Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

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

To my parents
and friends

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

VR	Virtual Reality
PC	Personal Computer
OS	Operative system
HMD	Head Mounted Display
JSON	JavaScript Object Notation
API	Application Programming Interface
UE	Unreal Engine
SDK	Software Development Kit
NDK	Native Development Kit
REST	Representational state transfer
HTTP	HyperText Transfer Protocol
APK	Android Package
IP	Internet Protocol
UI	User Interface
RPCs	Remote Procedure Calls
FPS	Frame Per Second
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

1 Introduction

1.1 Case introduction

At the university hospital of Padua, there is a very important cardiac surgery center where various heart interventions are performed on many people, the need to teach and visualize the case of operation is very important.
Thanks to MRI, surgeons can see not only pictures of the heart, but also can even make 3D models. Each 3D model can show every detail of the patient's heart, this can make surgeons able to analyze and show where and how to resolve the problem.

1.1.1 Virtual reality head mounted displays

The equipment: The equipment: The university of Padua has some Meta quest 2[1] [fig:1.1] a Head Mounted Display (HMD) for Virtual Reality (VR).
The Meta Quest 2 is a standalone HMD, this means that it doesn't need other peripherals like an external console or Personal Computer (PC) for working.
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.

[1] All rights to meta reserved

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.

Figure 1.1: Meta quest 2

Use cases of VR: Principally the surgeons are using VR equipment for training and showing critical health conditions of different patients' hearts. They are using an APP called Shapes XR, this APP has a web interface for uploading 3D models and then showing them on the HMD. 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, they tested with 14 users connected and there weren't any problems.

1.1.2 APP PROBLEMS

How Shapes XR works: Shapes XR 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

The Problems: Unfortunately Shapes XR 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 because for a lot of people 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 Shapes XR 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.

Feedbacks from surgeons and nurses: There was a lesson on 12/12/2023 with the integration of the Quest 2 and Shapes XR. 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

How we can see a new app for this use case could be useful, and this is the main reason why this project exists.

2 Requirements

2.1 Functional Requirements

The main features of the project 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 web portal 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.

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 Preliminary project

The project is made by three main components: VR APP, backend, Web User Interface (UI). This chapter will talk about them.

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 UI The Web app will let the professor to upload any OBJ models needed for the lecture by a simple form, and the web app will allow 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 project 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 project 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 project 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 project.
Like a lot of game engines, UE tries to render many frames as much 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 `VRpawn` 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:

- **VRcharacter:** 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 listening 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 Listening Server does not need a Stand-Alone server, simply one device acts

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

not only as a client but also as a server. Normally client-server is the best in performance and latency, but because this software is simple, the listening 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, its 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 Web Portal for managing the 3DModels. The multiplayer itself will be manage by the host HMD

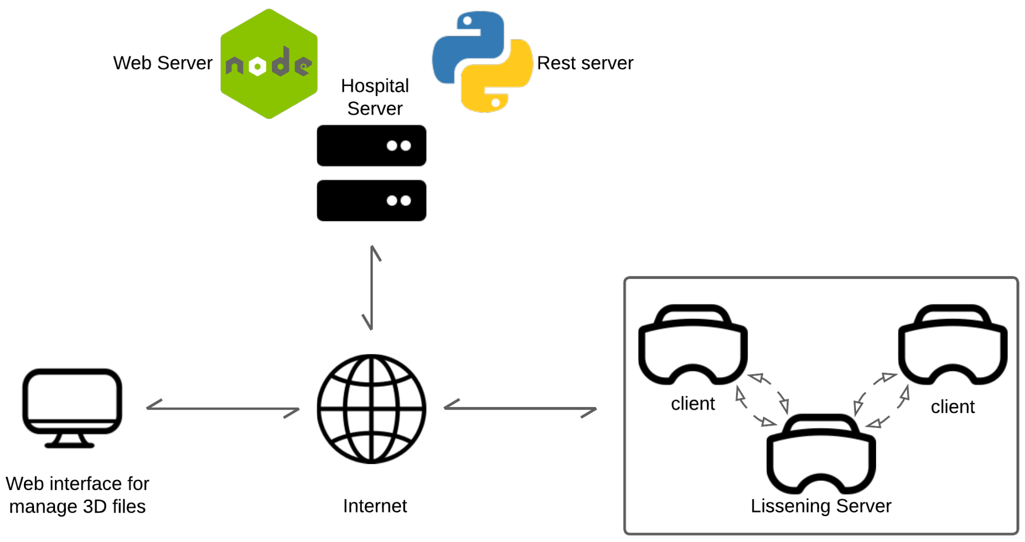


Figure 3.1: Network Schema

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 listening server. The library used is called Flask[2], it simply lets you run a function with respect to a HTTP response received.

The Web UI: The Web UI 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 UI library called MUI[3] and a 3D

[2] `https://flask.palletsprojects.com/en/3.0.x/`
[3] `https://mui.com/`

library for rendering 3D models called React Three Fiber[4].
The main functionality of the Web UI will be:

- previewing 3D models with the right colors
- upload 3D models
- delete 3D models

3.5 Development environment

It is important to set the right development environment for this project, even if for the Web UI 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 project, 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: [2] 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: [1].
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 project 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: [3]

[4] `https://r3f.docs.pmnd.rs/getting-started/introduction`

4 The project

This chapter will explain the development behind the project by starting from the fundamentals.

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 [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.

```
o object1_name # delaration of the objet 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

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 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 us: 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 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 respect 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.

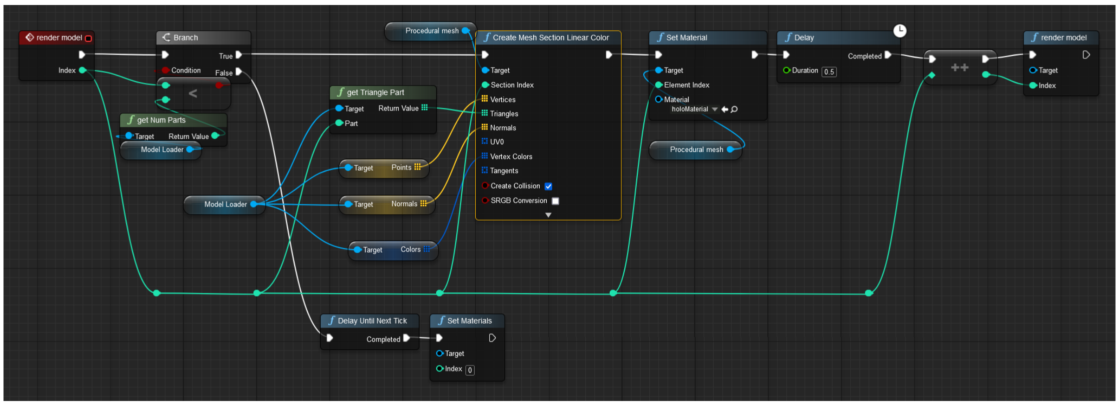


Figure 4.1: Loading mesh funcion

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.

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 on-

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
      $I_1, I_2, I_3 \leftarrow$ GetVertexIndeces($L$)
      $triangles[P]$ add $I_1, I_2, I_3$
      $normals[I_1, I_2, I_3] \leftarrow normalsTemp$[GetNormalsIndeces($L$)]
      $UVs[I_1, I_2, I_3] \leftarrow UVTemp$[GetUVsIndeces($L$)]
    end if
    if $L$ is quad then
      for $T$ in Split($L$) do             ▷ Splits the quad in two triangles
        $I_1, I_2, I_3 \leftarrow$ GetVertexIndeces($T$)
        $triangles[P]$ add $I_1, I_2, I_3$
        $normals[I_1, I_2, I_3] \leftarrow normalsTemp$[GetNormalsIndeces($T$)]
        $UVs[I_1, I_2, I_3] \leftarrow UVTemp$[GetUVsIndeces($T$)]
      end for
    end if
  end while
  return vertices, colors, normals, UVs, triangles
```

going 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 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 `VRCharater`, 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 `VRCharater` is an empty blank, and will need to implement some functions for making it fully functional. Here are the main components to develop:

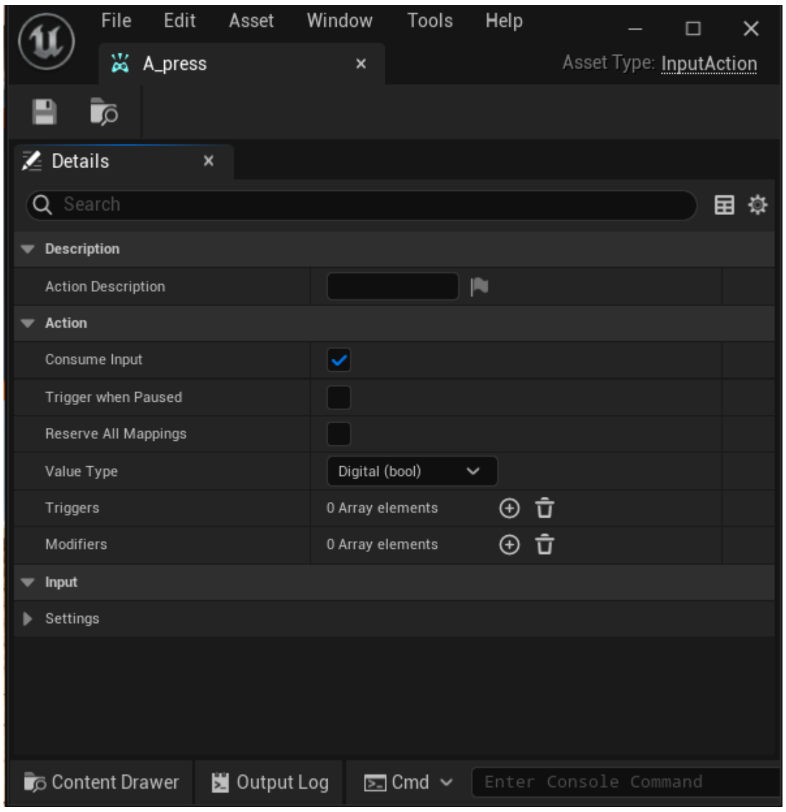
- Input management
- Widget Interaction
- Interaction pointer
- Side menu
- Grip framework
- 3D model size management
- Loading sphere

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 `VRCharater`.

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 find 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)`.



Description
Action Description
Action
Consume Input ✓
Trigger when Paused
Reserve All Mappings
Value Type: Digital (bool)
Triggers: 0 Array elements
Modifiers: 0 Array elements
Input
Settings

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's 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.

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 `VRCharacter` right controller. This is necessary to eliminate

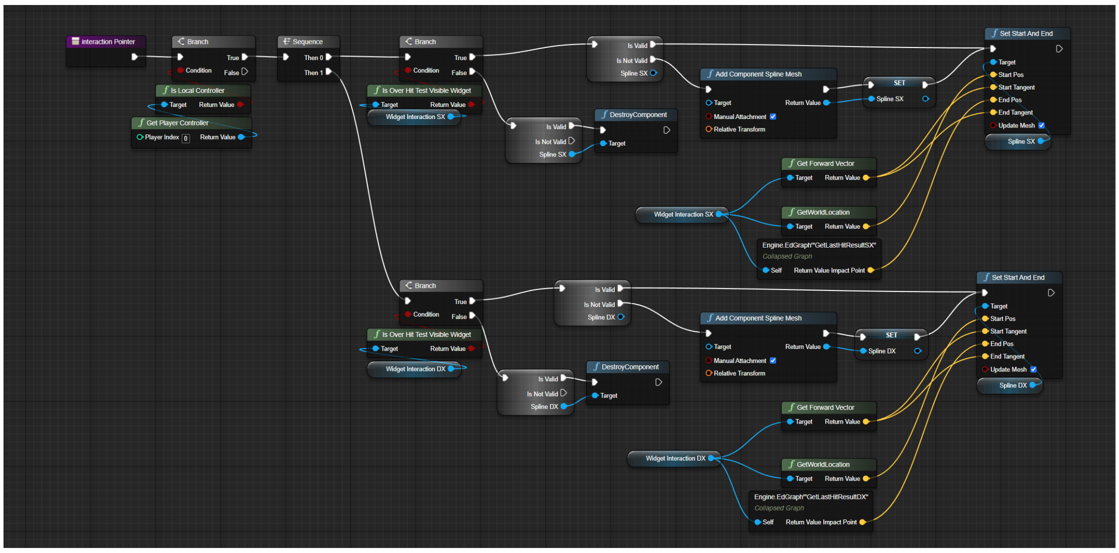


Figure 4.3: Widget Interaction code

the latency effect caused by the tick group of the `VRCharacter`, 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 impleming 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

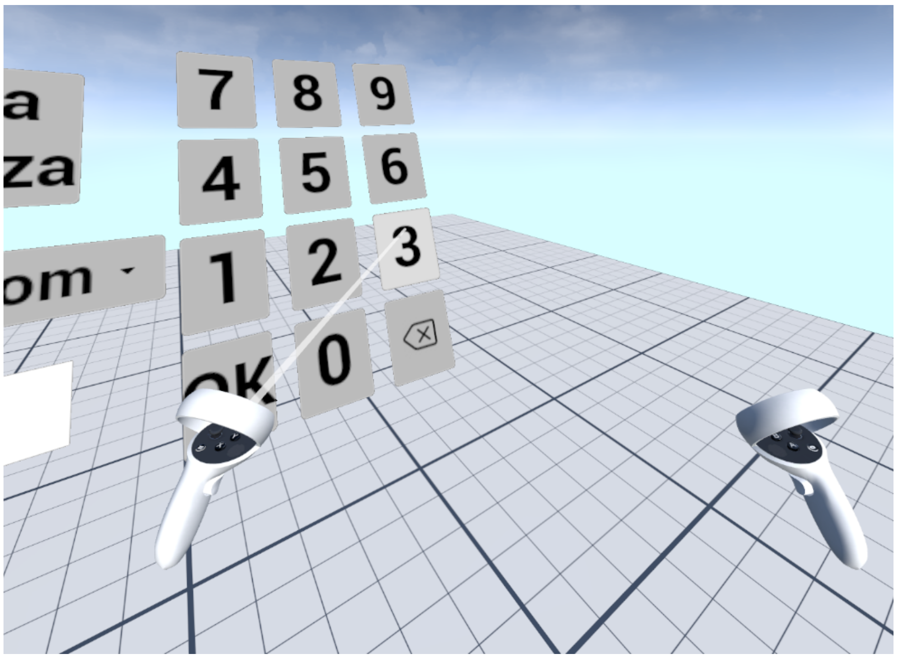


Figure 4.4: On the left a controller that create the spline pointing to the widget

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 relese the grip button.
The code for this action is siple, when it's decided that the actior is been resized there the distance between the two controllers will be calculated and the current scale vector of the actor will be saved, afther that the size will be decided by the following formula [4.1], for smoothness, it will be executed on every tick.

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

4.3 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 menù:** used for creating a session and joining it. It will have a number pad for inserting the code to enter a session.

- **Side menù:** 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.

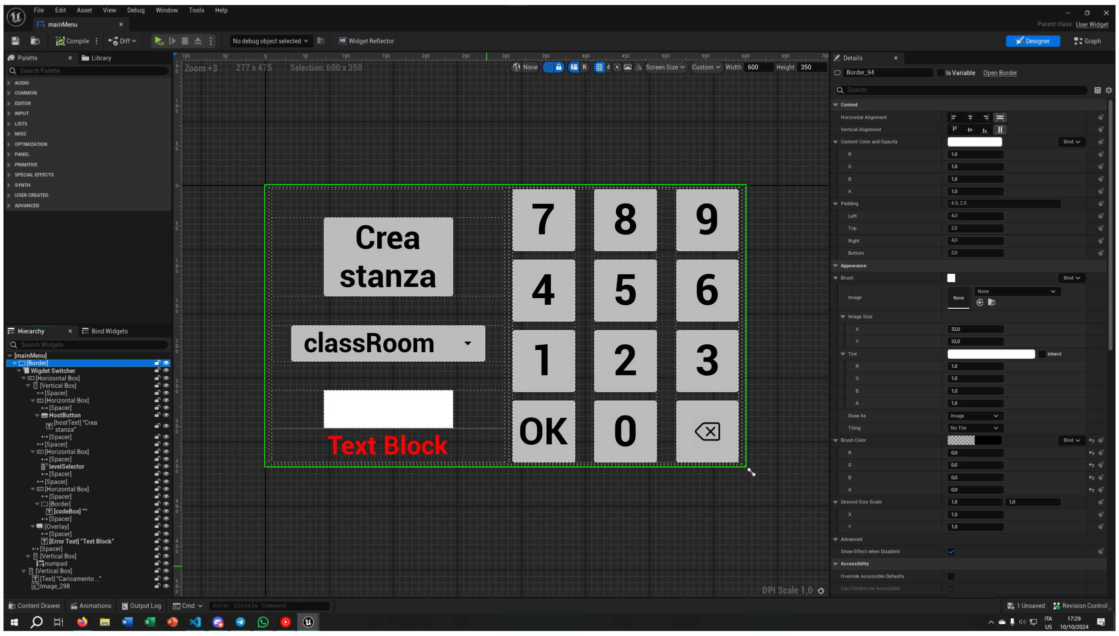


Figure 4.5: Main menu widget

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 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.

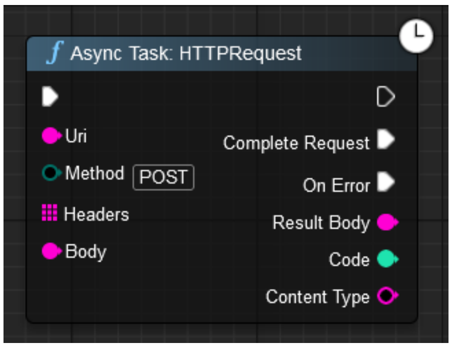


Figure 4.6: HTTP request node

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 [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 adress the client can connect to the host. If necessary, a new session can be opened. When a session is initiated,

URL	**Methods**	**Description**	**Response codes**
/create_session	POST	Creates a session if an IP addres is provvided in the JSON	200 400
/join_session	POST	Returns the IP addres of the host if the code in the JSON corrisponding to an open session	200 404
/update_session/<code>	GET	Updates timestamp for session	200 404
/list_files	GET	Rturns 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

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

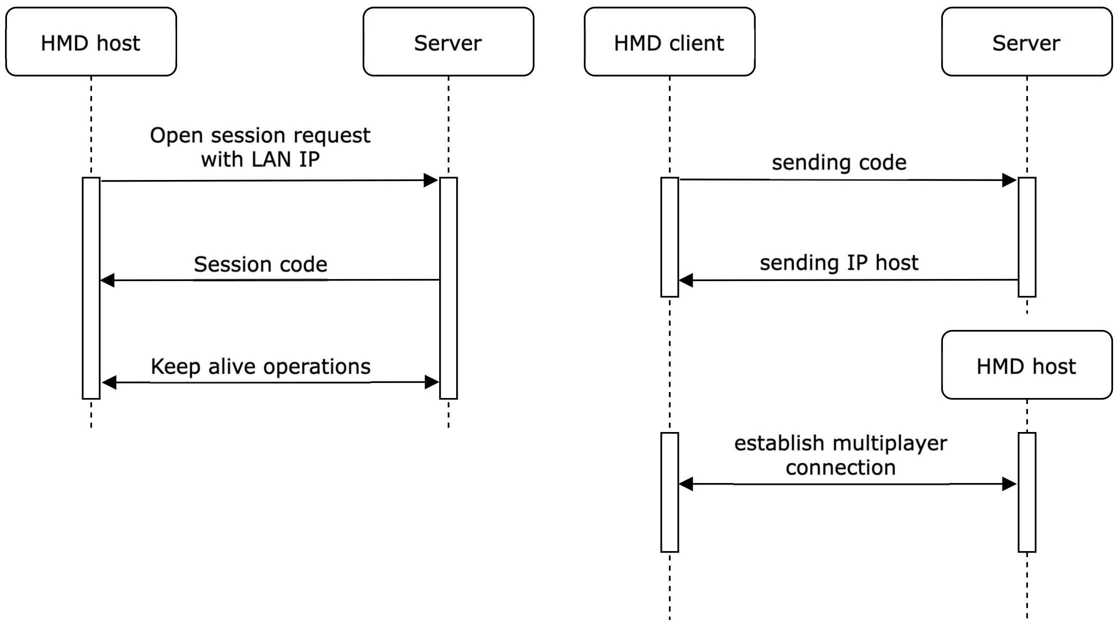


Figure 4.7: Network time diagram

5 Conclusions and Future Updates

A	**B**
C	D
E	F
G	H

Table 5.1: Table example

References

[1] 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.

[2] 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.

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

Acknowledgments