVector>&
     normalsVector, TArray<FVector>& tangentsVector, TArray<int32>&
     trianglesVector, TArray<FVector2D>& uvVector);

    UFUNCTION(BlueprintCallable, Category = "loader", BlueprintPure)
    int32 getNumParts();
    UFUNCTION(BlueprintCallable, Category = "loader", BlueprintPure)
    void getPart(int32 partIndex, TArray<FVector>& pointsVector,
     TArray<int32>& trianglesVector, TArray<FVector>& normalsVector);

    UFUNCTION(BlueprintCallable, Category = "loader")
    void generate(FString file);

    UFUNCTION(BlueprintCallable, Category = "loader", meta = (Keywords =
     "http"))
    void httpFileDownload(FString name, FonFinishLoading Out);
};
```

Code 4.3: Model loader header file

Resizable functionality As specified in the requirements, the 3D models must be resizable. Therefore, the actor needs an event that allows its size to be changed. This event must be an RPCs that is executed on the server, ensuring that the action is replicated across all clients.

4.2 The VR Character

Thanks to the VRExpansion plugin, we can use the standard `VR Character`, this `Character` has already implemented the online synchronization, and it also has the components for the controllers and camera management. The controllers can `grip` any `Actor` or `Component` that implements the interface `VRGrip`, this will be used for moving the 3D models. Other than that the `VR Character` is an empty blank, and will need to implement some functions for making it fully functional.
Here are the main components to develop:

- Input management
- Widget Interaction
- Interaction pointer
- Side menu
- Grip framework
- 3D model size management
- Loading sphere

4.2.1 Input management

Input in UE is managed by two data files: `Input Action`and `Input Mapping Context`. In the next two paragraphs will be addressed how the input works, and then will be used in the various components of `VR Character`.

Input Action Are files that are used to identify a certain input of the controller, each file should be named after an action more than the input used for making clear what they serve. For example:
For using the `A` button found in the right controller, you need to have a file that represents the button, the necessary settings are: Consume input which allows you to take into account that the input has been served, and the type of value that in this case will be `Digital (bool)`.

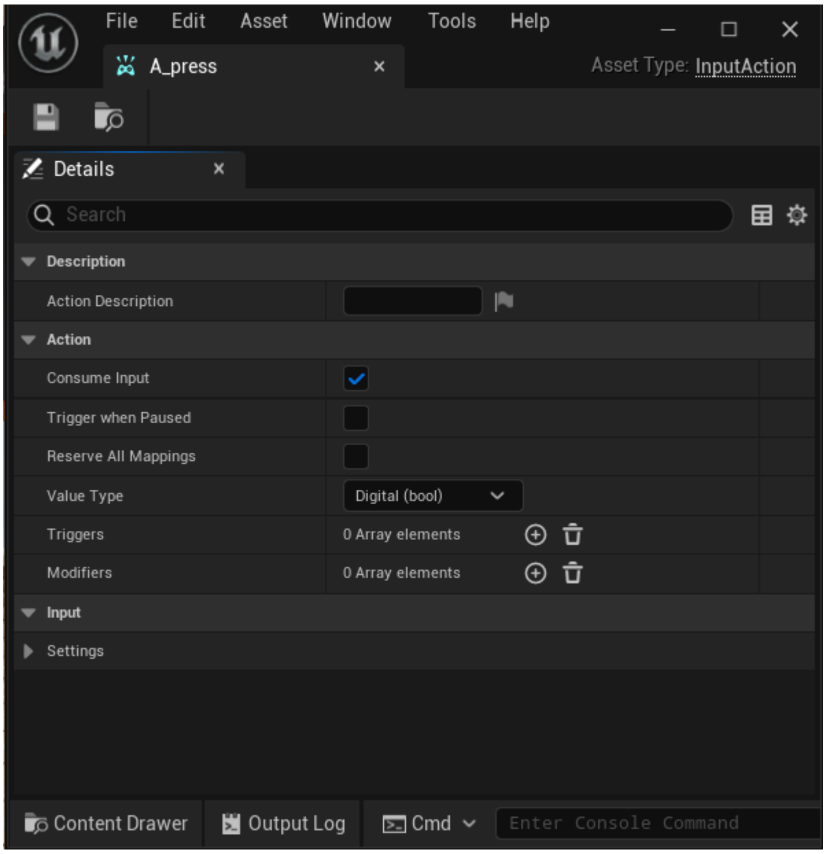


Figure 4.2: Input Action example

Input Mapping Context Files represent all the inputs used by an actor, an actor could change its inputs, so they can be multiple files for different occasions, Here each `Input Action` will be associated with the corresponding input. `Input Mapping Context` can be used for other objects so that they can override the standard behavior of the `Character`, for example by equipping a laser pointer and using the `A` button for toggle it. Each `Input Mapping Context` can be bound with different controllers, this can be useful if we will be porting the app for another HMD. For setting the `Input Mapping Context` there is a function called `Add Mapping Context`.

Input animations Thanks to the rigged 3D models of the controllers provided by Meta, it is possible to import them as `skeletal meshes`.
A `skeletal` mesh consists of a mesh with an underlying skeleton, allowing it to flex, stretch, or rotate its bones to create animations. We use these models to animate the controllers, simulating the user's inputs. This allows for visual representation of buttons being pressed, sticks being tilted, and the degree to which triggers and grips are engaged. These animations are controlled by functions that read the input

and adjust the position or rotation of the `skeletal` mesh bones accordingly.

4.2.2 Widget Interaction

In a normal application we are used to managing input mainly via mouse or touch screens, in VR we can not have that, so It is important to create some kind of UI. One of the most used approaches is showing some kind of virtual display with buttons so that the user can interact, for this will be using a blueprint called `Widget` and will be explained in chapter 4.3. Unfortunately `Widgets` are used for 2D menus but thanks to an actor component we can use it in a 3D environment.
For interacting with a `widget` in a 3D space, UE has a component called `widget interaction` that can evaluate if it is pointing to a `widget`, it can also give the world location of where it is pointing. This component will be attached to each `grip motion controller component`. For letting the user see exactly where the controller is pointed, when the controller is near the `Widget` a trace will be shown. For the trace will be used a Component called `Spline Mesh`, as the name suggests, uses various points and interpolates a mesh for creating a complex curve. Still our use will be simply by just using two points Fig.[4.4]. So the algorithm is simple: each tick a function called `InteractionPointer` will control both controllers if the `widget interaction` points to a `Widget`, then will draw the spline.

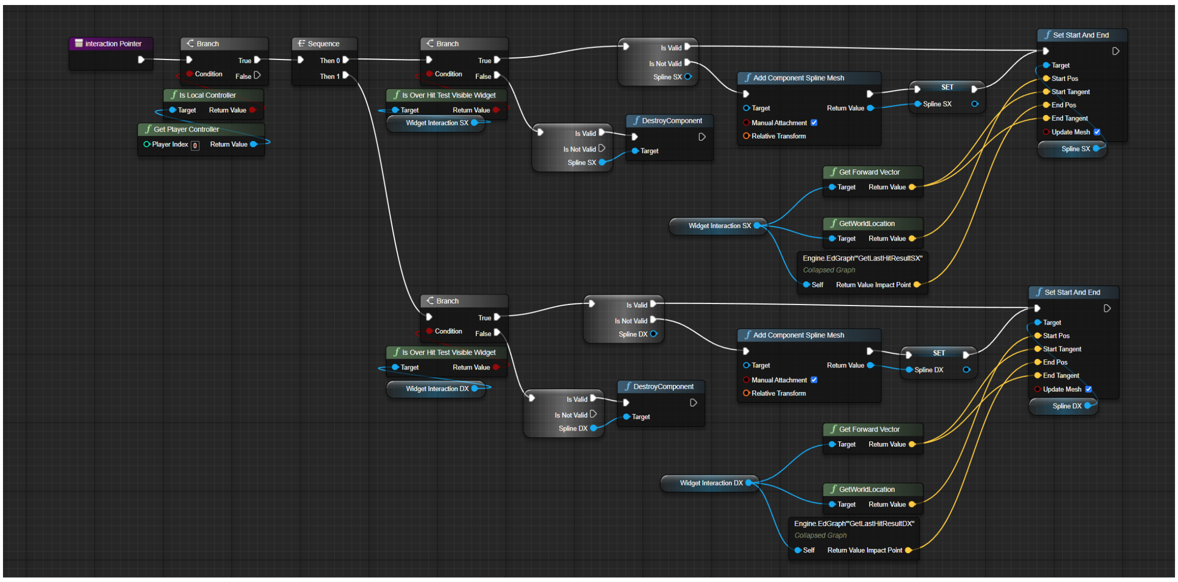


Figure 4.3: Widget Interaction code

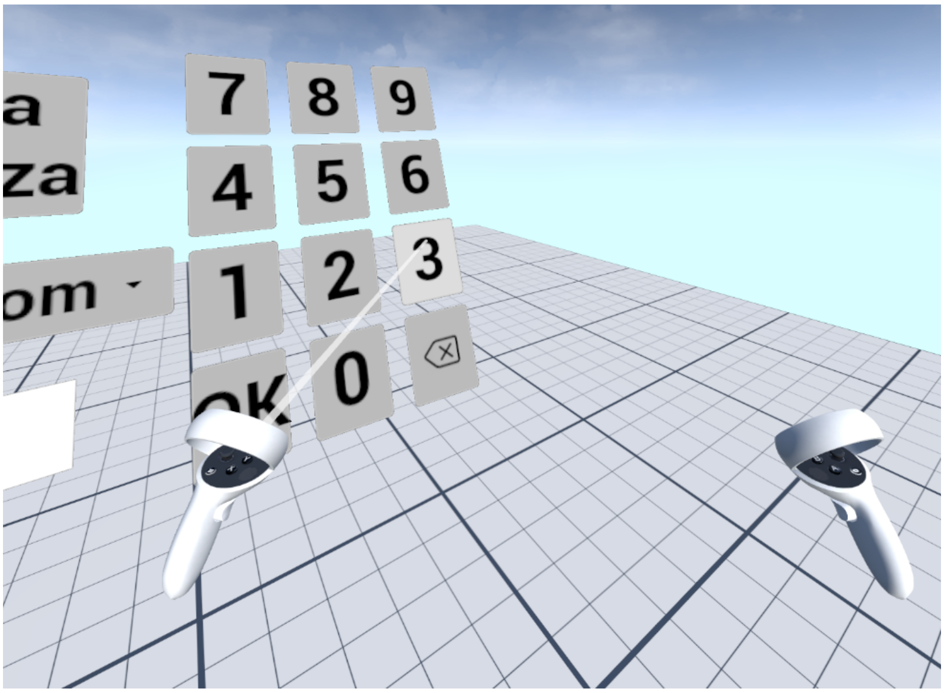


Figure 4.4: On the left a controller that create the spline pointing to the widget

4.2.3 Laser pointer

To create an effective laser pointer, we can simply use a red spline that starts from the controller and ends on the pointed surface. The code is very similar to the `widget interaction pointer` described in Chapter [4.2.2]. However, to find the pointer's location, we can use the line trace function, which detects the nearest point where a geometry intersects with an input segment.
The laser pointer will be an actor, and using a component called a child actor, we can attach it to the `VR Character` right controller. This is necessary to eliminate the latency effect caused by the tick group of the `VR Character`, which is set to `Pre Physic`. The laser pointer actor will stay in the `post update work`, allowing it to be ticked one frame before anything else.
The final result is identical to the `widget interaction pointer` shown in Fig.[4.4], but the color will be red, and it will be toggled using the `A button`.

4.2.4 Grippable system

Thanks to VRExpansion, we can use a grip function that allows us to attach an actor implementing the VRGripInterface. The only challenge is determining which element should be gripped, but UE provides a function called Sphere Trace for

Objects, which returns the nearest actor within a specified radius. If the actor implements the `VRGripInterface`, it can be gripped.
The grip is initiated when the controller grip button is pressed, and the object is dropped when the button is fully released. To avoid conflicts, if both controllers attempt to grip the same object, the object will be dropped when the second controller tries to grip it.

Resizable actor For implementing the resizable functionality, all actors that are resizable will implement the `Resizable actor` interface, that will implement the resizable event.
When two controllers will try to grip the same actor, they will drop it, but if both grips are still pressed, the action of moving the controllers closer and further away will allow you to decide the new size of the actor. For exiting this mode, one controller must release the grip button.
The code for this action is simple, when it is decided that the actor is been resized there the distance between the two controllers will be calculated and the current scale vector of the actor will be saved, after that the size will be decided by the following formula [4.1], for smoothness, it will be executed on every tick.

$$NewSize = StartingActorSize \frac{ControllersDistance}{StartingContrllersDistance} \tag{4.1}$$

4.2.5 Fade camera

As we will explain in Chapter [5], the average VR user can be very susceptible to sudden movements, such as teleportation or moving in 3D space using the analog stick. To mitigate the discomfort caused by sudden teleportation, every time a `VR Character` is teleported or loaded into another level, a fade-in and fade-out animation is applied.
This effect would be relatively simple to achieve using post-processing. However, for mobile development, post-processing is generally discouraged due to its high resource consumption. Therefore, it is important to adopt a less resource-intensive solution. Fortunately, UE materials can be manipulated in real time. By using a sphere with a specific material that has `depth test` disabled (so it renders on top of everything), and by animating the fade effect through a parameter controlling opacity.
The code will be implemented directly in the `VR Character`. The sphere will be

spawned and destroyed as needed, and the animation will be controlled by a built-in function that changes the opacity value over a specified animation time.
Another minor issue is that when the player enters a new session, they will initially spawn at coordinates `X=0, Y=0, Z=0`, where code execution may not yet be possible. To address this, a sphere similar to the one used for the fade effect will be placed at those coordinates in multiplayer levels.
This sphere will be destroyed once the player has been relocated to the correct position. This sphere can be seen in Fig.[4.9].

4.3 Widgets

Widgets are Blueprints used for creating User interfaces in UE, thanks to a lot of pre-made components such as: buttons, textbox, spacers... Each widget can have its own logic built with the Blueprint system.
For the app there will be 3 Menus:

- **Main menu:** used for creating a session and joining it. It will have a number pad for inserting the code to enter a session.
- **Side menu:** used as a summable menù that will be displayed over the left controller, used for exiting the session or letting the host use the centering functionality.
- **3d Model picker:** used for selecting what 3D model the host wants to visualize. It will have a dynamically scroll bar section, so that it can load all the names of the models loaded in the server.

Unfortunately, widgets are normally used for 2D interfaces, but thanks to an actor component it is possible to visualize them in a 3D space, and is it possible to interact with them thanks to the `widget interaction component`.
Because the menus have to do network calls, they will be able to show error messages and have a loading animation, for cosmetic purposes, the animation will be a schema of a beating heart.

4.3.1 HTTP request

For compatibility reasons with the HMD, I built a blueprint function for HTTP requests. Because there is an internet call there is the need to do it in background, fortunately unreal gives a special C++ class called `UBlueprintAsyncActionBase`, if

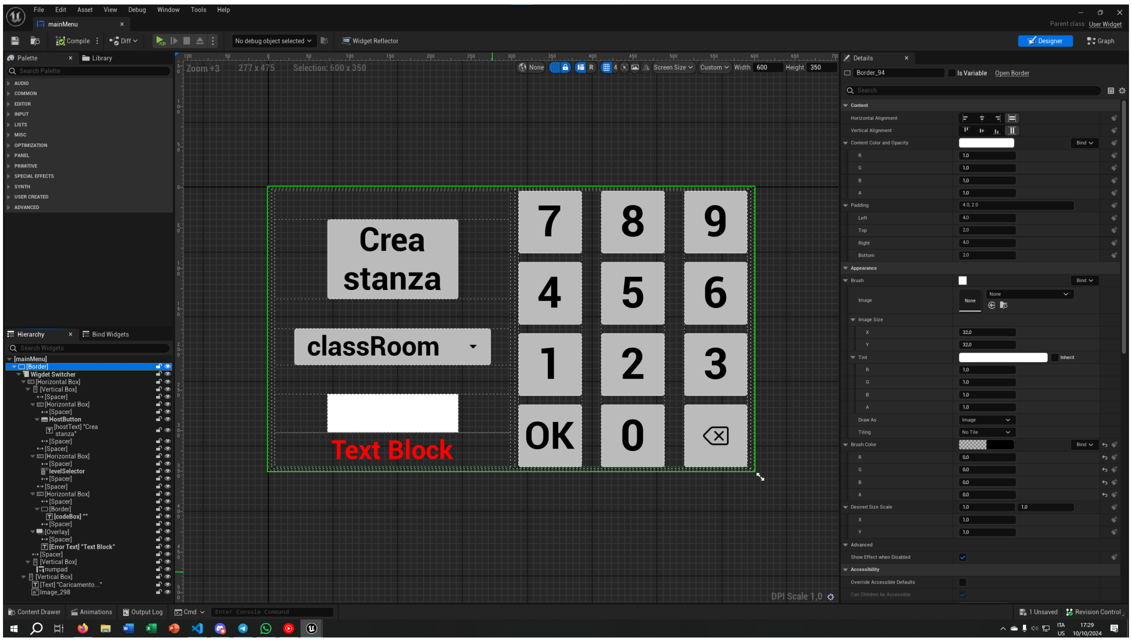


Figure 4.5: Main menu widget

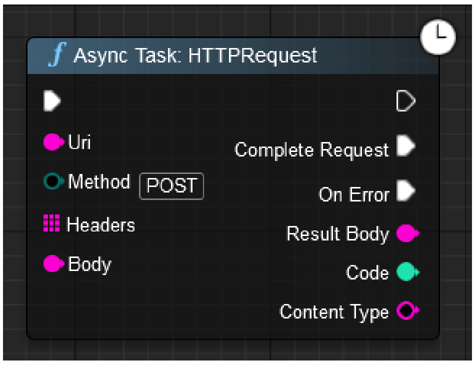


Figure 4.6: HTTP request node

we inherit it, we can create what in UE is called `latent node` Fig.[4.6], these nodes can activate their output pins without stopping the frame until they are complete. Because the app does not need a full HTTP implementation we will implement the block with this parameters an outputs:

- POST and GET methods using an enumeration
- The request headers will be in array of strings
- The request body will be of string type
- The content type response header will be a string type
- The response code

- Two output pins for knowing if it received a response or there were some kind of network error
- The body response will be of string type

For ease of use there is another class called `WorldVariables` that inherits `UBlueprintFunctionLibrary` for accessing the server Uniform Resource Locator (URL), this kind of functions are accessible in all Blueprints and `C++` code.

4.4 The backend

The backend will be built in Python using a library called Flask. This library allows associating a function with a specific URL. Since the app is expected to run in a secure local environment for now, no security measures will be implemented, however, this will change in future developments.
This table explains all the API endpoints Tab.[4.1].
Primarily, the server maintains a list of open sessions, each associated with a timestamp and the host's IP address, with the IP address the client can connect to the host. If necessary, a new session can be opened. When a session is initiated, a 5 digit code is generated and sent to the host. If a client submits the correct code, the server will provide the host's IP address.
Sessions must be refreshed by the server. Every minute, a thread checks for expired sessions and deletes them.

4.5 Multiplayer functionality

Thanks to UE and the VRExpansion plugin, it is relatively simple to establish and maintain consistency between the host and clients. Initially, the host will create a session so that clients can join. Afterward, the professor using the host HMD will announce the session code, which students can enter. Behind the widget logic, Unreal will attempt to check if the provided IP address has an open session on port 7777. Once connected, the host will decide where to spawn the client, with the location chosen randomly. This logic will run on the `Game mode` blueprint
During the session, it is important to replicate the following:

- The location of the controllers and HMD, managed by VRExpansion.
- The location of the 3D model, managed by Unreal. Since clients can also manipulate the model, it is crucial to allow clients to be authoritative so that the 3D model remains synchronized across all HMDs.

URL	**Methods**	**Description**	**Response codes**
/create_session	POST	Creates a session if an IP address is provided in the JSON	200 400
/join_session	POST	Returns the IP address of the host if the code in the JSON corresponding to an open session	200 404
/update_session/<code>	GET	Updates timestamp for session	200 404
/list_files	GET	Returns a JSON array with all OBJ files names	200
/get_file/<filename>	GET DELETE	Returns the OBJ file if exist, or delete it	200 403 404
/upload	GET OPTIONS	Uploads the OBJ to the server, OPTIONS is needed for JS	200 400

Table 4.1: API endpoint

- The toggle state of the laser pointer, so that all players can see it. However, it is unnecessary to replicate the laser spline, as it can be calculated directly on the device.

- Clients teleportation is another important aspect. When the host decides to enter center mode, clients will be teleported to the center. Since only the owning player controller can replicate their own character, the owning clients are responsible for teleporting their characters to the center or to predefined steps.

About center mode Center mode can be enabled by the host via the side menu. This will teleport all players to the center and disable the visualization of controllers and HMD meshes for non-owning players. Additionally, it is important to teleport the clients to the same location on the lecture hall steps to avoid creating confusion among students. To achieve this, when a new player joins the session, their starting point will be saved in their own `player controller`, making it easily accessible.

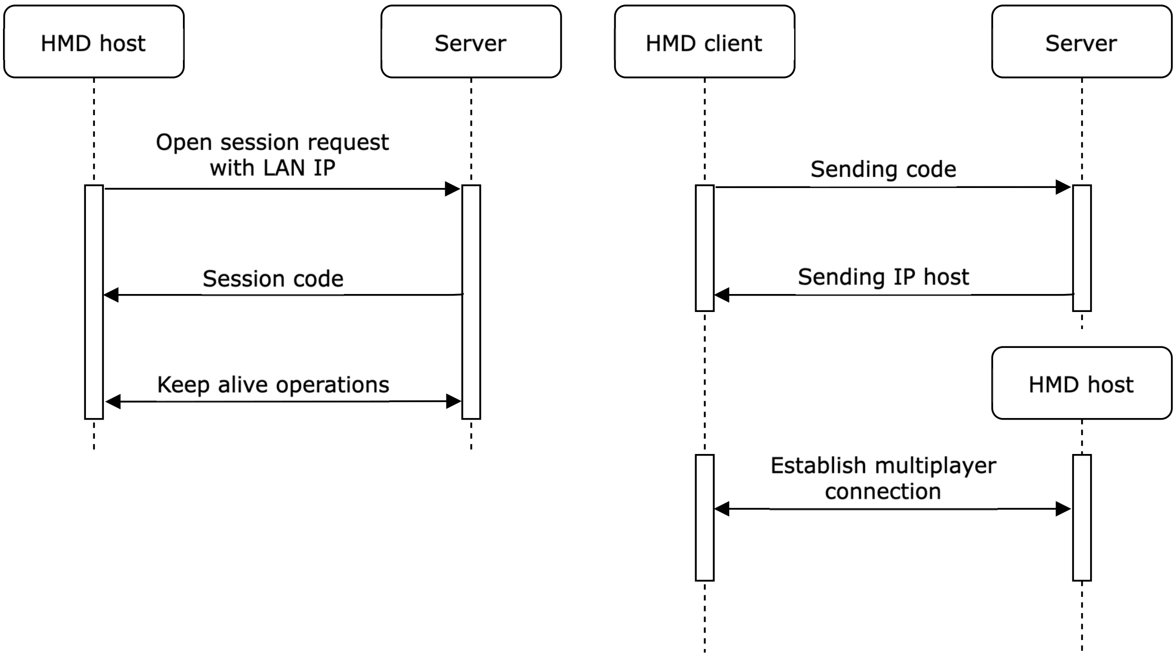


Figure 4.7: Network time diagram

4.6 Virtual Environment Development

Creating a 3D virtual environment is a complex task that requires the collaboration of multiple individuals with diverse skill sets, such as level designers, concept artists, and 3D artists. An easier solution is to use freely available assets from the Unreal Marketplace. The asset pack I found, called Assetsville Town[1]. These assets are ideal for use with the HMD due to their low-poly nature, including models such as chairs, books, blackboards, general office appliances and tile sets for creating various environments. Additionally, the pack provides a wide range of materials and textures for further customization. In the following sections, I will explain the two environments developed for the application.

4.6.1 Main menu

The main menu Fig.[4.8] is the first interface the user encounters upon entering the application. A table in the virtual environment will display the main menu UI widget, allowing the user to create or join sessions. Beside it, another table will host the `3DModelPicker` widget, enabling users to interact with individual 3D models.

[1] https://www.fab.com/listings/fd558d8c-bd7e-461f-8449-a7cc9c277078

Figure 4.8: Main menu screenshots

4.6.2 Classroom

The classroom Fig.[4.9] will be designed as a lecture hall, with students seated on steps to ensure a clear view of the 3D models, while the professor will be positioned in the middle. A blackboard at the center will display the session's code, and the bottom-right corner will indicate whether the center mode is enabled.

4.7 Web app

As described in the requirements, there is a need to display different 3D models from time to time. This functionality will be achieved through a React web app build. The main functionalities are:

- Upload 3D models

Figure 4.9: Classroom screenshots

- Visualize already uploaded models
- Delete 3D models
- Responsive design

The UI will be built using MUI[2], along with a free template provided by them. For 3D visualization, there is a library called React Three Fiber[3], which is essentially a wrapper around Three.js[4]. React Three Fiber was chosen because it easily supports the visualization and caching of 3D models, as well as vertex colors for OBJ files. The only issue is that it natively displays OBJ models with `Y-up` orientation. To resolve this, the model simply needs to be rotated on the `X` axis by -90°.

[2] `https://mui.com`
[3] `https://r3f.docs.pmnd.rs`
[4] `https://threejs.org`

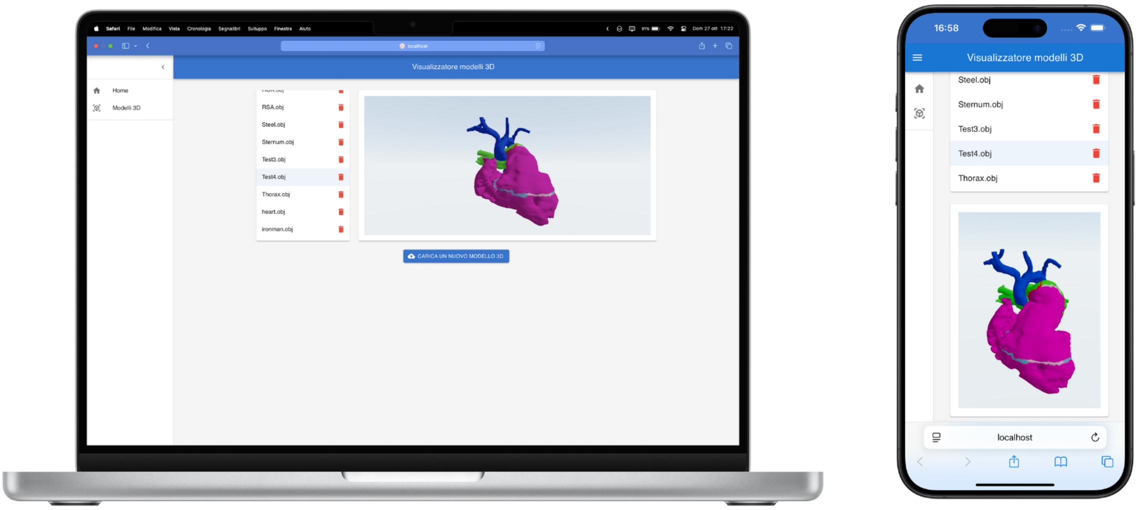


Figure 4.10: Web app mockups

5 Performance and optimization

Running a VR application is highly resource-intensive. Not only must the processor handle all sensor data, but it also has to run a 3D application that is rendered on two screens, aiming to achieve at least a stable 72 FPS experience. On top of that, the Meta Quest 2 has limited mobile hardware Tab.[5.1]. This chapter will explain all the optimizations necessary to run the application at 90 FPS, as well as demonstrate the performance of the OBJ parser build.

5.1 Meta XR performance check

Thanks to the Meta XR plugin, Meta directly suggests certain options to enable or disable. The most important aspects will be explained in the next paragraphs.

Post-processing must be disabled because it can be very demanding on the device. As discussed in Chapter 4.2.5, mobile HDR is already turned off, however, UE still

Characteristic	**Specs**
Chipset	Qualcomm Snapdragon XR2
CPU	Octa-core Kryo 585 (1×2.84 GHz, 3×2.42 GHz, 4×1.8 GHz)
GPU	Adreno 650
RAM	6 GB LPDDR5

Table 5.1: Meta Quest 2 specs

enables lens flares by default. To disable it, we need to place a `Post Process Volume` in the scene and set the lens flare intensity to `0`.

Dynamic lighting must be disabled. This option is probably the most impactful. When enabled, the app runs at less than 60 FPS, but with it disabled, performance reaches a stable 90 FPS, yielding an improvement of over 30 FPS and providing a more stable experience overall. Unfortunately, dynamic lights can enhance immersion, however, the only moving elements are the `VR Characters` and the `3D Model Viewer`, so the absence of dynamic lighting will not be very noticeable.
Static lighting will still be used, allowing the environment to have shadows cast by a directional light that simulates the sun, as well as by rectangular and point lights used for building interiors.

Half precision float for materials and shaders. As is well known, using smaller float values can improve system performance by sacrificing some precision, though fortunately, the side effects are often negligible. However, in our case, this optimization has limited impact because the majority of materials used are static.

5.1.1 Other optimizations for Meta quest

In addition to the previously described options, there are some additional settings that can help boost performance. These are simple Android manifest tags that enable the HMD to activate additional functionality:

- **Suggested CPU and GPU levels:** This setting allows the HMD to operate at maximum performance levels.
- **Processor favor:** This option lets us choose whether the HMD should prioritize maximum performance for either the GPU or CPU. This is useful because the device has limited power and heat dissipation, preventing it from running at 100% performance continuously without the risk of throttling.
- **Enable dual core:** Normally, the HMD uses only one core for the foreground app, but a meta tag enables a second core for background activities and parallel operations. This is particularly beneficial for the parser and asynchronous operations needed by UE.

5.2 Optimizations for Unreal engine VR

In addition to the options provided by the Meta XR performance check, there are settings within UE Bib.[6] that can further optimize the app for the HMD.These optimizations are explained in the following paragraphs.

Instanced stereo When rendering a frame for an HMD, the screens are typically managed as two separate entities. This is demanding for the GPU, as it has to render two frames simultaneously. However, with Instanced Stereo, we can generate both views in a single pass. UE achieves this by applying a small transformation to the loaded vertices, correcting the view difference between the left and right eyes. Additionally, this transformation is applied to shaders as well. This rendering method relies on low-level APIs that enable it, the Meta Quest 2 uses the Vulkan graphics API, which supports this feature.

5.2.1 Variable Rate Shading and Fixed Foveated Rendering

We don't always need to shade each material with maximum precision, so UE provides tools to help mitigate this.

Variable Rate Shading (VRS) is a technique that allows shaders to render at a lower resolution in specific situations. Normally, the standard shading rate is 1:1, meaning one pixel shader invocation corresponds to one pixel target on the screen. UE offers various shading rates, for instance, applying a shading rate of 2x2 means that a single shader invocation will cover a 2x2 pixel area, resulting in one shading calculation for every 4 pixels.

Fixed Foveated Rendering (FFR) is a technique that utilizes VRS to determine which parts of the screen can be rendered at a lower resolution. Studies, such as Bib.[1], have shown that when viewing a screen, the eye typically focuses on the center. This allows for lower resolution rendering in the peripheral areas. Consequently, as the distance from the center increases, we can apply a lower shading rate, conserving resources without noticeably impacting visual quality.
An example of FFR zones is in Fig.[5.1] where the darker blue represents the lowest rate shading zone.

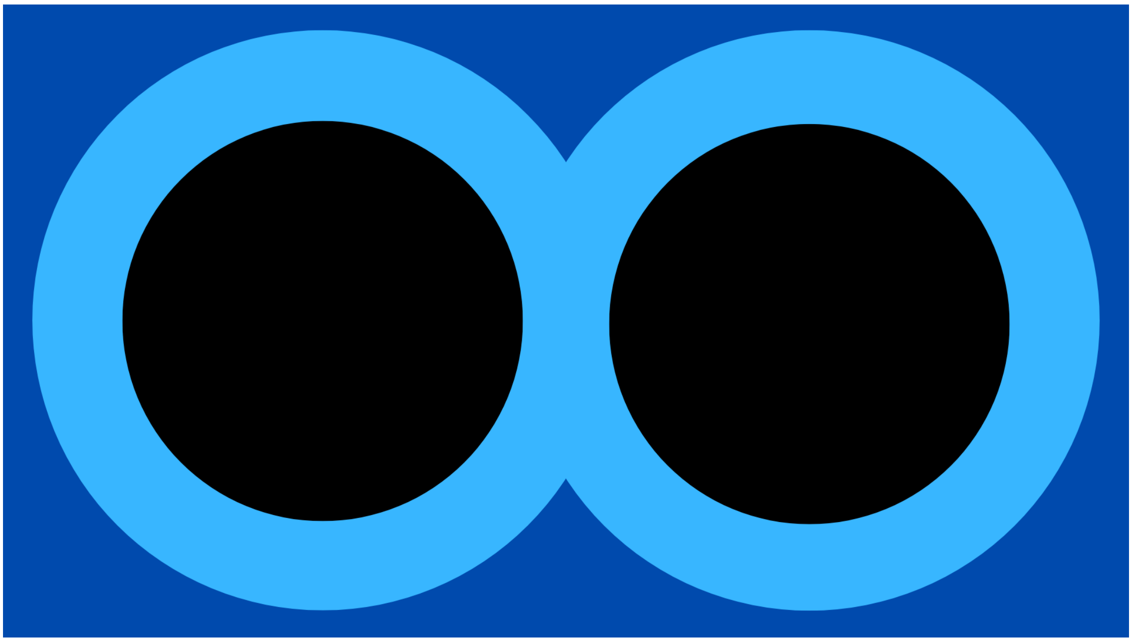


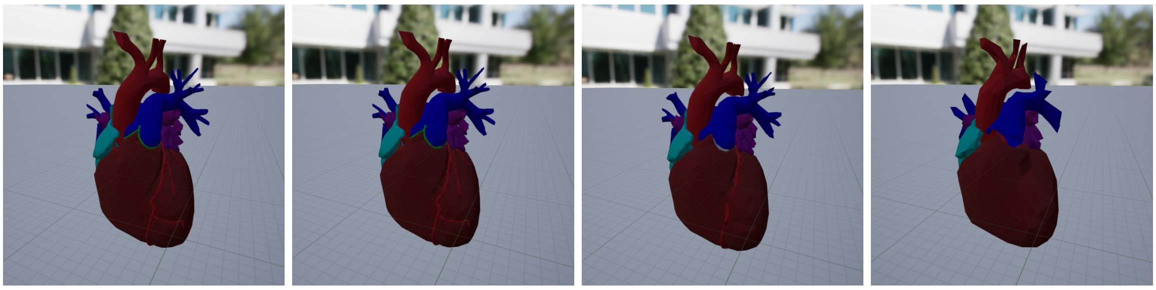
Figure 5.1: Example of FFR zones

As expected, FFR is not an optimal approach, as the eye occasionally shifts to the periphery of the HMD display. In the future, HMDs are likely to be equipped with eye-tracking modules, allowing FFR to be replaced by dynamic foveated rendering, which adjusts rendering zones based on where the user is looking.

5.3 General engine optimizations

UE is one of the most important game engines on the market, offering many standardized features to optimize various types of games.

Level Of Detail (LOD) were created to reduce the number of vertices and triangles in a scene. The concept is straightforward: small or distant objects do not require as much detail as closer or larger ones. Based on their position and size in the scene, each mesh has different LOD levels that load according to the scene's requirements. LOD levels can be generated programmatically or modeled by a 3D artist, naturally, manual creation leads to more accurate LOD.
LODs can modify materials or reduce the number of triangles and vertices. An example of programmatically generated LODs that reduce polygon count can be found in Fig.[5.2].



100% 25% 10% 2%

Figure 5.2: LODs and their percentages of a model of 96.180 triangles

Removing opacity: Opacity is one of the heaviest options in shading, so for reducing GPU overhead it will be removed.

Clever level design: When watching 3D animations or playing video games, we might assume that every environment is fully constructed with maximum detail. This is not the case. For example, in this case study, it's unnecessary to build parts like the exterior of a building or create a second floor. Similarly, distant mountains may appear to have realistic depth, but they are often floating in empty space and are not actually far away. This approach prevents the engine from loading hidden models, which would otherwise waste resources.
For the same reason, shaders are applied only to one side of a triangle. For example, shading the inner side of thick building walls is unnecessary and would only consume resources without adding to the visual quality.
These tricks can been seen in Fig.[5.3]

5.4 General performance

One of the primary requirements was for the application to run at 90 FPS for optimal comfort, and this target was achieved. As shown in Fig.[5.4], the app maintains a stable 90 FPS, with minor performance drops occurring mainly during rendering and manipulation of 3D models.
To minimize these performance drops further, reducing the polygon count of the 3D models may be necessary to lighten their GPU load. Fig.[5.4] also illustrates that the bottleneck lies with the GPU, which handles rendering and manipulation of the 3D models.

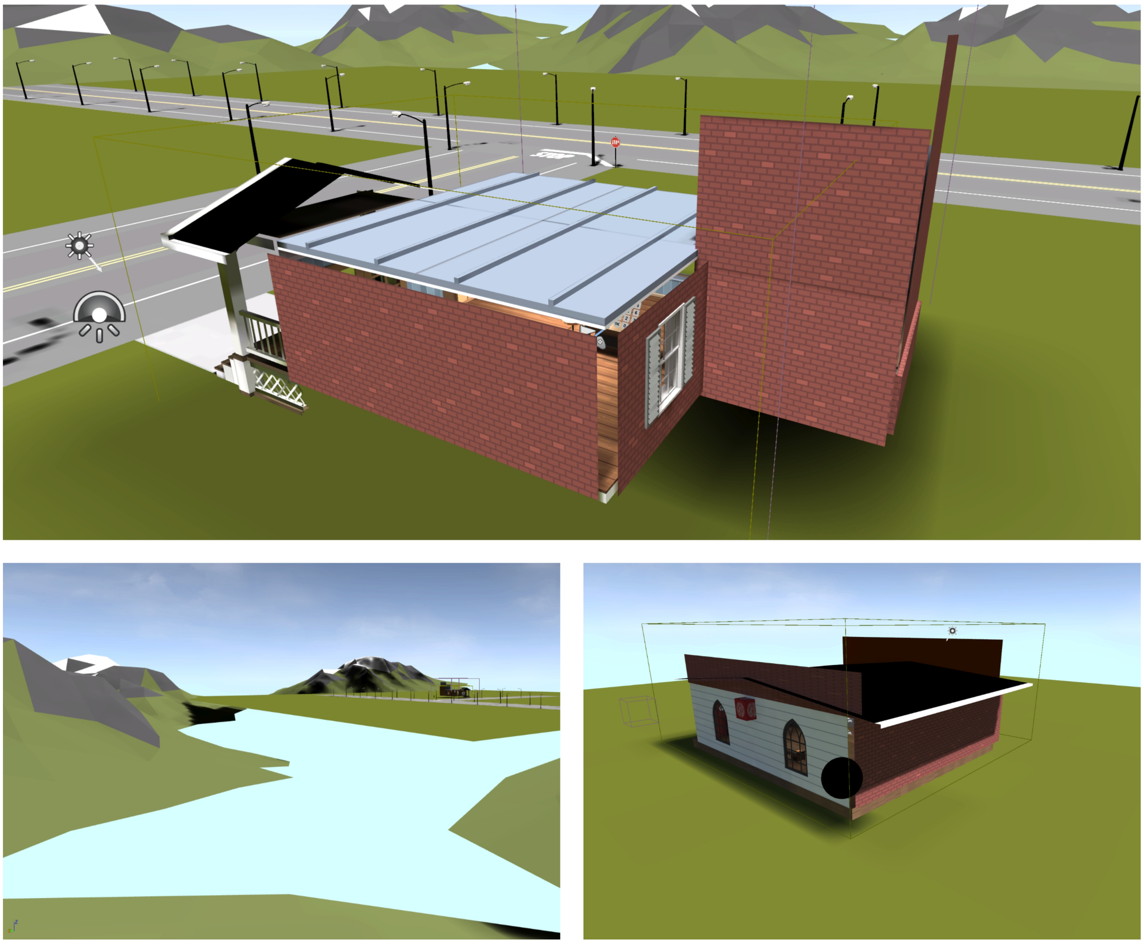


Figure 5.3: Exterior of buildings

5.5 Parser performance

The parser performance must be efficient enough to avoid making the user wait too long. Fortunately, there is a generous upper time limit, as the models need to be downloaded first. Ideally, parser performance should be $O(n)$, however for cleaner code, we are using a function called `ParseIntoArrayLines`, which returns the file as an array of its rows. According to the UE documentation[1], this function is expected to have $O(n^2)$ complexity in terms of allocation. However, my results suggest this may not be accurate—likely due to processor cache compensating for access speed.

[1] `https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Runtime/Core/Containers/FString/ParseIntoArrayLines?application_version=5.2`

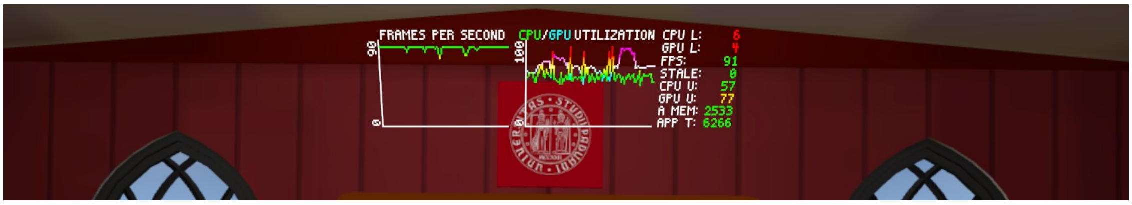


Figure 5.4: Performance graph

Size	Vertices	Triangles	Parsing time avg	Download time avg
188kB	1284	2568	0.013700s	0.160968s
276kB	1856	3704	0.020405s	0.198955s
540kB	3590	7192	0.037820s	0.215038s
724kB	4768	9564	0.050700s	0.230261s
1.7MB	10968	21968	0.116693s	0.324465s
6.2MB	38207	77474	0.412519s	0.732149s
9.7MB	60333	120774	0.630291s	0.985465s
11MB	62674	126376	0.670590s	1.120609s

Table 5.2: Parser loading time

Benchmark Table Tab.[5.2] presents benchmark results showing the parser performance alongside the network time required to download it. The test was conducted using a developer build of the app with all functionalities enabled. The connection was Wi-Fi 5 (up to 866 Mbps), with the access point approximately 2 meters from the HMD. The router was connected to a 1 Gbps fiber line, providing ideal network conditions. As shown, the time to download the model is the real bottleneck and in real-world conditions, however, the HMD may not benefit from such optimal network connectivity.
Thus, we can conclude that the parser performance is adequate for these use cases.

6

Development and testing

This chapter describes the methods used for software testing.

6.1 Feedbacks from surgeons

The development of this thesis has been conducted in close collaboration with cardiac surgeons who are the main stakeholders of the project. The development followed an iterative approach, Through a series of iterative meetings and discussions with the surgeons, we could assure the alignment of the software being developed with the requirements.
During the year of development, periodic testing sessions have been organized to assess the current state of the project and decide some development details, mainly concerning the user experience. Some of these sessions involved a reduced number of users, while others were done inviting a group of people to recreate the setting of a real lesson. These collaborative sessions provided an opportunity to gather detailed feedback, which was then systematically integrated into the design and implementation phases. This ongoing dialogue not only allowed for continuous improvement but also fostered a multidisciplinary approach.
The iterative development process led to a solution that is both technologically robust and highly tailored to the specific needs and expectations of the end users.

Figure 6.1: Testing session

6.2 Science4All

A special mention goes to the test of the application during the event Science4all hold in Padova in September 2024. Science4All is a science outreach and inclusion software in Padua, aimed at making complex scientific topics accessible to a broad audience, from young people to adults without a scientific background.
During the 2024 event, the VR app was presented in the 3D printing section focused on medical applications. For the occasion, considering the young age of the intended public, a specially designed environment Fig.[6.2]. The environment was designed to allow an operator to hide the hearts by shrinking them and concealing them within

the office equipment in the room. A group of three children could then try to find all the hidden hearts. By doing so, we allowed children to try the HMD, for many, it was their first time experiencing VR. They had the opportunity to view a heart model, exploring both the exterior and interior in detail. Both children and parents were captivated by the app capabilities and the medical use case for which it was developed.
This provided valuable insights into additional use cases, new features, and bugs, thanks to the children's feedback and imagination, which will be useful for future development.

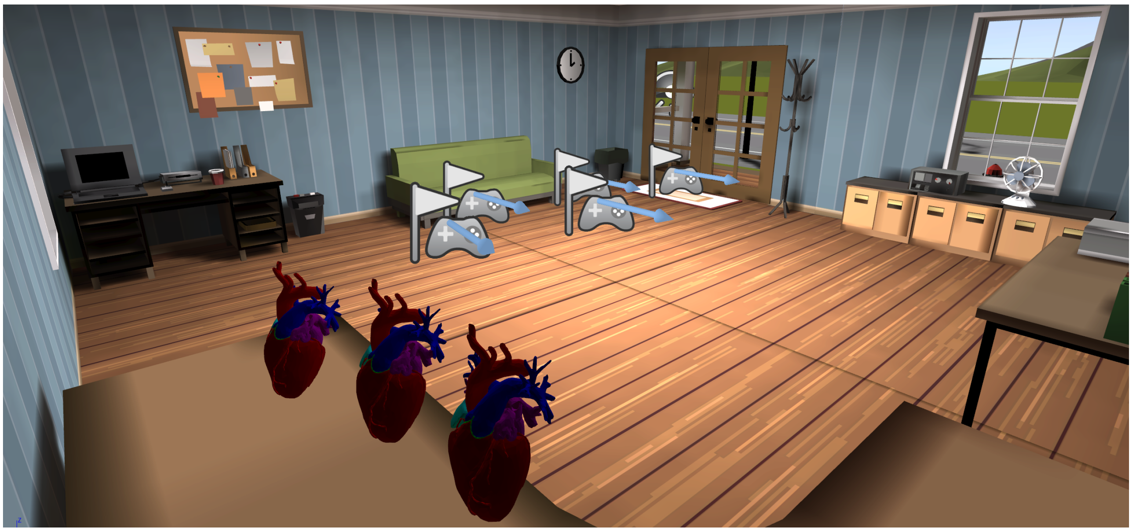


Figure 6.2: Science4All environment

7 Future work

In this thesis, we successfully designed, developed, and tested a comprehensive software solution that meets all the requirements specified by the surgeons.
As with any software development software, there is always room for additional features and refinements. The following paragraphs outline potential future work to enhance the functionality and overall outcome of this software.

7.0.1 Publication in the store

The software is still in its prototype stage, so it requires further attention before it can be published on the Meta Quest Store. The server is currently in its prototype stage and needs to implement user authentication. It must also be deployed in a public domain to allow users outside the university to access the software. Additionally, it should be scalable, so it can manage a number of users that the server is currently unable to support. To make this software more desirable to a larger audience, it could be enhanced with additional features, as explained in the following sections.

7.0.2 Slicing 3D models

At present the app allows viewing the model from the exterior. To view it from the interior, the user must "walk" inside the model, to improve this experience a special cut out model should be created and loaded so that different parts of the heart can be seen individually. Another useful way to view the inside of models could be a tool operated by the teacher or user that allows to slice the model along planes so that surgeons could gain deeper insights from the 3D model if it could be sliced along planes, like CT scans and MRI, this could be done locally thanks to UE functions.

7.0.3 Real time coloring

At present, the app supports colored models, but the colors need to be set before the rendering and, as described before, they are encoded in the model itself. As heart models can be quite complex, a real time coloring tool could be a useful tool during a lecture, to highlight some surfaces of the heart by coloring them to emphasize particular areas.

7.0.4 Visualization of other media

By leveraging the capabilities of VR to display various types of media, it will be beneficial to include images or videos of CT scans and MRI, providing surgeons with multiple resources to work with. Thanks to VR this media can be shown in very different ways, we could build a simple virtual monitor to see them but also be more creative by recreating in VR devices like tablets or smartphones.

7.0.5 Procedural LODs for 3D models

Because the 3D models used are quite demanding in terms of the number of vertices and triangles. If the backend could dynamically generate LODs, the HMD would benefit from significant performance gains. The algorithms required for creating LODs are quite demanding in time execution, this is why it should be done on the server. Additionally, the OBJ file format is not suitable for storing such data, so transitioning to a more advanced file format like FBX or GLB would be advisable.

7.0.6 3D models caching

Because lectures can be highly dynamic, requiring the display of various heart models. Additionally, previously used 3D models could be downloaded and cached on the HMD, reducing the time needed to load and display them. To implement it, we need to create some caching rules and update the server APIs for enabling this functionality.

8

Conclusions

The development of the VR app enabled us to create an immersive training experience for cardiac surgeons. The process was challenging due to long development cycles, primarily caused by the need to compile APK files for testing on the HMD.
The software not only represents an application of the VR technology, but also the features that the UE has in developing not only VR applications but also in general 3D applications.
As explained in Chp.[7] the software is far from finished but it is an opportunity for the University of Padua to create a unique application for teaching.

Bibliography

[1] Markus Bindemann. "Scene and screen center bias early eye movements in scene viewing". In: *Vision Research* (2010). URL: https://www.sciencedirect.com/science/article/pii/S0042698910004025.

[2] Vladimiro Vida Fabio Gaccione. "3D printing of cardiac valves from multimodal imaging for simulation and planning of cardiac surgery procedures". Available at https://hdl.handle.net/20.500.12608/73605. MA thesis. University of Padua, 2023.

[3] Vladimiro Vida Francesco Bertelli. "Advanced 3D-Based Teaching Models in Congenital Cardiovascular Disease". MA thesis. University of Padua, 2019-2020.

[4] Epic Games. *Getting started with Android*. https://dev.epicgames.com/documentation/en-us/unreal-engine/how-to-set-up-android-sdk-and-ndk-for-your-unreal-engine-development-environment?application_version=5.2.

[5] Epic Games. *Getting started with Visual Studio*. https://dev.epicgames.com/documentation/en-us/unreal-engine/setting-up-visual-studio-development-environment-for-cplusplus-projects-in-unreal-engine?application_version=5.2.

[6] Epic Games. *XR Performance Features*. https://dev.epicgames.com/documentation/en-us/unreal-engine/xr-performance-features-in-unreal-engine?application_version=5.2.

[7] Meta. *Creating Your First Meta Quest VR app in Unreal Engine*. https://developer.oculus.com/documentation/unreal/unreal-quick-start-guide-quest.

[8] Vladimiro Vida. *Congenital heARts - A Virtual Atlas of Congenital Heart Disease in Prenatal and Postnatal Life*. Bantam, 2023. ISBN: 978-88-299-3335-8.

Acknowledgments

I would like to acknowledge the creators of Unreal Engine (Epic Games), Unity (Unity Technologies), and Meta for providing the development platforms used in this work. This research is not affiliated with, nor endorsed by, these companies.

The development of this project made extensive use of Unreal Engine for the creation of immersive experiences, and Meta VR devices provided the necessary hardware for testing.

I would like to acknowledge the developers of VRExpansion, which has been used under the MIT License in this project. The license permits free usage, modification, and distribution with proper attribution to the original authors.
link to the license:
`https://github.com/mordentral/VRExpansionPlugin?tab=MIT-1-ov-file`