Development of a Mixed Reality Application for Emergency Training

Student Report
Final Year Internship
Program: Software Engineering and Computer Systems

Academic Supervisor:
Alexandre GUITTON

Internship Supervisor:
Hao YU

Internship Duration:
5 months

Defense Date:
26/08/2025

Sami ABDUL-SALAM
20/08/2025

Abstract

This internship is part of the development of a Virtual Reality (VR) solution using Unity, aimed at assessing the feasibility of the project and developing an application deployed on a VR headset to locate the nearest emergency exit based on the user's position within the UiT university building.

This goal, which proved difficult to achieve according to the research conducted, was redefined as the project progressed and depending on the available time. Carried out in a pair with Rémi BOUDRIE, a second-year student in the "Virtual Reality and Embedded Systems" program at ISIMA, this project allowed me, as a student from a different field, to learn Unity and strengthen this learning through paired development.

Built with Unity 6 and deployed in Android Package (APK) format on a Meta Quest 3 (entry-level) headset, the application meets the redefined scope objectives, though significant further development is needed to cover the entire building.

With limited time remaining, efforts were focused on improving documentation to assist potential successors in pursuing this ambitious project.

Keywords

Virtual Reality, Unity, Meta Quest 3, immersive application

Résumé

Ce stage s'inscrit dans le cadre de la programmation d'une solution en Réalité Virtuelle (RV) via Unity, visant à étudier la faisabilité du projet et à développer une application déployée sur un casque de RV pour trouver l'issue de secours la plus proche en fonction de la position de l'utilisateur dans le bâtiment de l'université de l'UiT.

Cet objectif, difficilement atteignable selon les recherches réalisées, a été redéfini au fil de l'avancement et du temps disponible. Réalisé en binôme avec Rémi BOUDRIE, étudiant en deuxième année de la filière « Réalité Virtuelle et Systèmes embarqués » à l'ISIMA, ce projet m'a permis, en tant qu'étudiant d'une autre filière, d'apprendre Unity et de renforcer cet apprentissage à travers le développement en binôme.

Réalisée avec Unity 6 et déployée au format APK sur un casque Meta Quest 3 (entrée de gamme), l'application répond aux objectifs fixés dans le périmètre redéfini, même si un important développement reste nécessaire pour couvrir l'ensemble du bâtiment.

Le temps restant étant limité, nous avons privilégié l'amélioration de la documentation afin de guider d'éventuels successeurs souhaitant poursuivre ce projet ambitieux.

Mots-clés

Réalité Virtuelle, Unity, Meta Quest 3, application immersive

Acknowledgement

I would first like to thank my school, the *Institut Supérieur d'Informatique, de Modélisation et de leurs Applications* (ISIMA), for supporting my academic journey in the field of computer science.

I am especially grateful to Loïc YON for trusting me with the topic of the internship and allowing me to further refine my professional path.

I would also like to thank Alexandre GUITTON, my academic supervisor, for his support and valuable advice throughout the internship.

I am also thankful to Hao YU, my internship supervisor, for trusting me, warmly welcoming me to UiT, and checking in on me regularly during the internship.

Finally, I would like to express my sincere gratitude to Marius WANG, who was a great technical supervisor, guiding and advising me on technical matters throughout the internship.

Table of Contents

Table of Figures

Introduction

According to its website [1], UiT The Arctic University of Norway is the northernmost university in the world and plays an important role in knowledge-based development at regional, national, and international levels. Its unique location on the edge of the Arctic significantly shapes its mission, focusing on critical issues like climate change, Arctic resource management, and various environmental challenges. The university offers a wide range of study programs and carries out well-known research in areas such as polar studies, marine ecosystems, indigenous peoples, and space physics. With a diverse academic community and many international partnerships, UiT is an important center for education and research in the High North.

The internship focused on developing a Virtual Reality (VR) solution using Unity, with the goal of researching the feasibility of creating an application that could help users locate the nearest emergency exit, based on their position within the UiT university building, for training purposes using a VR headset. More specifically, the project aimed to create an immersive application in Mixed Reality (MR), which involves integrating a virtual environment with the real world. It is important to note that VR and MR are distinct fields. Compared to developing a VR application, which only creates a virtual world, MR involves adding interactions between virtual elements and real-world objects that the headset interprets.

Regarding our goal, I first needed to become familiar with the Unity game engine, as I am a student specialized in another field, before starting the project development with my teammate, Rémi Boudrie, who is a second-year student at ISIMA, partially specialized in this field.

For this project, we used the Meta Quest 3 headset, known for its VR features and immersive experiences, and capable of running MR programs. In Unity, we needed to create a working environment compatible with MR, specifically for the Meta Quest 3 headset, and find the appropriate packages to achieve our goal. We also had to configure the Meta Quest 3 to run the application, mainly for developer purposes.

With these tools, we had to work together on the same computer, researching and developing the application at every stage of the project. Despite these circumstances affecting the distribution of the workload and the productivity rate in the project's development, this allowed us to adopt a collaborative approach, combining our skills, knowledge and ideas, which made the learning process more effective and enjoyable.

This report is structured to provide a comprehensive overview of the internship project. It begins by briefly presenting the university and the internship's mission and objectives. The tools used, including the Meta Quest 3 headset and Unity, are discussed in the next section. After that, the report covers the basics of getting started with Unity, presenting key elements to make the project easier to understand. It also includes some practice projects I completed to train myself in Unity at the start of the internship. Following this, the report explores the development process, highlighting the main challenges and solutions encountered. Finally, it concludes with reflections on the experience and suggestions for future work in this area.

1 | Context and Work Environment

1.1 UiT The Arctic University of Norway

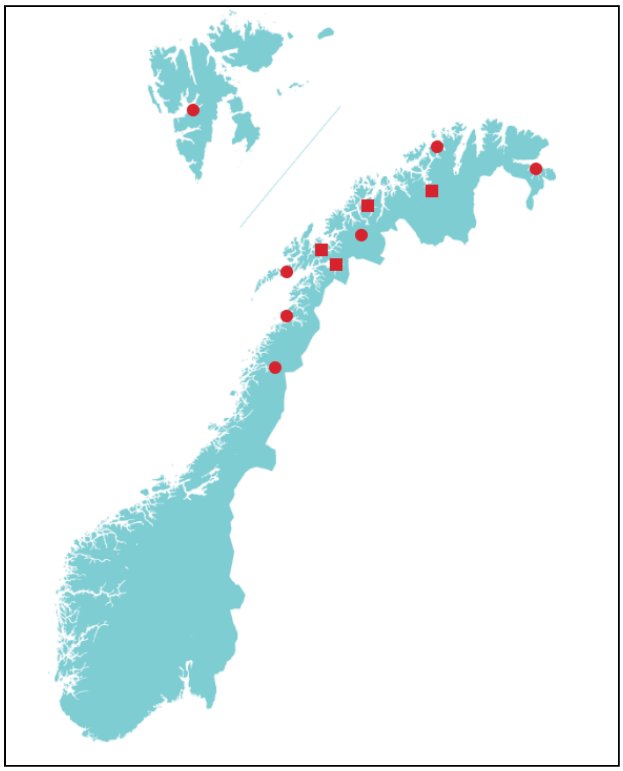

Figure 1: Map of UiT Campuses across Northern Norway

As we can see on the map above, UiT The Arctic University of Norway is present in 11 different locations across the country. Among these sites, we can mention Narvik, where our main campus is located, as well as Tromsø, Alta, and other towns in northern Norway. This geographical diversity enriches the academic and cultural experience for both students and staff.

As already mentioned in the introduction, UiT is a higher education and research institution that stands out for its commitment to innovation and international collaboration. It offers a wide range of academic programs and is well known for its significant contributions in various research fields.

Our academic supervisor, Hao Yu, gave my teammate and me a tour of the university. We noticed that it is modern and well equipped, with collaborative workspaces and advanced research facilities. The atmosphere is friendly and encourages learning. Narvik itself is a quiet town with beautiful landscapes, offering a pleasant living environment for students.

1.2 Internship Mission and Objectives

After our first meeting with Hao Yu, we met Marius Wang, a teacher at UiT Narvik who is more specialized in the field of virtual reality. He showed us their storage area as well as the rooms and equipment dedicated to research in this field. Among the different project proposals related to virtual reality, we were most interested in a research project to create an application for evacuation training at UiT Narvik in case of an emergency. The goal is to be able to find the nearest exit route.

The immersive aspect of this application, involving mixed reality, is even more interesting for us, as it is a field that is not very well known or explored by the general public. After brainstorming the subject with Marius Wang, we had to give a presentation to Hao Yu, Marius Wang, and other UiT members to present the project we were about to work on for this internship. This presentation was a good opportunity to discuss and exchange ideas and suggestions.

Before starting the project, we had to complete a template document in which we defined and described the project in detail. This included setting a clear scope and planning the project using a preliminary Gantt chart and milestones to set objectives over time.

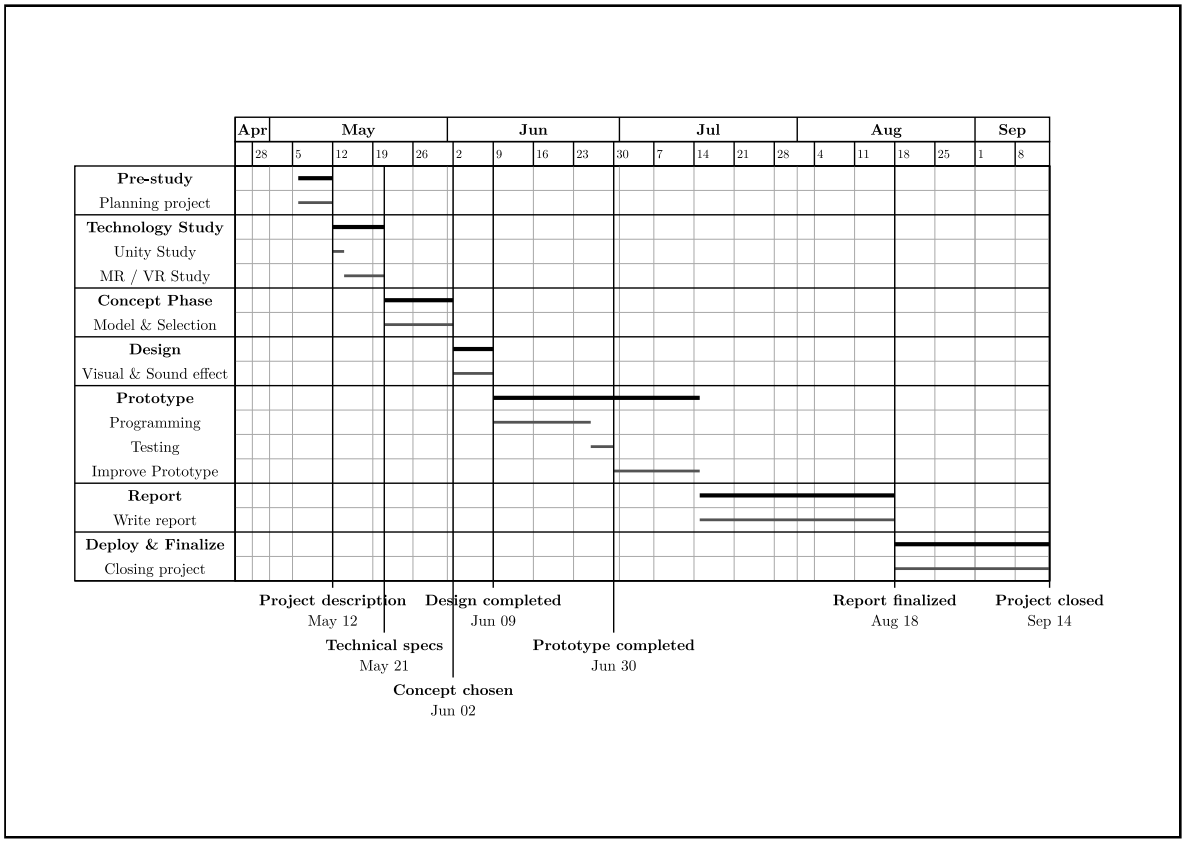

Figure 2: Preliminary Gantt Chart

This preliminary Gantt chart allowed us to plan the main steps of the project. We can see that we only used keywords and very general terms. Each step may be changed or adapted depending on the progress of the project and any unexpected events. It is therefore important to stay flexible and adapt to the needs of the project as it develops.

1.3 Tools Used

In this section, the tools assigned for this internship are presented. Their use cases, advantages and disadvantages, and relevance to the project are described. They are also compared with similar tools, and the reasons for selecting them over the alternatives are explained.

1.3.1 VR Headset: Meta Quest 3

Figure 3: Meta Quest 3 Headset and Controllers

During the Meta Connect 2023 event, the Meta Quest 3 was presented as Meta's most advanced virtual reality headset. More specifically, they mentioned its ability to be used for mixed reality, which is the case for our project. It is equipped with cameras that capture the real environment and integrate virtual elements into it. The headset also comes with wireless controllers, allowing more freedom of movement during use. More precisely, the headset permits the user to register their rooms in the headset to create mixed reality experience and interact safely with virtual objects within their real environment. This opens up many new possibilities for developers and users, whether for entertainment, gaming, fitness, virtual object design, simulation, and many other fields available in their library [2].

Compared to similar products from other brands, such as the Apple Vision Pro or HTC Vive XR Elite, the Meta Quest 3 stands out for its affordable price and decent performance, especially for an average user. However, it is still interesting to look at the differences between an entry-level headset like the Meta Quest 3 and a high-end product like the Apple Vision Pro.

Here is a comparison table highlighting the main differences between the two headsets. The aim of this comparison is to give a quick overview of their technical characteristics, making it easier to distinguish between an entry-level and a high-end headset without going too much into detail. The specifications are based on an article from TS2 Space [3].

	Meta Quest 3	Apple Vision Pro
Price	$499	$3499
Display	LCD 2064 x 2208 pixels per eye at 120Hz	OLED 2160 x 3840 pixels per eye at 100HZ
Design	Plastic body	Aluminum and glass body
Performance	Optimized for running one immersive app at a time; may lag with very complex scenes.	Handles multiple complex apps smoothly; OS feels fast and responsive.
Mixed Reality	Color passthrough via dual 4MP cameras, decent quality	High-fidelity color passthrough by default, very sharp and clear with excellent color accuracy
Eye Tracking	No	Yes

Table 1: Quick comparison of Meta Quest 3 and Apple Vision Pro

In fact, this comparison shows that the price difference between the two headsets is justified by the quality of materials, display resolution, and advanced features like eye tracking or multitasking. The Meta Quest 3 is therefore better suited for general and occasional use, while the Apple Vision Pro is for users who want a top-quality mixed reality experience and are ready to spend more..

For our project, it makes more sense to use a Meta Quest 3, as it is more affordable and sufficient for developing a mixed reality application.

1.3.2 Unity

In this section, we briefly explain why we use Unity instead of another option, the use cases for Unity, and its role in the video game and virtual reality industry.

In the industry, Unity is one of the most used game engines for making video games and virtual reality applications. It is popular because it is flexible, easy to use, and has a large community of developers. Unity allows creating interactive experiences in 2D and 3D, and it works with many devices, including virtual reality headsets like the Meta Quest 3.

There is also another popular game engine, Unreal Engine, which is often used for more complex projects with advanced graphics. However, Unity is usually preferred for virtual reality projects because it is easier to use and has many available resources.

It is important to note that the basics presented in this report are brief and intended to help understand Unity's vocabulary and key concepts. They include only what is necessary to follow my contribution to the project development. Each element, component, or feature in Unity could be studied in much more depth, as they have many options and details. However, for the purpose of this report, it is sufficient to focus on the essential ideas and terminology. This step is also crucial for the internship, as I need to learn how to use Unity, which is a new tool for me, before being able to contribute to the project. The practical work I carried out, although it does not directly contribute to the project, has helped me become familiar with the tool, which is important for the internship.

2 | Getting Started with Unity

2.1 Unity Basics

In this section, we will cover the basics of Unity, including its interface and the main concepts of development in Unity. This will help us understand how to use Unity effectively for our internship project. The different parts will be described with the help of the official Unity manual [4].

2.1.1 GameObjects & Components

First, it is important to understand that Unity is based on GameObjects and Components, all placed inside a scene. A scene is a workspace in Unity where we put our virtual elements, the GameObjects, which make the virtual environment richer.

A GameObject is an element in the scene, such as a character, tree, or building, in 3D or 2D. According to Unity's documentation, it can represent anything in the scene. Each GameObject can contain several Components, which are scripts or features that define its behavior. These Components act like properties or actions, enabling interactions and animations within the virtual environment.

It is important to remember that a GameObject is an empty container that only becomes meaningful when Components are added. Without them, it does nothing in the scene.

Here is an example of a GameObject with Components:

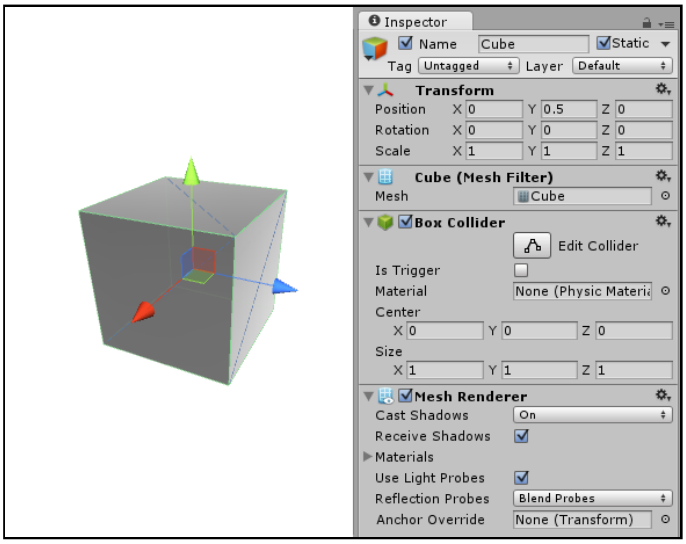

Figure 4: Example of a GameObject with Components in Unity

In this example, we have a GameObject called "Cube" with four Components: Transform, Mesh Filter, Mesh Renderer, and Box Collider. The Transform Component sets the position, rotation, and scale of the GameObject. The Mesh Filter specifies the 3D model, and the Mesh Renderer makes it visible. The Box Collider is used for collision detection with other GameObjects. Note that the figure is from an old version of Unity, the interface of our Unity version will be presented later in the report.

A GameObject in Unity has, by default, a Transform Component, which is essential because it defines the GameObject's position, rotation, and scale in the scene. Without it, the GameObject would not exist in the game world. The Transform Component also allows Parenting, a way to link GameObjects together in a hierarchy.

Indeed, a GameObject can be the parent of one or more other GameObjects. This is useful for organizing the scene and creating complex structures. For example, a GameObject that represents a car can have child GameObjects for its wheels, lights, and body. When the parent GameObject moves, all its children move with it, maintaining their relative positions. Parenting is a powerful feature in Unity that helps manage complex scenes and animations. This involves an important concept that we will cover in detail during the project development: world and local positions.

In Unity, it is also possible to create what are called Prefabs, which are preconfigured GameObjects that can be reused in multiple scenes. They are saved as files with the extension ".prefab". Prefabs help save time and keep the project consistent. For example, if you create a character with specific animations and behaviors, you can turn it into a Prefab and use it in different scenes without recreating it each time.

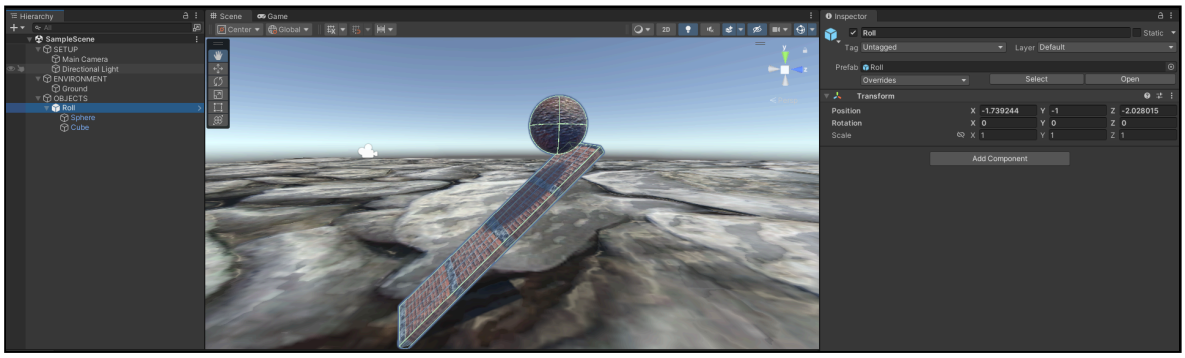

Figure 5: Example of a Prefab in Unity

In this example, we have a Prefab called "Roll". On the left, we can find in the hierarchy section blue GameObjects that correspond to the prefab used in the scene. This prefab has, as children, a GameObject called "Sphere" and another one called "Cube"; they have their own Components (Sphere has physics Components like Rigidbody and Sphere Collider so it can fall and roll over the plank, which is the Cube GameObject that has a Box Collider Component to detect collisions). On the right, we can see the Inspector section that shows the Components of the selected GameObject, in this case, it has only the Transform Component.

If we look at the Cube example, we can also see that GameObjects can be identified using Tags. Tags are labels that can be assigned to GameObjects to organize them and find them easily in the project. For example, you can tag a GameObject as "Player" or "Enemy" to tell it apart from other objects in the game. This helps manage GameObjects and locate them quickly.

We can also notice the Layers, which are rendering layers used to organize GameObjects in the scene. Layers control the visibility and interaction between GameObjects. For example, you can create one Layer for interactive objects and another for non-interactive objects, making it easier to manage collisions and interactions in the game.

2.1.2 Materials & Shaders

Materials and Shaders control the appearance of GameObjects in a Unity scene. They define how objects look, including their color, texture, and other visual properties.

A Material contains the information needed to render a GameObject, such as its color settings, textures, and the Shader it uses. A texture is an image applied to a Material to give it a specific look — for example, a wood texture for a wooden object or a metal texture for a metallic one.

The Shader is a program that runs on the GPU and decides how the Material is displayed based on factors like light, camera position, and other elements in the scene. Shaders can be simple, only applying colors, or more advanced, creating effects such as reflections, refractions, or transparency.

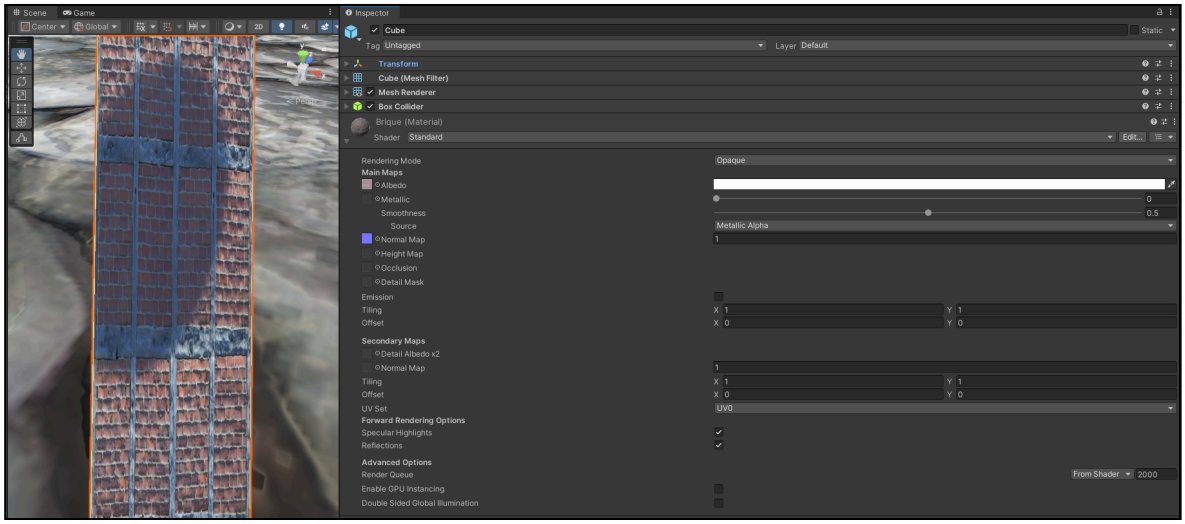

Figure 6: Example of a Material in Unity

In this example, without going too deep into details, we can see a Material component called "Brique". First, we can see that the Material is linked to a Shader, here "Standard", which is Unity's default Shader. Then, the Rendering Mode, which corresponds to how the Material is rendered in the scene, is set to "Opaque", meaning it does not allow light to pass through. This is important for how the Material interacts with light and shadows in the scene. We can also see that the Material has a base color (Albedo), a texture (in this case, a brick image), and other properties like metallicity and smoothness that affect how the Material looks in the scene. We can also notice that the texture has a Normal Map, which is an image used to simulate surface details by creating the illusion of bumps and textures on the Material's surface. We will not go further into details, because there are many possible settings and properties for Materials and Shaders, but the main idea is to understand that they are used to define the appearance of GameObjects in the scene.

Now we understand that Materials and Shaders are essential for creating realistic and attractive visuals in Unity. They allow you to customize the appearance of objects according to the needs of the project, using textures, colors, and special effects, all based on the settings and placement of lights and the camera used.

2.1.3 Editor Interface

Here is a look at the Unity Editor interface, which is the main workspace where you will create and manage your projects. The interface is divided into several sections, each serving a specific purpose:

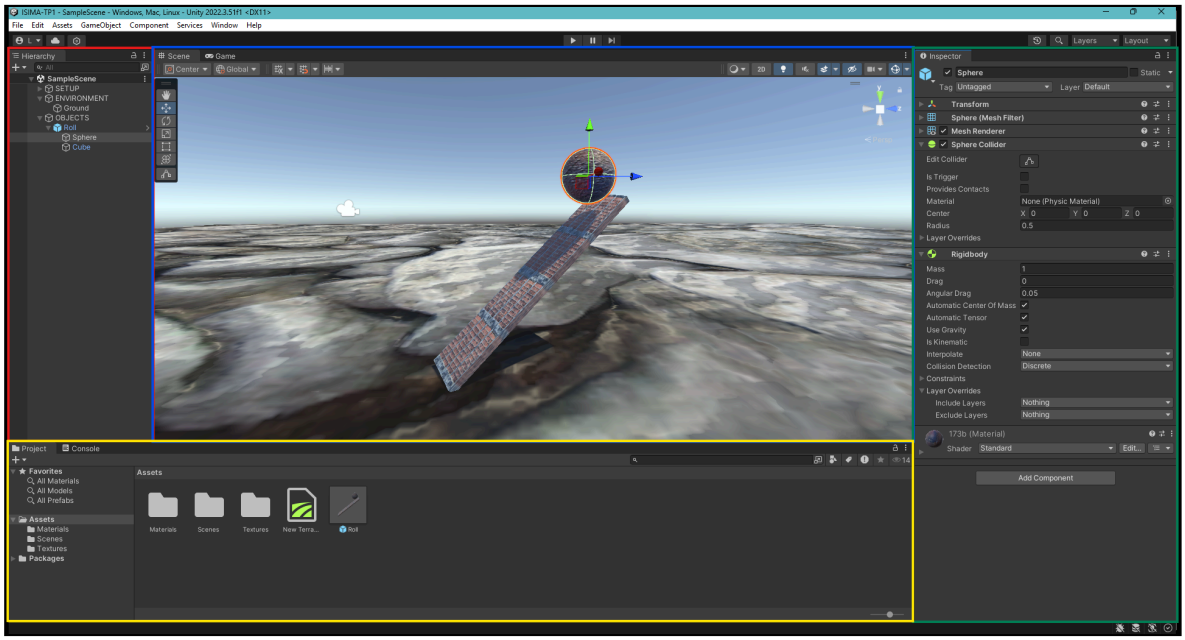

Figure 7: Unity Editor Interface

To start, we have the Scene View (blue rectangle), which is the main area where you can see and edit your GameObjects in the scene. You can navigate in this view to position and manipulate GameObjects. You can switch to the Game View to see how the scene will look when played, once you click on the play button above the Scene View, which is useful for testing and debugging. Being able to select, manipulate, and modify objects in the Scene View are some of the first skills you must learn to begin working in Unity.

The Hierarchy window (red rectangle) shows all the GameObjects in the current scene, allowing you to organize them and manage their relationships. Some of these can be instantiated directly in the Editor (such as 3D models), while others may be Prefabs, as we have seen before. It is here where you organize parent-child relationships between GameObjects, which is essential for managing complex scenes. It is also used to toggle the visibility of GameObjects in the Scene View, making it easier to work with large scenes. A Unity project can contain multiple scenes, and you can switch between them using the Scenes tab (green rectangle) at the top of the interface. This is useful for organizing different parts of your project, such as levels or menus, but this is not the case in our project.

The Inspector window (green rectangle) displays the properties of the selected GameObject, including its Components, Materials, and other settings. You can modify these properties directly in this section, and also add new Components to the GameObject.

The Project window (yellow rectangle) shows all the Assets in your project, including scripts, textures, models, and other files (Materials, Prefabs, etc.).

You can organize your Assets into folders to keep your project tidy. This is where you can import new Assets and create Prefabs. It also contains a Console tab, which displays messages, warnings, and errors from your scripts and the Unity engine. This tab is essential for debugging and understanding what is happening in your project.

2.1.4 C# Scripting

C# is the primary programming language used in Unity for scripting. It allows you to create custom behaviors and interactions for your GameObjects. Unity uses a component-based architecture, meaning that scripts are attached to GameObjects as Components, enabling you to define their behavior. Scripts can be used to create graphical effects, control the physical behaviour of objects or even implement a custom AI system for characters in the game. These scripts are attached to GameObjects, whose interactions and behaviors are defined by MonoBehaviour components. This means that the classes from these scripts inherit from the MonoBehaviour class, which is the base class for all Unity scripts. This allows access to specific methods and properties, such as methods that run at startup, update every frame, or handle collisions with another GameObject. In short, there are many existing methods in MonoBehaviour that our class can use by inheriting from it, allowing us to define the behavior of our GameObject through the different methods we edit inside the class in the C# script assigned to it.

```C#
using UnityEngine;
using System.Collections;

public class NewMonoBehaviourScript : MonoBehaviour {

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    void Start()
    {
        // Initialization code
    }

    // Update is called once per frame
    void Update()
    {
        // Code to execute every frame
    }
}
```

Listing 1: Default MonoBehaviour C# script in Unity

The main things to note are the two methods inside the class. The `Update` method is where you put code that runs every frame for the GameObject, such as movement, triggering actions, or responding to player input. It is used for anything that needs to be updated over time during gameplay. Before this, the `Start` method is called once by Unity before gameplay begins, making it the perfect place to set variables, read preferences, and connect with other GameObjects for initialization.

An interesting point to note is that, even though we are not going into the details of Object-Oriented Programming here, it is important to know how to manage the resources used when creating an object (that is, an instance of a class) and how to destroy an instance without causing memory leaks. In Unity, however, we do not use any specific method for creating or destroying an object. The Unity Editor handles this automatically, and defining a constructor in a class that inherits from MonoBehaviour can cause problems.

One important and widely used concept in Unity is events. Events are actions that happen in the game and can be listened to by other scripts. By "listened to," we mean that you can define methods that will be called when a specific event occurs. For example, an event can be triggered when a player picks up an item, when an enemy is defeated, or when a door opens. Events make it possible to create interactions between different GameObjects and react to specific actions in the game. They are often used to manage interactions between the player and the environment, such as collecting items or triggering animations.

Collisions are a specific type of event that happen when GameObjects interact physically. Unity provides methods like `OnCollisionEnter`, `OnCollisionStay`, and `OnCollisionExit`, which are called when two GameObjects with colliders first collide, stay in contact, or separate. These methods are useful for handling physical interactions like bounces, impacts, or force effects. There are also the `OnTriggerEnter`, `OnTriggerStay`, and `OnTriggerExit` methods, which work similarly but are used for colliders marked as triggers. The difference is that triggers do not create physical collisions but allow you to detect interactions without applying physical forces.

```C#
using UnityEngine;
using UnityEngine.Events;

public class EventExample : MonoBehaviour {
    public UnityEvent pDeath;
    public bool triggerEvent;

    void Update() {
        if (triggerEvent) {
            // Simulate player death
            PlayerDied();
        }
    }

    void PlayerDied() {
        // Trigger the event when the player dies
        pDeath.Invoke();
    }
}
```

```C#
using UnityEngine;

public class EventListener : MonoBehaviour {
    public EventExample ev;

    void Start() {
        // Subscribe to the event
        ev.pDeath.AddListener(Dead);
    }

    void Dead() {
        Debug.Log("Player has died!");
    }
}
```

Listing 2: Example of an UnityEvent script with listener script for the event.

In this example, we have two scripts: one that triggers an event when the player dies and another that listens to that event. The first script uses a UnityEvent type to define the event, with the method `PlayerDied` that invokes the event when the player dies. More specifically, the `pDeath` event is triggered when the `triggerEvent` boolean is set to true in the `Update` method, `pDeath.Invoke()` will call all methods that are subscribed to this event. This is handy because it allows us to regroup different methods from various scripts that need to be called when a specific event is triggered, providing a clean and organized way to manage interactions in the game.

The second script listens to the event by the method `AddListener` (line 8) on a UnityEvent type with the method that will be called when the event is triggered. We can see that the `Dead` method, which is added to the UnityEvent `pDeath`, is called when the player dies, and it simply logs a message to the console. This is a simple example of how events can be used to create interactions between GameObjects in Unity.

2.2 Lab Work

In this section, we will apply the concepts learned in the previous section to work on simple and small Unity projects. All the work was done at the beginning of the internship, and it is important to note that it is not related to the main project we will be working on.

Firstly, on my first project, as we already saw in Figures 5, 6, and 7, we can use the Unity Editor for creating and managing GameObjects, Components, Materials, and Shaders. There is a simple scene with a few GameObjects, such as a cube, a sphere, and another cube that represents a plane by its shape. Especially in Figure 7, we can see in the Inspector that the sphere is selected, and we can see its Components, such as the Transform, Mesh Filter, Mesh Renderer, Sphere Collider, and a Rigidbody. The Rigidbody Component is used to apply physics to the GameObject, such as gravity, and the Sphere Collider is used for collision detection with other GameObjects. With both, our sphere will roll over the plane if it also has a Box Collider Component, which is the case we can see in Figure 6.

Then, the second project consists in making a 3D model rotating, and we had to add UI elements to the scene, such as a button that allows us to controll some rotation settings that we have to implement.

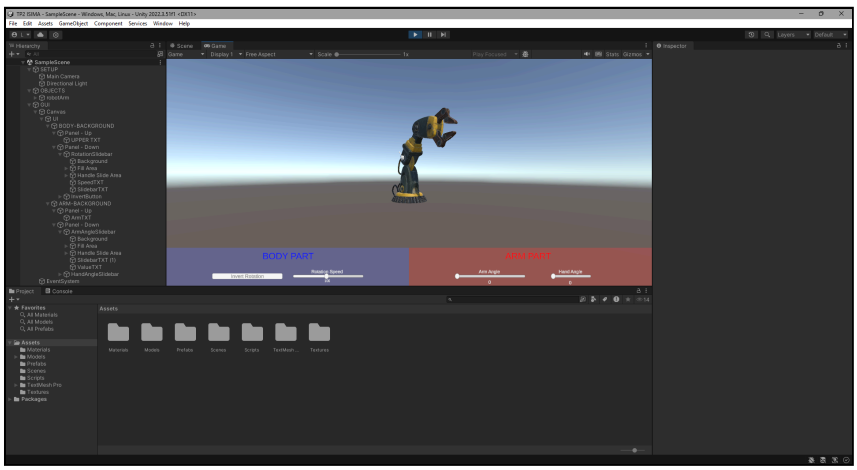

Figure 8: Second Lab – Rotating 3D model in Unity with UI elements

First, we will focus on rotating the 3D model. To do this, we will create a C# script that manages the model's rotation based on the parameters set by the user. We will use the `Update` method to apply continuous rotation to the model, and we will add controls to adjust the rotation.

```C#
public class RotationAuto : MonoBehaviour
{
    public const float MAX_SPEED = 200f; // Maximum speed limit
    public const float MIN_SPEED = 10f; // Minimum speed limit

    [Tooltip("Speed of rotation in degrees per second.")]
    [Range(MIN_SPEED, MAX_SPEED)]
    public float _speedRotateY = 100f; // Speed of rotation

    [Tooltip("Invert rotation direction.")]
    public bool invertRotation = false; // Invert rotation direction

    void Start() {}

    void Update()
    {
        float speed = invertRotation ? -_speedRotateY : _speedRotateY;
        transform.Rotate(0, speed * Time.deltaTime, 0);
    }

    public void InvertRotation() { invertRotation = !invertRotation; }

    public void SetSpeed(float speed)
    {
        _speedRotateY = (speed >= MIN_SPEED) && (speed <= MAX_SPEED) ?
        speed : _speedRotateY; // Set speed if within range
    }
}
```

Listing 3: C# script - Rotating a 3D model in Unity with parameters

In this script, we define a public attribute `_speedRotateY` that allows us to set the rotation speed in degrees per second. To make an element rotate on itself, we need to apply a rotation on the Y axis, the vertical axis. We can note the use of `[Tooltip]` and `[Range]` to add information and a range of values for the `_speedRotateY` attribute in the Unity Editor, allowing the user to easily adjust the rotation speed. Then we have the `invertRotation` attribute, which is a boolean used to reverse the rotation direction. All attributes are public, which allows them to be modified directly in the Unity Inspector. The other way would be to use `[SerializedField]` to keep the attributes private while still being able to edit them in the Inspector, thus respecting the concept of Encapsulation.

We also have the `MAX_SPEED` and `MIN_SPEED` attributes to define the rotation speed limits, ensuring it stays within a reasonable range.

Nothing is done in the `Start` method, as we only need to apply the rotation in the `Update` method.

In the `Update` method, we apply the rotation to the GameObject using `transform.Rotate`, which takes three parameters: the rotation on the X, Y, and Z axes. We use `Time.deltaTime` to ensure that the rotation speed is consistent across different frame rates. The `invertRotation` boolean is used to determine whether to apply a positive or negative rotation speed. More precisely, `transform` corresponds to the Transform Component of the GameObject to which this script is attached, allowing us to manipulate its position, rotation, and scale. But there is also a very important method, `GetComponent<>`, which allows us to access the Components of the GameObject to manipulate them directly in the script. This is useful for editing properties of the GameObject. It also permits navigation through the hierarchy of GameObjects and access to their Components, which is essential for creating complex behaviors in Unity. For example, in this case, we could use `GetComponent<Transform>().Rotate` to apply the rotation directly to the Transform Component of the GameObject, but in our case, we can just directly use `transform`.

Then, we have two public methods: `InvertRotation` and `SetSpeed`. The first method toggles the `invertRotation` boolean, allowing the user to change the rotation direction. The second method sets the `_speedRotateY` attribute to a new value, ensuring it is within the defined limits. This allows for dynamic control of the rotation speed from the Unity Editor or through other scripts.

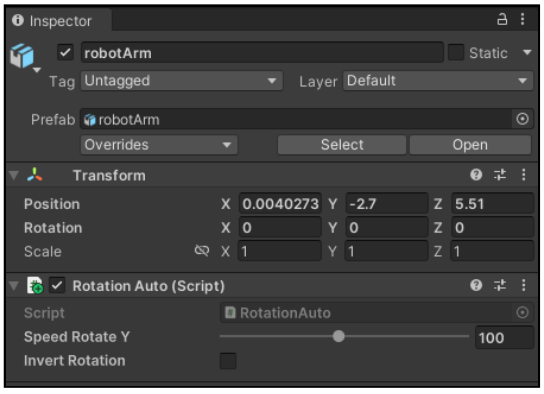

Figure 9: RotationAuto Component attached to a GameObject

So, we have a first script that makes our 3D model rotate based on the speed set by the user and whether the rotation is inverted. In this case, we attach our script to the `robotArm` GameObject, giving it a `RotationAuto` Component, which is the name of our script.

For the next part of this project, we want to be able to raise and lower the robot's arm. The `robotArm` GameObject is a prefab with child GameObjects for each part of the arm, which allows us to manipulate each part individually. To do this, we will create a C# script (`ControlRotation`) that manages the model's rotation based on the parameters set by the user. Actually, the script will be similar to the previous one, but the `Update` method will be used to apply new angle values to the GameObject's `Transform` Component, allowing us to raise and lower the arm.

```C#
void Update()
{
    GetComponent<Transform>().localRotation =
    Quaternion.AngleAxis(_currentAngle, new Vector3(0, 0, 1));
}

public void SetAngle(float angle)
{
    if (angle >= MIN_ANGLE && angle <= MAX_ANGLE)
    {
        _currentAngle = angle;
    }
}
```

Listing 4: Update and SetAngle methods for editing Z angle GameObject in Unity

With this script, just like with `RotationAuto`, we can attach it as a Component to both the arm part and the hand part.

Now, we need to give the user a way to control the element's rotation. For this, we will create a user interface (UI) with buttons to adjust the rotation. Unity has a built-in UI system that makes it easy to create interactive elements.

Without going into UI details, we will only show the elements needed to control the rotation. In this project, we first have a button to change the rotation direction. For that, we have a GameObject called `InvertButton` which has a `Button` Component. This Component can call methods when we click the button using its `OnClick()` method.

In the Inspector, we can add the method `InvertRotation` from our `RotationAuto` script, which is placed on the `robotArm` GameObject. This method changes the rotation direction of the robotic arm. We also specify the source of the script, because this works with references. If we had several GameObjects with the same script, we would not know which one to choose.

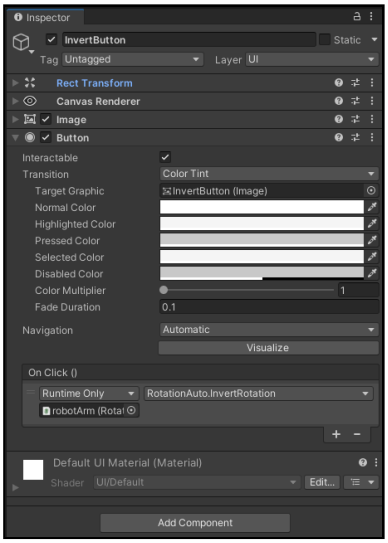

Figure 10: InvertButton Component with InvertRotation added to OnClick

For rotation and angle, we create what is called a `Slider`, which is a UI element that lets the user select a value from a range by moving a handle. You can find it in the UI section of Unity. The concept is similar to a button: we add the `RotationAuto` script and the `SetSpeed` method in the `OnValueChanged` section, just like we add a method to a `Button` using `OnClick`. We do the same for the `ControlRotation` script, but with the `SetAngle` method for another `Slider`. This is very similar to the previous case, so we will not show extra figures for it.

For the last project, we want to handle object detection using a raycast, which is a virtual line cast in 3D space to detect objects.

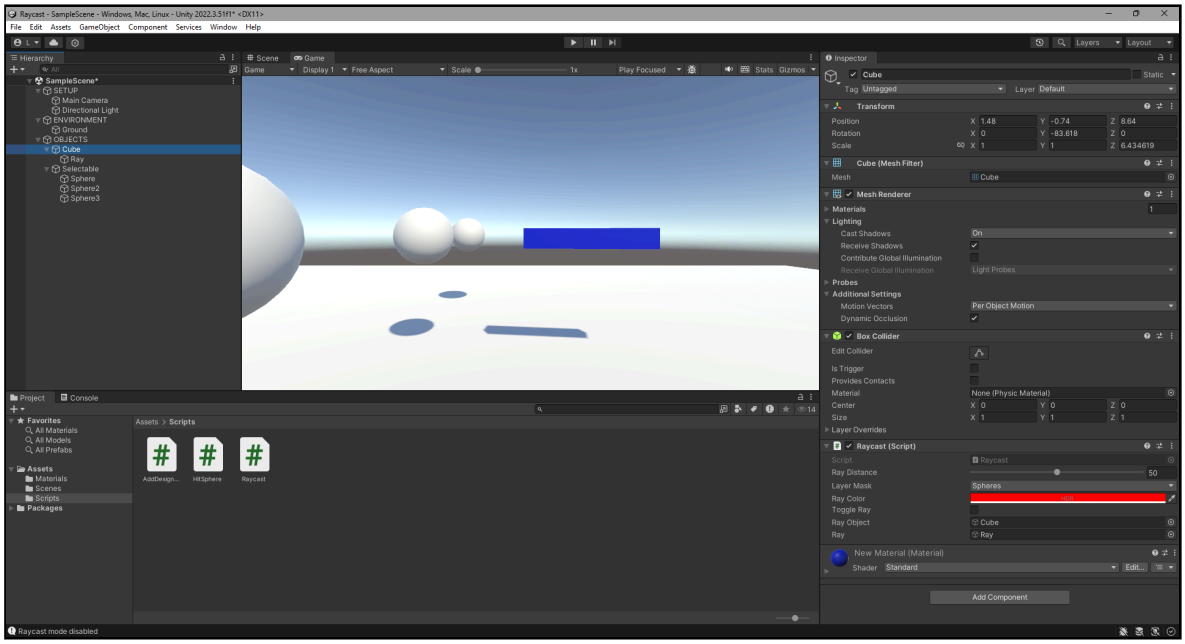

Figure 11: Third Lab – Raycasting in Unity

The first step is to create a script that handles the raycast. Specifically, we have a `Cube` GameObject that has a child GameObject `Ray` with a `Line Renderer` component. This component allows us to draw a straight ray and detect objects hit by this ray, so we can interact with the objects that are hit.

The `Raycast` script is long, so we will break it into several parts. First, we summarize the attributes: we start with `rayDistance`, which is the maximum distance of the ray, `layerMask`, which filters the objects detected by the ray so it only affects objects on this layer and ignores the others, and `toggleRay`, which is a boolean that can turn the raycast on or off. The `layerMask` is related to what we already saw about Layers on any GameObject in the Unity Basics section. We also have an attribute `lastTarget` to remember the last object hit by the ray, so we can reset it to its original state when the ray no longer hits it. The idea is the following: when we hit an object with the ray, we can get a reference to the GameObject and then call its method to change its state.

```C#
void Start()
{
    transform.GetComponentInChildren<LineRenderer>().positionCount = 2;
    transform.GetComponentInChildren<LineRenderer>().SetPosition(0, transform.position);
    transform.GetComponentInChildren<LineRenderer>().enabled = false;
}
```

```C#
 void Update()
 {
     if (Input.GetMouseButtonDown(0)) {switchMode();}
     if (toggleRay)
     {
         RaycastHit hit;
         Vector3 origin = transform.position;
         Vector3 direction = transform.forward;
         if (Physics.Raycast(origin, direction, out hit, rayDistance, layerMask))
         {
             transform.GetComponentInChildren<LineRenderer>().SetPosition(1, hit.point);
             GameObject currentTarget = hit.collider.gameObject;

             if(currentTarget != lastTarget)
             {
                 if (lastTarget != null)
                 {
                     lastTarget.GetComponent<HitSphere>().resetColor();
                 }
                 currentTarget.GetComponent<HitSphere>().hitByRaycast();
                 lastTarget = currentTarget;
             }
         }
         else
         {
             transform.GetComponentInChildren<LineRenderer>().SetPosition(1, origin + direction * rayDistance);
             if (lastTarget != null)
             {
                 lastTarget.GetComponent<HitSphere>().resetColor();
                 lastTarget = null;
             }
         }
         transform.GetComponentInChildren<LineRenderer>().enabled = true;
     }
 }
```

Listing 5: Start and Update methods of the Raycast script

In the `Start` method, we initialize the `LineRenderer` Component, which is used to draw the ray in the scene. We set the number of positions to 2 (the start and end points of the ray) and set the first position to the position of the GameObject. This is done by finding the `LineRenderer` Component in the child GameObject and setting its properties. We also disable the `LineRenderer` at the start, as we only want to show it when the raycast is active.

In the `Update` method, we check if the left mouse button is pressed to toggle the raycast mode. If it is, we set `toggleRay` to true or false, which will enable or disable the raycast. To avoid making the report too long, we will not go into too much detail about the `switchMode` method. It allows changing the state of the raycast, turning it on or off, and changing the color of the GameObject to show its state. If the raycast is on, the `Cube` GameObject changes color and we can detect objects hit by the ray. If the raycast is off, the `LineRenderer` is turned off, the GameObject color is reset, and any object that was hit by the ray is also reset. If the raycast is enabled, we create a `RaycastHit` variable to store information about the object hit by the ray. We then define the origin and direction of the ray based on the position and forward direction of the GameObject. We use the `Physics.Raycast` method to cast the ray and check if it hits an object within the specified distance and layer mask. If it hits an object, we set the end position of the ray to the hit point and get the GameObject that was hit. If the current target is different from the last target, we reset the color of the last target by calling his `resetColor` method and call the `hitByRaycast` method on the current target, which changes its color to indicate it was hit by the ray. We also update `lastTarget` to keep track of the current target.

Here is the `HitSphere` script, which is attached to the GameObjects that can be hit by the ray. We must make sure that the GameObjects have the layer matching the raycast's `layerMask`, otherwise they will not be detected.

```C#
public class HitSphere : MonoBehaviour
{
    void Start()
    {
        transform.GetComponent<Renderer>().material.color = Color.green;
    }
    void Update() {}

    public void hitByRaycast()
    {
        transform.GetComponent<Renderer>().material.color = Color.red;
    }

    public void resetColor()
    {
        transform.GetComponent<Renderer>().material.color = Color.green;
    }
}
```

Listing 6: HitSphere script for selectable objects hit by the raycast

3 | Project Development

3.1 Initial Research

To begin with, it is important to note that everything presented so far concerns the development of a virtual scene; in other words, we have not yet entered the field of mixed reality. To move forward, my teammate and I researched the different possibilities offered by Unity and the available packages to make our virtual scene compatible with mixed reality. As a side note, lighting setups were not covered in the Unity basics, as in mixed reality there is no need to configure them because the goal is to be able to see the real world through the headset. The Unity setup was understanded with the help of tutorials made by Ludic Worlds, which is a YouTube channel from a professional VR developer, with over 20 years experience in games and interactive media [5]. We then looked into how to ensure compatibility with the Meta Quest 3 headset. More specifically, we had to decide whether to make the application compatible with multiple headsets or only with Meta devices. Each option requires different packages, which in turn affects the way the application is developed.

3.1.1 Meta OpenXR

Concerning the compability of the application, we have chosen to have cross-platform compatibility, which means that the application will be able to run on different headsets. To achieve this, we have chosen to use OpenXR. OpenXR is a standard for Extended Reality (XR) applications that allows developers to create applications that can run on multiple devices without needing to rewrite the code for each device. However, to access some features specific to the Meta devices, we need to install the Meta OpenXR package, which is an extension of the OpenXR package.

Here are the other additional required packages that are dependencies of the Meta OpenXR package:

- OpenXR Plugin – Provides OpenXR support and extensions required to build mixed reality applications.
- XR Core Utilities – Offers a set of helper scripts and utilities commonly used in XR development.
- XR Legacy Input Helpers – Provides a user-friendly interface for configuring XR input and settings. It uses a plugin architecture to support a wide variety of XR platforms, such as Meta, Vive, or Windows Mixed Reality.
- XR Plugin Management – Allows developers to manage XR plugins in Unity, enabling or disabling them and configuring their settings.
- AR Foundation – Will be presented later.

Another package that needs to be installed is the XR Interaction Toolkit. This package is not a dependency of the Meta OpenXR package, but it is essential for developing mixed reality applications. It allows setting up the XR Origin (Rig), which is the interface between the VR hardware (headset and controllers) and the VR scene. It contains virtual counterparts of your VR hardware devices and tracks their position, orientation, and input actions.

3.1.2 AR Foundation

AR Foundation is a Unity tool that lets you create Augmented Reality (AR) or Mixed Reality (MR) applications that work on different devices and platforms. In your Unity project, you can choose which AR features to use by adding special components called managers to your scene. When you run the app on an AR device, AR Foundation uses the device's own AR system to make those features work. This means you can build your app once and use it on many popular AR platforms without rewriting it. The details of the different AR Foundation components can be found in the official Unity documentation for AR Foundation [6].

In Unity, to have the Passthrough feature, which permits to see the real world through the headset, we need to add the AR Camera Manager and AR Session components to our scene. The AR Camera Manager is responsible for managing the camera used for AR, while the AR Session component manages the lifecycle of the AR experience.

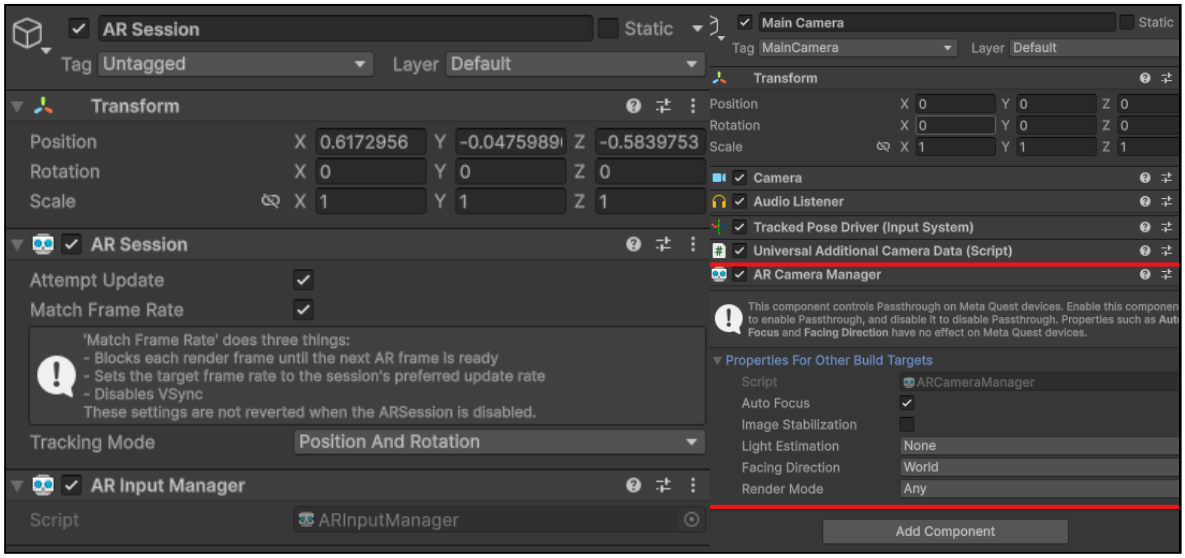

Figure 12: AR Session and AR Camera Manager components in Unity

This component is placed on the `Main Camera` GameObject, which represents the camera in the scene and is a child of the XR Origin (Rig) GameObject mentioned earlier. The options we are interested in are Facing Direction, which should be set to World to make sure we use the rear camera (this is just a precaution, as it should not affect Meta Quest headsets, as mentioned in the GameObject description), and Render Mode, which should be set to Any so that the camera provider decides when to render the background.

Next, we need to add the AR Session component to the scene. This component controls the life cycle of the AR experience by enabling or disabling AR features on the Meta Quest headset. In practice, you can create the GameObject in the Unity Editor by right-clicking in the Hierarchy window and selecting XR > AR Session. The GameObject can be placed anywhere in the scene and will contain the AR Session component. It also includes an AR Input Manager component, which manages input from the headset and controllers.

3.1.3 Headset Features & Limits

To start with the headset, it is important to note that the Meta Quest 3 must be in Developer Mode to run the application. This mode allows developers to test their applications directly on the headset. To enable Developer Mode, you need to create a developer account on the Meta website and link it to your headset. Creating a developer account involves providing some personal information to verify your identity and joining or creating a developer organization. Once this is done, you can enable Developer Mode in the Meta Quest app on your smartphone [7].

The Meta Quest 3 has a feature called Spacial Data, which allows the headset to understand the environment in which it is used. This feature is essential for creating mixed reality applications, as it enables the headset to detect walls, furniture, and other objects in the room.

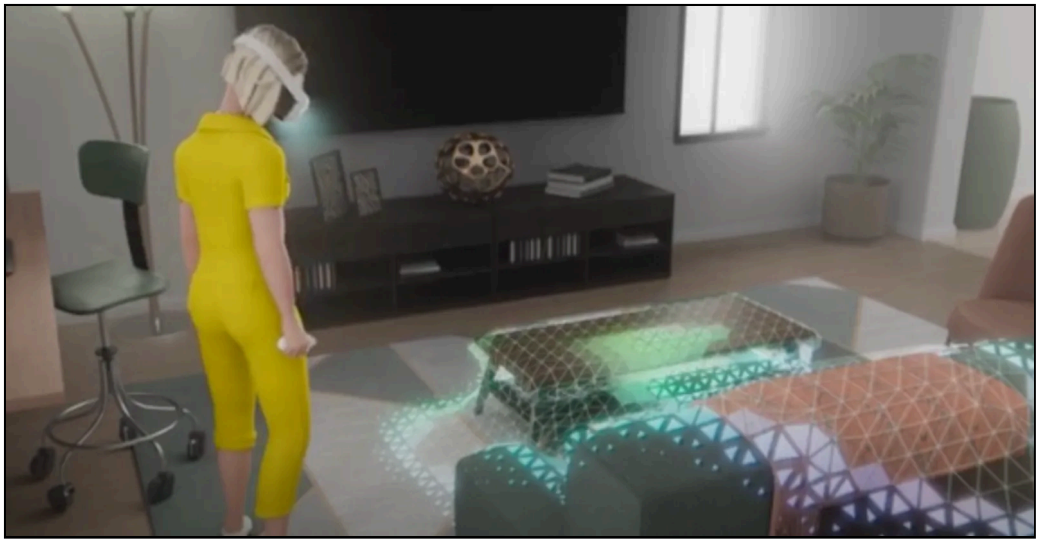

Figure 13: Representation of the scanning process in the Meta Quest 3

Before running an application, the spatial data of a room is obtained by performing a scan. During this process, the user moves around the room to allow the headset to detect all corners and objects. This creates a map of the environment that the headset can later use.

However, the real-world elements that make up the room are not dynamically recognized by the headset while an application is running. Instead, the headset uses the previously stored spatial data to represent the room. Some applications can request a new scan of the environment if needed, which allows them to update the spatial data before continuing.

Remember that our initial project is to be able to find the nearest emergency exit no matter where we are in the building, which already poses a challenge. Without dynamic recognition, everything must be set up in advance in the application to make the project feasible. Another limitation is that the headset cannot store spatial data indefinitely. Currently, it can store up to 15 rooms. To have accurate information for a room, that room cannot be very large, about the size of a room in a house for example. With this limitation, it is clear that scanning an entire building is not possible. These constraints make our initial project very ambitious and require exploring alternative solutions to make it achievable.

3.2 Development Strategy

From what we mentioned earlier, it is clear that the original project of finding the nearest emergency exit in an entire building is too ambitious. We need to redefine the project scope to make it achievable. The goal will stay the same, but we will focus on finding the emergency exit starting from a single room. The pathfinding will then be done within the Unity scene itself, using GameObjects that represent the doors to take to reach the emergency exit.

It is important to note that this report only covers part of the project, specifically the process of detecting planes using the headset. Detecting these planes provides spatial references in the real world. By using scripts, along with research and testing, we can move and rotate virtual elements in our Unity scene so that they match the real environment. This is especially important for aligning doors and other key elements with their real-world positions.

The other parts of the project, such as the design of the doors and their data structure, the application's UI, and the pathfinding algorithm for the doors, are covered in my teammate's report for this project.

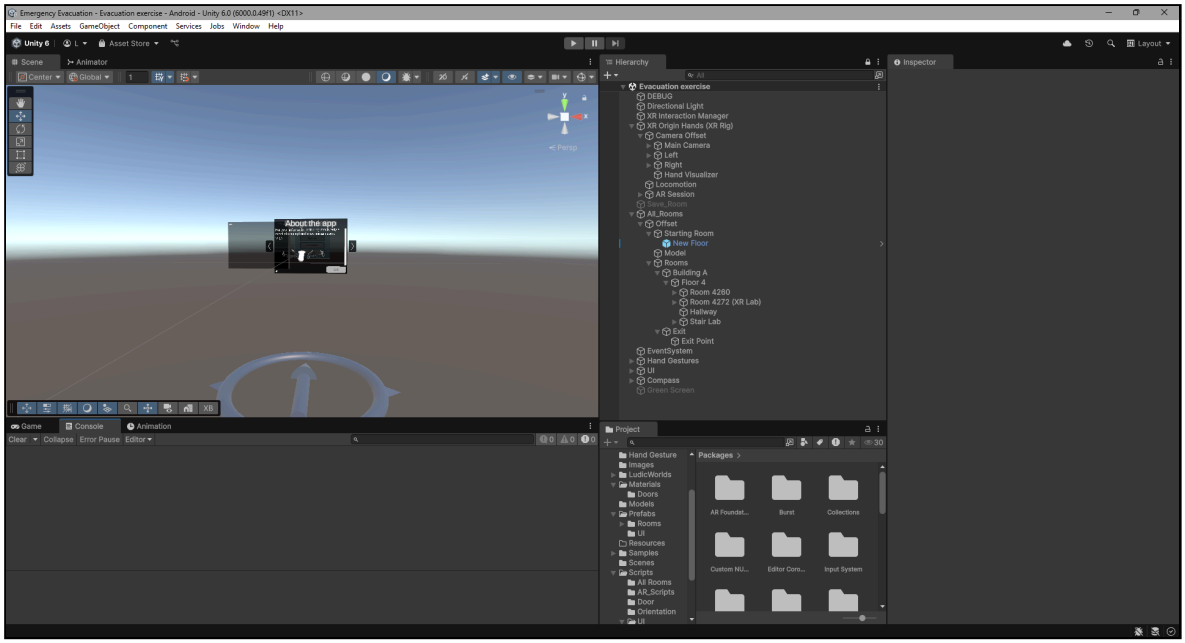

Figure 14: Overview of the project in Unity

This figure shows the current state of the project in Unity. You can see the hierarchy of GameObjects, which contains a lot of information. We will mainly discuss the different scripts attached to the GameObjects that allow us to achieve our goals. The scene may look messy and unorganized, which is normal in mixed reality development. Objects are moved and initialized according to the scripts, so the scene view shows everything at the same time and in the same place. Although we will try to describe them as clearly as possible, all scripts can be found in the project's GitHub repository [8].

3.3 Plane Detection, Management & Classification

Figure 15: Display of planes detected in the Simple AR Sample scene

The first step in our project is to detect planes in the real world using the headset. This is done with the AR Plane Manager component, which is part of the AR Foundation package. However, it is not really useful to have all the furniture and objects of the room; we only want the floor because it is the most useful as a reference point for moving and rotating virtual elements.

The AR Plane Manager retrieves planes from spatial data saved in the headset. Then it will create GameObjects to represent these planes in the scene. The AR Plane Manager is added to the XR Origin (Rig) GameObject, which is the parent of the Main Camera GameObject.

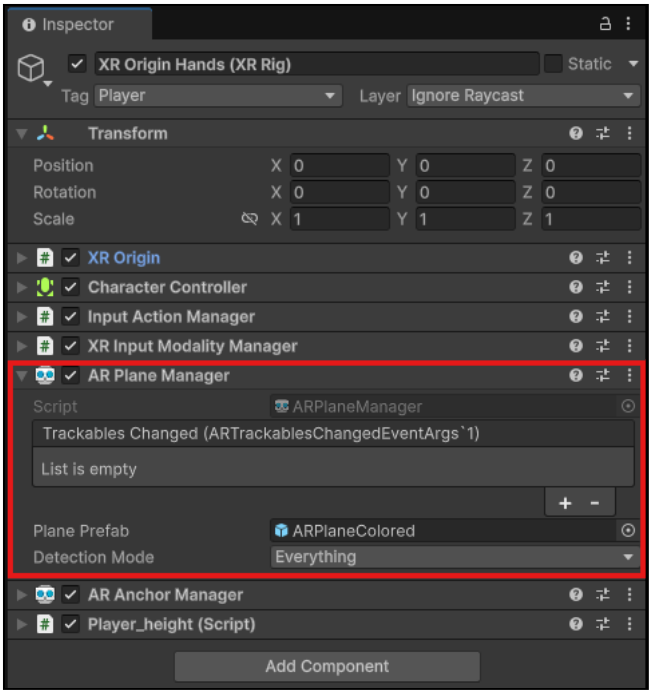

Figure 16: AR Plane Manager component in Unity

This component has several options, but the most important one is the Plane Prefab option. If we want to visualize our plane data, we need to populate this slot with a prefab that specifies exactly how each plane is rendered; this is known as a plane prefab. The AR Plane Manager goes through each individual plane in the spatial data and instantiates the plane prefab to render it in the scene.

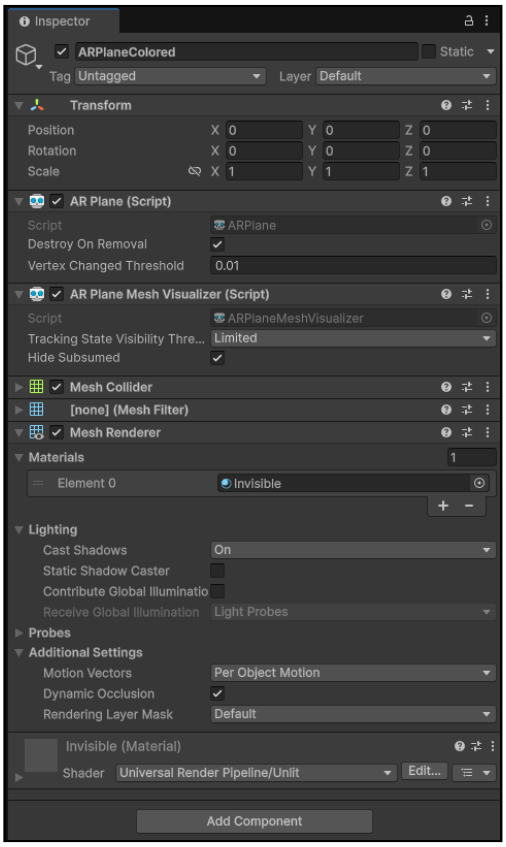

Figure 17: Plane Prefab used by the AR Plane Manager

This prefab comes from a project made by Ludic Worlds and is used to visualize the planes detected by the AR Plane Manager [9]. Each AR Plane represents a detected surface with its own data and follows a life cycle with three main phases: added, updated, and removed. When first detected, the AR Plane Manager creates the plane and notifies the app. If a plane is no longer tracked, it can be removed if the `Destroy On Removal` option is enabled. The AR Plane Mesh Visualizer component is attached to the plane prefab and is responsible for generating and rendering the plane mesh. We can notice that the material used for the mesh renderer is called `Invisible` and this material is fully transparent. This is because the planes are not meant to be displayed directly, but only for debugging purposes, through options in the application's UI that allow them to be shown or hidden.

As mentioned earlier, we do not want all the detected planes, but only the floor plane. To achieve this, we created a script called `AR Plane_Manager`. This script manages the planes detected by the AR Plane Manager and filters out only those that correspond to the floor. We will first examine the attributes of the `AR Plane_Manager` class, and then the methods used to handle the initialization and lifecycle of the detected planes.

```C#
public class ARPlane_Manager : MonoBehaviour
{
    [SerializeField]
    Material _planeMaterial;
    [SerializeField]
    Material _defaultPlaneMaterial;
    private ARPlaneManager _planeManager;
    private List<GameObject> _FloorPlanes = new List<GameObject>();
    private bool _planeVisibility = false;
    private List<Material> _materialsFloor = new List<Material>();
    private List<Material> _materialsDefault = new List<Material>();

    // methods...
}
```

Listing 7: AR Plane_Manager attributes

For the attributes of the `ARPlane_Manager` class, we have:

- `_planeMaterial`: material applied to the planes detected as "floor".
- `_defaultPlaneMaterial`: default material, which is invisible.
- `_planeManager`: the AR Foundation manager that detects surfaces.
- `_FloorPlanes`: the list of planes that are floors.
- `_planeVisibility`: boolean to know if the planes are visible or not.
- `_materialsFloor` and `_materialsDefault`: lists that contain the materials to apply. Actually, to apply a material to a game object, we need to give it a list of materials, which is why we have two separate lists.

```C#
void Start()
{
    _planeManager = GetComponentInParent<ARPlaneManager>();
    _planeManager.trackablesChanged.AddListener(OnPlanesChanged);
    _materialsFloor.Add(_planeMaterial);
    _materialsDefault.Add(_defaultPlaneMaterial);
}
private void OnDestroy()
{
    _planeManager.trackablesChanged.RemoveListener(OnPlanesChanged);
}
```

Listing 8: Start and OnDestroy methods from AR Plane_Manager script

The `Start` method is called when the script is first initialized, just after the application is launched. It retrieves the AR Plane Manager component from the parent GameObject and adds a listener to the `trackablesChanged` event, which is triggered whenever planes are added, updated, or removed. To be more precise, our script is attached to a GameObject named `AR Plane Manager`, which is not directly linked to AR Foundation.

It is a child of the `AR Session` GameObject, which itself is a child of the XR Origin (Rig) containing the `AR Plane Manager` component. The method `GetComponentInParent<ARPlaneManager>()` searches the entire hierarchy above the current GameObject and retrieves the first `AR Plane Manager` component it finds. The materials for the floor and default planes are also added to their respective lists. The `OnDestroy` method is called when the script is destroyed, and it removes the listener from the `trackablesChanged` event to avoid memory leaks, but also to avoid calling the method when the script is no longer active.

```C#
private void OnPlanesChanged(ARTrackablesChangedEventArgs<ARPlane> args)
{
    if (args.added.Count > 0)
    {
        foreach (var plane in _planeManager.trackables)
        {
            if (plane.classifications != PlaneClassifications.Floor)
            {
                plane.gameObject.SetActive(false);
            }
            else
            {

                plane.GetComponent<MeshRenderer>().SetMaterials(_materialsFloor
                int LayerIgnoreRaycast = LayerMask.NameToLayer("ARPlane");
                plane.gameObject.layer = LayerIgnoreRaycast;
                _FloorPlanes.Add(plane.gameObject);
                SetPlaneVisibility(_planeVisibility);
            }
        }
    }
}
```

Listing 9: OnPlanesChanged method from AR Plane_Manager script

The `OnPlanesChanged` method is registered to the `trackablesChanged` event in the `Start` method, which means it will be called whenever planes are added, updated, or removed. First, it checks if any planes were added, because we only want to process new planes detected by the AR Plane Manager. When a new plane is added, we go through all detected planes and check if it is a floor. AR Foundation allows classifying planes using the `PlaneClassifications` enum, which includes options like Ceiling, Couch, DoorFrame, Floor, Seat, Table, WindowFrame, and more. In our case, if the plane is not a floor, we deactivate its GameObject. Otherwise, we assign it the floor material for rendering. Initially, we tried a different method to display the AR Plane, but we decided to use a colored material and adjust its transparency depending on whether we want to show it or not (so now, `_defaultPlaneMaterial` and `_materialsDefault` are useless).

We also assign the plane a LayerMask, which will be useful later for detecting it with a raycast. Finally, we add the plane to our `_FloorPlanes` list and call the `SetPlaneVisibility` method, passing the boolean that controls the visibility of planes.

```C#
public void TogglePlaneVisibility()
{
    _planeVisibility = !_planeVisibility;
    SetPlaneVisibility(_planeVisibility);
}
public List<GameObject> GetFloorPlanes()
{
    return _FloorPlanes;
}

private void SetPlaneVisibility(bool isVisible)
{
    float fillAlpha = isVisible ? 0.5f : 0f;
    Debug.Log("-> Setting visibility for plane");
    foreach (var plane in _FloorPlanes)
    {
        SetTrackableAlpha(plane, fillAlpha);
    }
}

private void SetTrackableAlpha(GameObject trackable, float fillAlpha)
{
    MeshRenderer meshRenderer = trackable.GetComponent<MeshRenderer>();
    if (meshRenderer == null)
    {
        meshRenderer = trackable.GetComponentInChildren<MeshRenderer>();
    }

    if (meshRenderer != null)
    {
        Color color = meshRenderer.material.color;
        color.a = fillAlpha;
        meshRenderer.material.color = color;
    }
    else
    {
        Debug.Log("-> Can't find 'meshRenderer' - on: " + trackable.name);
    }
}
```

Listing 10: Floor Visibility methods from AR Plane_Manager script

The `TogglePlaneVisibility` method changes the visibility of all floor planes: if they are visible, it makes them invisible, and if they are invisible, it makes them visible again by updating their transparency, all this by calling `SetPlaneVisibility` method after changing the boolean value. This method is public so that any other script can change the visibility of the floor planes. For example, as seen in my second lab work, a UI element can trigger this method to show or hide the floor planes.

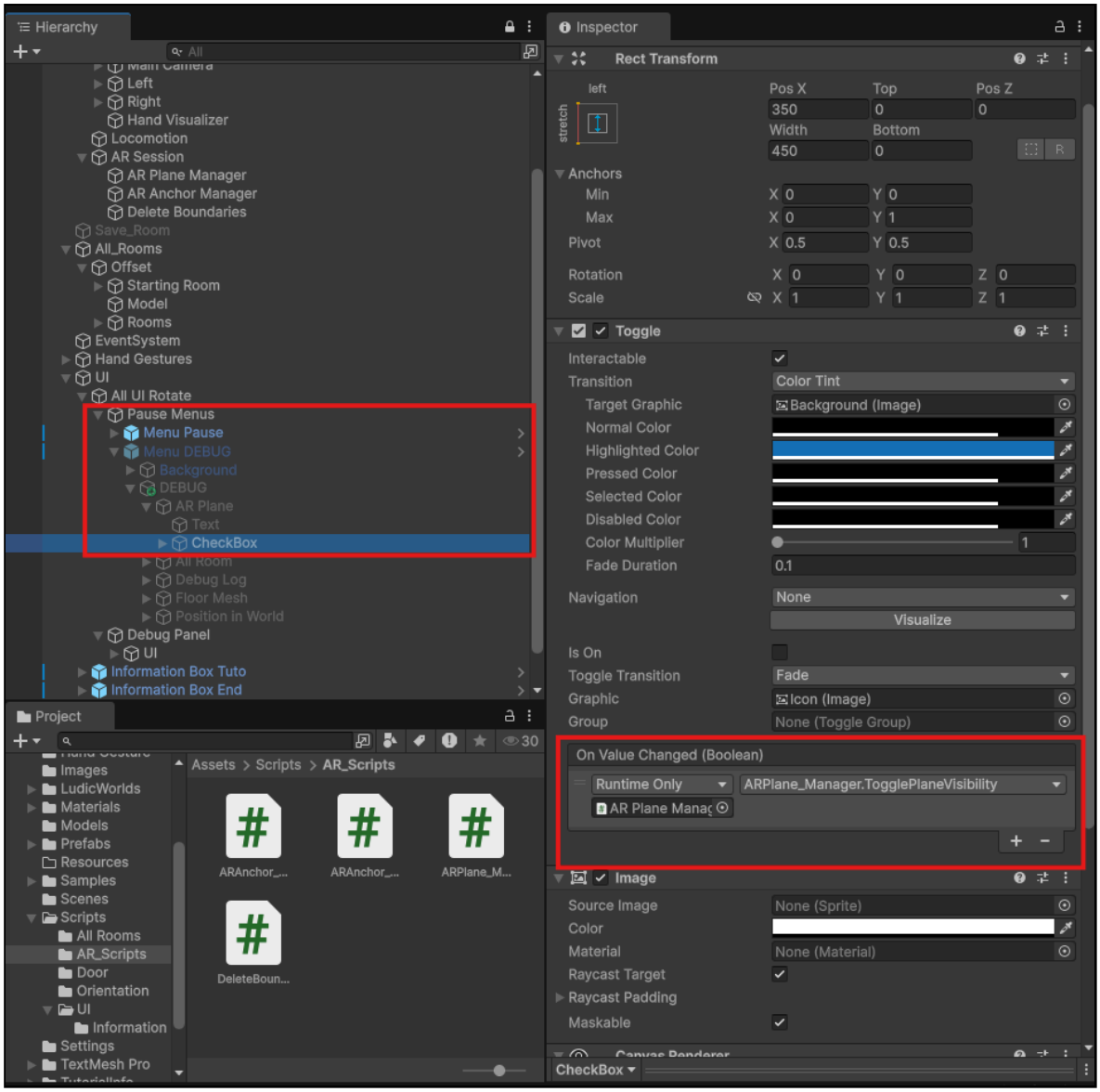

Figure 18: Checkbox calling TogglePlaneVisibility when checked or unchecked

`GetFloorPlanes` is a public method that simply returns the list of all floor planes that have been detected, so other scripts can use or interact with them.

`SetPlaneVisibility` calls `SetTrackableAlpha` with the correct alpha value: 0 for full transparency or 0.5 for semi-transparency. Then, `SetTrackableAlpha` finds the MeshRenderer component on the object. If it is not on the object, it looks in the child GameObjects. After that, it copies the MeshRenderer color, changes the alpha, and assigns the new color back to the MeshRenderer.

Figure 19: Displayed floor plane in our application

3.4 Virtual Element Integration

3.4.1 Floor Mesh Copy

Now that we have our AR Floors, we need to be able to get information from the plane in order to reproduce this same floor as a virtual element in the Unity scene. To do this, we implement a `Floor_Plane` script that allows us to perform a raycast using the `ARPlane` layer mask we added in the previous script, so that it only detects AR Planes and nothing else.

```C#
public class Floor_Plane : MonoBehaviour
{
    private GameObject _floorPlane = null;
    private LayerMask _layerMask;
    private Player_height _playerHeight;

    void Start()
    {
        int layer = LayerMask.NameToLayer("ARPlane");
        _layerMask = 1 << layer;
        _playerHeight = GetComponentInParent<Player_height>();
    }

    // other methods...
}
```

Listing 11: Floor_Plane attributes and initialisation

To begin, we have the following attributes:

- `_floorPlane`: a GameObject that stores the AR Plane detected by the raycast.
- `_layerMask`: represents the "ARPlane" LayerMask used to filter the elements detected by the raycast. A bit shift to the left is applied (`1 << layer`), which means that the LayerMask is not just a simple number but a binary mask where each bit corresponds to a layer in Unity. By shifting 1 to the left by the index of the "ARPlane" layer, we activate only the bit for that layer, so the raycast will only detect objects assigned to "ARPlane".
- `_playerHeight`: an attribute that provides information and methods about the player's height during the execution of the application, based on the camera's y position and the detected AR Floor Plane. The class looks for the `Camera` component during initialization and retrieves its y position. It can then return this value to other scripts through the `GetPlayerHeight` method, and update it with the `UpdatePlayerHeight` method, which takes as a parameter the floor detected in the `Floor_Plane` script.

```C#
public class Player_height : MonoBehaviour
{
    private float _playerHeight;
    private Camera _playerCamera;

    private void Start()
    {
        _playerCamera = GetComponentInChildren<Camera>();
        _playerHeight = _playerCamera.transform.position.y;
    }

    public float GetPlayerHeight() { return _playerHeight; }

    public void UpdatePlayerHeight(GameObject currentARFloor)
    {
        _playerHeight = Math.Abs(currentARFloor.transform.position.y -
        _playerCamera.transform.position.y);
    }
}
```

Listing 12: Player_height script

With all this, we want to search for an AR Plane under our feet, since we kept only the AR Planes of type 'Floor'. The method is the same as the one we discussed in Lab Work 3.

```C#
 void Update()
{
    if(_floorPlane == null)
    {
        Ray ray = new Ray(transform.position, Vector3.down);
        RaycastHit hit;

        if (Physics.Raycast(ray,out hit, 5.0f, _layerMask))
        {
            Debug.Log("Raycast hit");
            if (hit.transform.GetComponent<ARPlane>() != null)
            {
                if (_floorPlane == null)
                {
                    _floorPlane = hit.transform.gameObject;
                    _playerHeight.UpdatePlayerHeight(_floorPlane);
                    Debug.Log("Floor plane found: " + _floorPlane.name);
                }
            }
        }
    }
}

public GameObject GetFloorPlane()
{
    return _floorPlane;
}
```

Listing 13: Getting Floor Plane with Floor_Plane script

As long as we have not found any Floor Plane, the `Update` method, which runs every frame, continuously searches for this plane using a raycast oriented to our feet with `Vector3.down` in the instance created for the `ray` variable. Since the script is attached to a child GameObject of the `Main Camera`, `transform.position` corresponds to the camera's position. If a GameObject is detected with the same LayerMask, we check if it has an AR Plane component. If so, and if we have not already found a floor, we assign it to the `_floorPlane` attribute and update the player's height. This will be useful later for adding a practical element in the project that we will discuss afterward. With the public method `GetFloorPlane`, we can now provide other scripts with information about the AR Plane that represents the floor based on spatial data from the headset.

We need a method to show the floor's MeshFilter properties: vertices, triangles, and normals. Vertices are points in 3D space defining the mesh shape. Triangles are sets of three vertices forming the mesh surfaces. Normals are vectors perpendicular to the surface, showing which way it faces.

```C#
public void PrintFloorMesh()
{
    Mesh mesh = _floorPlane.GetComponent<MeshFilter>().sharedMesh;
    Debug.Log("-> Floor Mesh\nPosition: " + _floorPlane.transform.position + ", Rotation: " + _floorPlane.transform.eulerAngles + "\nVertices:");

    string vertices = "";
    for (int i = 0;  i < mesh.vertices.Length; i++)
    {
        vertices += mesh.vertices[i].ToString();
        if(i <  mesh.vertices.Length - 1)
        {
            vertices += ", ";
        }
    }
    Debug.Log(vertices + "\nTriangles:");
    string triangles = "[";
    for (int i = 0; i < mesh.triangles.Length;i++)
    {
        triangles += mesh.triangles[i].ToString();
        if (i < mesh.triangles.Length - 1)
        {
            triangles += ", ";
        }
    }
    Debug.Log(triangles + "]\nNormals:");
    string normals = "";
    for(int i = 0;i < mesh.normals.Length;i++)
    {
        normals += mesh.normals[i].ToString();
        if( i < mesh.normals.Length - 1)
        {
            normals += ", ";
        }
    }
    Debug.Log(normals);
}
```

Listing 14: Printing Mesh properties from AR Floor

The method seems long, but it only reads the different fields and displays them in a readable format in the debug logs. With this information, we can start creating scripts that generate a GameObject with geometric properties similar to our AR Plane.

3.4.2 Creating GameObject from AR Floor

First, we create a script that allows us to access the MeshFilter component of a given GameObject and fill its mesh properties with the same data displayed by the `PrintFloorMesh` method from the `Floor_Plane` script. This `Room_Mesh` script will let us manually set the desired properties and apply them to the GameObject's MeshFilter.

```C#
public class Room_Mesh : MonoBehaviour
{
    [SerializeField]
    GameObject _objectToSave;

    void Start()
    {
        Mesh mesh = new Mesh();
        Vector3[] vertices = new Vector3[6];

        vertices[0] = new Vector3(4.66f, 0f, 3.24f);
        // ...
        mesh.vertices = vertices;

        Vector3[] normals = new Vector3[6];
        normals[0] = Vector3.up;
        // ...
        mesh.normals = normals;

        int[] triangles = {0, 1, 5, 1, 2, 5, 2, 3, 5, 3, 4, 5};
        mesh.triangles = triangles;

        mesh.RecalculateBounds();
        mesh.RecalculateNormals();
        mesh.RecalculateTangents();
        mesh.RecalculateUVDistributionMetrics();
        mesh.name = "RoomMesh";

        if (_objectToSave != null)
        {
            _objectToSave.GetComponent<MeshFilter>().mesh = mesh;
        }
    }
}
```

Listing 15: Room_Mesh script

In fact, we want a GameObject with a corresponding MeshFilter permanently, not just during the application's runtime. To achieve this, we found a way to save a GameObject as a prefab when entering Unity's Play Mode.

```C#
public class Save_GameObject : MonoBehaviour {
    #if UNITY_EDITOR
    [MenuItem("GameObject/Save as Prefab")]
    static void SaveObject() {
        GameObject selectedObject = Selection.activeGameObject;
        string localPath = "";
        string directoryPath = "";
        if (selectedObject.transform.parent != null)
        {
            if (!Directory.Exists("Assets/Prefabs/Rooms/" +
            selectedObject.transform.parent.name))
                AssetDatabase.CreateFolder("Assets/Prefabs/Rooms",
                selectedObject.transform.parent name);
            directoryPath = "Assets/Prefabs/Rooms/" +
            selectedObject.transform.parent.name + "/";
            localPath = directoryPath + selectedObject.name + ".prefab";
        } else {
            directoryPath = "Assets/Prefabs/Rooms/";
            localPath = "Assets/Prefabs/Rooms/" + selectedObject.name +
            ".prefab";
        }
        MeshFilter meshFilter = selectedObject.GetComponent<MeshFilter>();
        if (meshFilter != null)
        {
            Mesh mesh = meshFilter.sharedMesh;
            if (mesh != null)
            {
                string meshPath = directoryPath + selectedObject.name +
                "_Mesh.asset";
                AssetDatabase.CreateAsset(mesh, meshPath);
                meshFilter.sharedMesh =
                AssetDatabase.LoadAssetAtPath<Mesh>(meshPath);
            }
        }
        localPath = AssetDatabase.GenerateUniqueAssetPath(localPath);
        bool prefabSuccess;
        PrefabUtility.SaveAsPrefabAssetAndConnect(selectedObject, localPath,
        InteractionMode.UserAction, out prefabSuccess);
        if (prefabSuccess == true)
            Debug.Log("Prefab was saved successfully");
        else
            Debug.Log("Prefab failed to save" + prefabSuccess);
    }
    #endif
}
```

Listing 16: Save_GameObject script

All this without needing to deploy the application to the headset. The `selectedObject` corresponds to the GameObject selected by a right click on it in the hierarchy window. It checks if the GameObject has a parent and creates a folder in "Assets/Prefabs/Rooms/" accordingly. If the GameObject has a MeshFilter with a mesh, the mesh is saved as a separate asset and assigned back to the MeshFilter. Finally, the GameObject is saved as a prefab using `PrefabUtility.SaveAsPrefabAssetAndConnect`, and a message is logged to indicate whether the save succeeded.

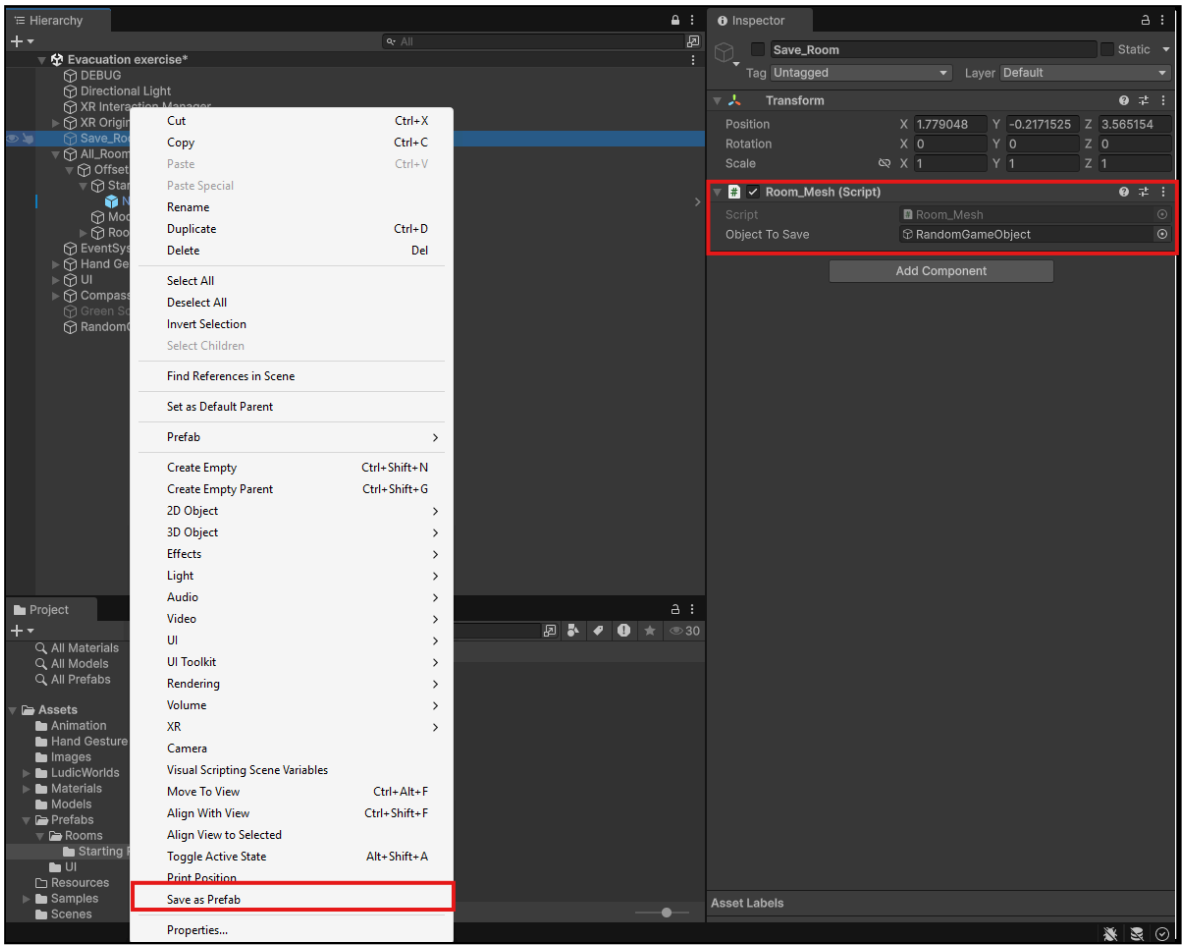

Figure 20: Saving prefab in Editor

Note that in the example shown in the figure above, you need to right-click on `RandomGameObject` and select `Save as Prefab`, not on the `Save_Room` GameObject which was only used for the screenshot, after entering Play Mode, making sure that this GameObject has a MeshFilter component, whether it is empty or not. Also, don't forget to reference `RandomGameObject` in the `Room_Mesh` script, as shown in the Inspector of the editor in the figure above. The `Save as Prefab` option is made possible with the `Save_GameObject` script with `[MenuItem("GameObject/Save as Prefab")]`.

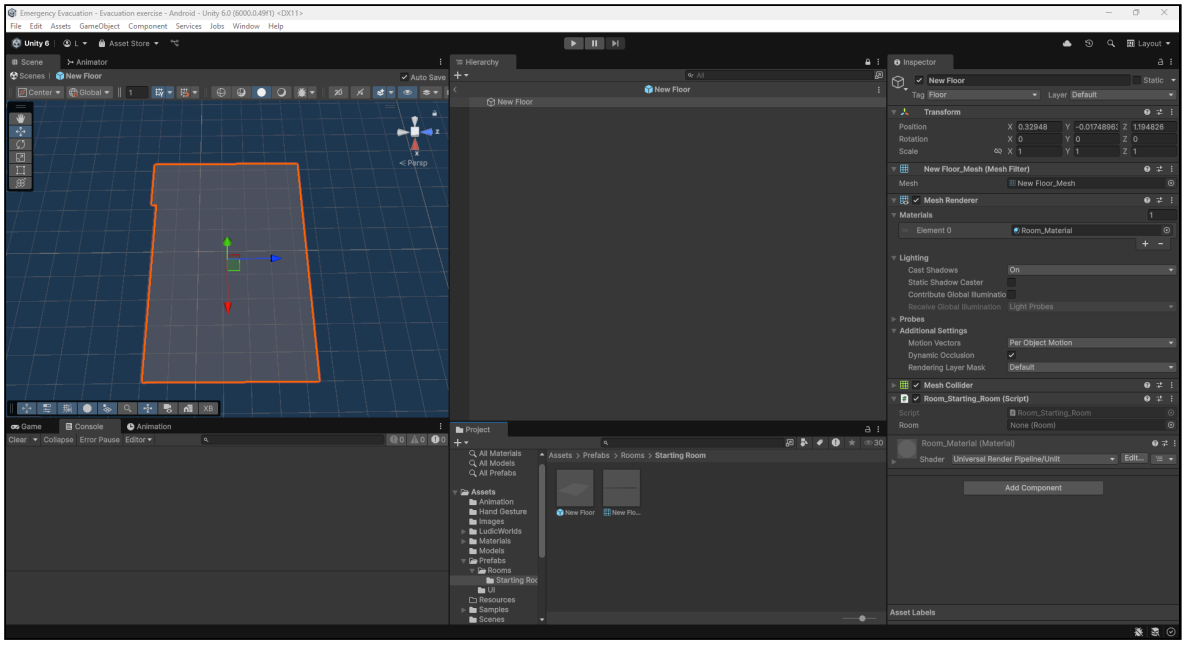

Figure 21: Result of saving prefab with floor mesh properties

If we compare this floor with what can be seen in Figure 19, we can roughly see that we have successfully obtained a floor with similar geometric properties.

3.5 Reality Alignment

Before talking about any script, it is important to understand how the GameObjects representing virtual elements are displayed in the scene, and especially how they are organized in the GameObject hierarchy.

In our case, we can see the following figure showing all the virtual floors and doors that our application will use to create a path of doors and calculate the shortest route to find the nearest emergency exit.

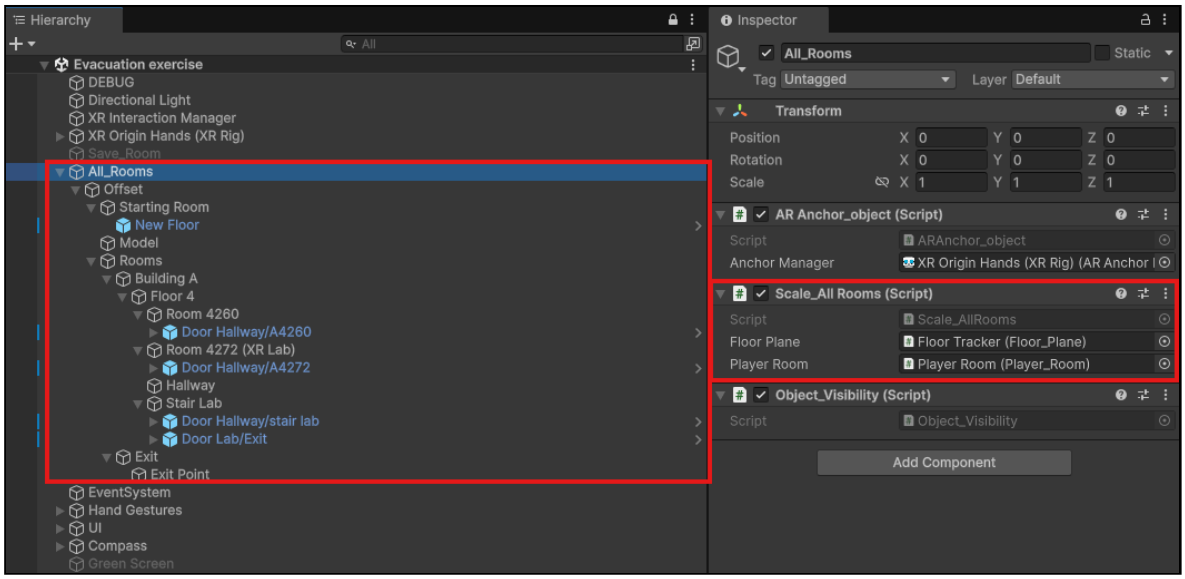

Figure 22: All_Rooms GameObject with Scale_All Rooms script

We can observe that the parent of all these elements is named `All_Rooms`. Among its children, there is the `Starting Room`, which contains the prefabs of the starting floors. In our project, within our scope, we only have one. Then, we find the `Rooms` GameObject, which first includes GameObjects corresponding to the different building sections. In our case, we only have the one where the starting room is located. Inside it, we also find the different floors, as well as `Exit`, which is used to create exit points for the last doors, representing the emergency exits. The details of this are explained in my teammate's report. Within the `Floor 4` GameObject, we can see the different rooms, each having child prefabs of doors, also described in the teammate's report. What interests us here is the `Scale_All Rooms` script, which allows us to rotate and move the `All_Rooms` GameObject, and therefore to move all the virtual elements at once. The script is quite long, so we will break it down step by step.

```C#
public class Scale_AllRooms : MonoBehaviour
{
    [SerializeField]
    private Floor_Plane _floorPlane;

    [SerializeField]
    private Player_Room _playerRoom;

    private List<GameObject> _allRoomsFloor = new List<GameObject>();
    private Dictionary<GameObject, Vector3> _allRoomsFloorDefaultCoordinate =
    new Dictionary<GameObject, Vector3>();

    private bool _currentFloorFound = false;
    private GameObject _currentARFloor = null;
    private GameObject _currentAllRoomFloor = null;
}
```

Listing 17: Scale_All Rooms attributes

Here is a description of the different attributes:

- `_floorPlane`: As we saw in the `Floor_Plane` script, this attribute stores the AR Floor Plane and will be used as a spatial reference to place the floor prefab correctly.
- `_playerRoom`: Represents the room where the player is located, as well as the last door the player went through. More details about the `Player_Room` script can be found in my teammate's report.
- `_allRoomsFloor`: Contains the list of all floor prefabs. In our case, we only have one, but the idea before narrowing our scope was to store several starting rooms with multiple floor prefabs.
- `_allRoomsFloorDefaultCoordinate`: A dictionary that links each floor prefab with its (x, y, z) coordinates as a `Vector3` value.
- `_currentFloorFound`: A boolean set to false when no floor prefab is found in the `Starting Room` GameObject's children, and true otherwise.
- `_currentARFloor`: Stores the AR Floor GameObject found by `_floorPlane`.
- `_currentAllRoomFloor`: Stores the floor prefab GameObject that best matches the AR Floor.

```C#
void Start()
{
    for (int i = 0; i < transform.GetChild(0).childCount; i++)
    {
        Transform child = transform.GetChild(0).GetChild(i);
        if (child.CompareTag("Start"))
        {
            for (int j = 0; j < child.childCount; j++)
            {
                Transform grandChild = child.GetChild(j);
                if (grandChild.CompareTag("Floor"))
                {
                    _allRoomsFloor.Add(grandChild.gameObject);

                    _allRoomsFloorDefaultCoordinate.Add(grandChild.gameObject,
                    grandChild.transform.position);
                }
            }
        }
    }
}

void Update()
{
    if(!_currentFloorFound)
    {
        _currentARFloor = _floorPlane.GetFloorPlane();
        if (_currentARFloor != null)
        {
            _currentFloorFound = true;
            SearchMatchingFloor();
            ApplyRoomPlayer();
        }
    }
}
```

Listing 18: Start and Update methods from Scale_AllRooms script

At initialization, we go through the children of the `All_Rooms` GameObject, starting with `Offset`. Then we check its children until we find the `Starting Room` GameObject with the tag "Start". From there, we look for the floor prefabs, making sure they have the tag "Floor", and we store them in the list and dictionary together with their `transform.position` linked to each prefab.

The `Update` method checks every frame to detect the AR Floor. Once it finds it, two methods are called. The first one, `ApplyRoomPlayer`, is not detailed here but it sets the current player to the room linked to the selected floor prefab. The second one, and the most important for us, is the `SearchMatchingFloor` method.

```C#
private void SearchMatchingFloor(){
    List<float> floatList = new List<float>();
    foreach (var plane in _allRoomsFloor){
        Mesh meshAllRoomFloor = plane.GetComponent<MeshFilter>().sharedMesh;
        Mesh meshARFloor = _currentARFloor.GetComponent<MeshFilter>().sharedMesh;
        if (meshAllRoomFloor != null && meshARFloor != null){
            float cumulDistance = 0;
            for (int i = 0; i < meshARFloor.vertexCount; i++){
                float min = float.MaxValue;
                for (int j = 0; j < meshAllRoomFloor.vertexCount; j++{
                    float distance = Vector3.Distance(meshARFloor.vertices[i], meshAllRoomFloor.vertices[j]);

                    if(distance <  min){
                        min = distance;
                    }
                }
                cumulDistance += min;
            }
            floatList.Add(cumulDistance);
        }
    }
    float minVal = float.MaxValue;
    int pos = 0;
    for (int i = 0; i < floatList.Count; i++){
        if (floatList[i] < minVal){
            minVal = floatList[i];
            pos = i;
        }
    }
    _currentAllRoomFloor = _allRoomsFloor[pos];
    MoveAllRoom();
}
```

Listing 19: SearchingMatchingFloor method from Scale_AllRooms script

The method tries to find the floor prefab (in `_allRoomsFloor`) that is the most similar to the detected AR Floor (`_currentARFloor`) by comparing their meshes (vertices only). Once found, it stores it in `_currentAllRoomFloor` and calls `MoveAllRoom` to align the whole scene with this floor.

We go through each floor prefab (`foreach (var plane in _allRoomsFloor)`) and get its mesh. We also get the mesh of the detected AR floor. The process is as follows:

For each vertex (3D point) of the AR mesh (`meshARFloor`), we look for the closest vertex among all vertices of the current floor prefab mesh (`meshAllRoomFloor`). We measure the distance using `Vector3.Distance`. Once we find the smallest distance for this vertex, we add it to `cumulDistance` (in other words, we find the most likely/similar vertex of the prefab to the AR Floor vertex).We will add each minimal distance to the float by doing `cumulDistance += min`. Once done, we add the score `cumulDistance` to a float list to save the score of the current floor prefab.

Once this is done for all prefabs, we go through all the `cumulDistance` scores stored in a list `floatList`. We get the index of the smallest value in this list, and we obtain the most similar prefab with `_currentAllRoomFloor = _allRoomsFloor[pos]` where `pos` is the index of the smallest value in the `floatList`. It will then call the `MoveAllRoom` method as shown below:

```C#
public void MoveAllRoom()
{
    transform.GetChild(0).transform.localPosition = -
    transform.GetChild(0).transform.InverseTransformPoint(_currentAllRoomFloor.

    transform.parent = _currentARFloor.transform;
    transform.SetLocalPositionAndRotation(Vector3.zero, Quaternion.identity);

    Transform pos = _currentAllRoomFloor.transform;
    Vector3 angle = Vector3.zero;

    while (pos.parent != transform)
    {
        angle += pos.localEulerAngles;
        pos = pos.parent;
    }

    transform.localEulerAngles = -angle;
    transform.parent = null;
}
```

Listing 20: MoveAllRoom method from Scale_AllRooms script

This method was hard to make and is kind of a “quick fix.” It was tough to find a way to move and rotate, but it works. First, we need to mention that the `Offset` GameObject in the `All_Rooms` hierarchy exists only as a pivot for this method, making the movement possible with our approach.

The method proceeds as follows:

First, we get the coordinates corresponding to the local distance between our floor prefab and the `Offset` GameObject using `transform.InverseTransformPoint()` at line 3, which gets the prefab's coordinates relative to `Offset` (for example, if the world coordinates of `Offset` are (10, 0, 0) and those of the prefab are (12, 0, 0), we will get (2, 0, 0)). We use the negative because we want to apply an offset for the next steps. When we talk about local positions, it means we are referring to the position relative to a specific GameObject, not to the world's origin.

Then, we temporarily set the AR Floor as the parent of `All_Rooms`, making sure that `All_Rooms` has a local position of (0,0,0) and the same rotation relative to the AR Floor. With AR Floor being parent of `All_Rooms`, that also means that `Offset` and the prefab become AR Floor's children and this affects their position as follows: with the `Offset` having the distance between itself and the prefab, having this distance as negative in `Offset.transform.localPosition` will shift the prefab to be "centered" to `All_Rooms`, and as said before, `All_Rooms` became a child of the AR Floor, which means that we now have the prefab aligned to the AR Floor. The figure below schematizes the result of this process, green arrows represent the hierarchy of each, the points represent the central axis relative to AR Floor and the red line represents the distance obtained at the beginning of the method applied to `Offset.transform.localPosition`.

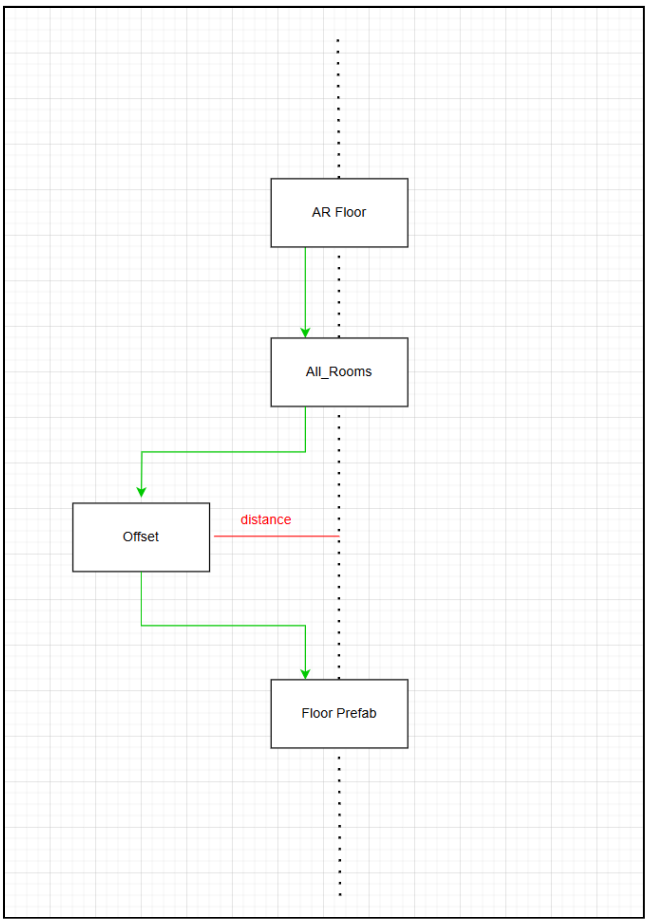

Figure 23: Diagram showing the Offset shift

Finally, we set a variable called `pos`, which is a Transform Component, to the floor prefab. Then, we traverse the hierarchy by moving up levels until we find the GameObject `All_Rooms`. During this traversal, we accumulate the different local rotation angles, allowing us to find the angle difference between the floor prefab and `All_Rooms` (remember that `All_Rooms` is spatially identical to the AR Floor thanks to line 5 and 6, where we clearly set `All_Rooms` at the origin relative to the AR Floor). We then finish by applying the inverse rotation to `All_Rooms` to rotate all the virtual elements, not just the floor prefab and setting `All_Rooms` parent to `null`.

3.6 Feature Enhancements

3.6.1 Directional Compass

To enhance our project, we decided to add a compass at the player's feet to indicate the direction they should follow to reach the correct door in the path determined by the application. First, here is the 3D model of the compass, represented by a GameObject `Compass` with two child GameObjects: the body and the arrow.

First, for the GameObject `Compass`, we have a first script, `Follow_player`, which updates the position of the 3D model to place it at the player's feet.

C#

```
1  public class Follow_player : MonoBehaviour
2  {
3      [SerializeField]
4      private Camera _camera;
5
6      [SerializeField]
7      private Player_height _player_Height;
8
9      void Update()
10     {
11         if( _camera != null && (_camera.transform.position.x !=
           transform.position.x || _camera.transform.position.z !=
           transform.position.z))
12         {
13             var newPos = _camera.transform.position;
14             newPos.y = _camera.transform.position.y -
               _player_Height.GetPlayerHeight();
15
16             transform.position = newPos;
17         }
18     }
19 }
```

Listing 21: Follow_player script

This script makes the GameObject follow the player's camera on the X and Z axes. It updates the GameObject's position each frame so that its height (Y position) stays a certain distance below the camera, based on the player's height. Remember that we have already seen the `Player_height` script, which allows us to get the player's height. This way, we can set the compass's height by subtracting the `_player_Height` Y position from the Camera's Y position.

Next, we have another script, attached to the GameObject representing the arrow, which points to the desired GameObject. In our project, this is the next door the player should go to.

This script controls the compass arrow to point towards a target object:

- In `Start`, it collects all MeshRenderers of the compass (and its children) and disables them, so the compass is initially invisible.
- In `Update`, if a target (`_currentFollow`) is set, the script first enables all the MeshRenderers if they are not already visible, making the compass appear. Then it rotates the GameObject to face the target using `LookAt`. The X and Z rotation axes are reset to 0, and 180 degrees is added to the Y axis to align the arrow correctly. A target is set via the public `ChangeObjectFollow` method, allowing any other script to change it.
- If no target is set and the compass is visible, `Update` disables all MeshRenderers, hiding the compass.

The full script can be found on the next page.

```C#
public class Object_Follow : MonoBehaviour
{
    private Transform _currentFollow = null;
    private bool _isVisible = false;
    private List<MeshRenderer> _compassVisual = new List<MeshRenderer>();

    private void Start()
    {
        foreach (MeshRenderer renderer in transform.parent.GetComponentsInChildren<MeshRenderer>(true))
        {
            _compassVisual.Add(renderer);
            renderer.enabled = false;
        }
    }

    void Update()
    {
        if (_currentFollow != null){
            if(!_isVisible){
                foreach(MeshRenderer renderer in  _compassVisual) {
                    renderer.enabled = true;
                }
                _isVisible = true;
            }
            transform.LookAt(_currentFollow.position);
            var rotation = transform.eulerAngles;
            rotation.x = 0;
            rotation.y += 180;
            rotation.z = 0;
            transform.eulerAngles = rotation;
        }
        else if (_isVisible) {
            foreach (MeshRenderer renderer in _compassVisual){
                renderer.enabled = false;
            }
            _isVisible = false;
        }
    }

    public void ChangeObjectFollow(Transform obj){
        _currentFollow = obj;
    }
}
```

Listing 22: Object_Follow script

3.6.2 Recalibrating Virtual Environment

Here, we make a tool that can help when the virtual objects get out of place. This can happen, for example, if you take off the headset and it goes into sleep mode. If you move or turn the headset while it is in sleep mode, when you go back to the app, the virtual objects will probably be in the wrong place because the headset is not in the right position anymore.

To fix this, we add a `Recalibrate` button in the UI with the right methods to call when the button is pressed.

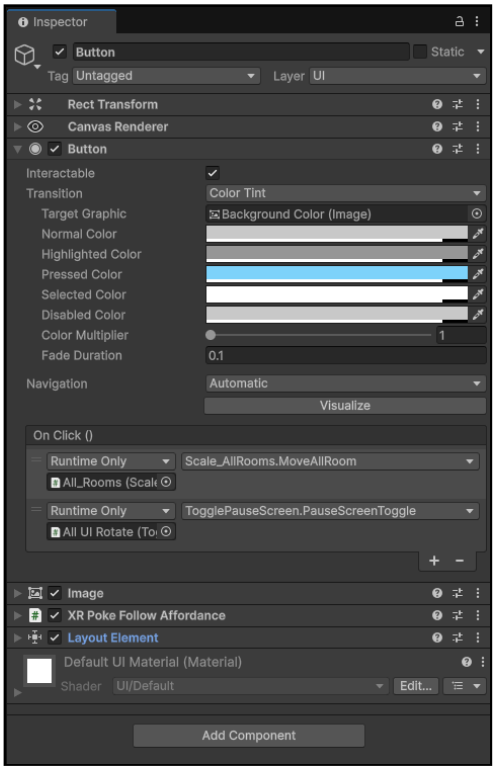

Figure 24: Recalibrate button with OnClick methods

Among the methods called by `OnClick`, there is the `MoveAllRooms` method, which simply restarts the alignment process. Then, it calls another method not described in this report, which is the pause menu toggle. Since the pause menu must be visible to access this button, this will close the pause menu, showing the player that something has happened.

So, the development of the project, as far as my part is concerned, ends here. The other components, scripts and features are described in detail in my teammate's report. We will now shift our focus to the internship review, where I will discuss the next steps of my internship, reflect on what I have learned so far, and finally, provide a conclusion to close this report.

4 | Internship Review

4.1 Review and Feedback

To start this review, we can first look at the actual Gantt chart for my internship.

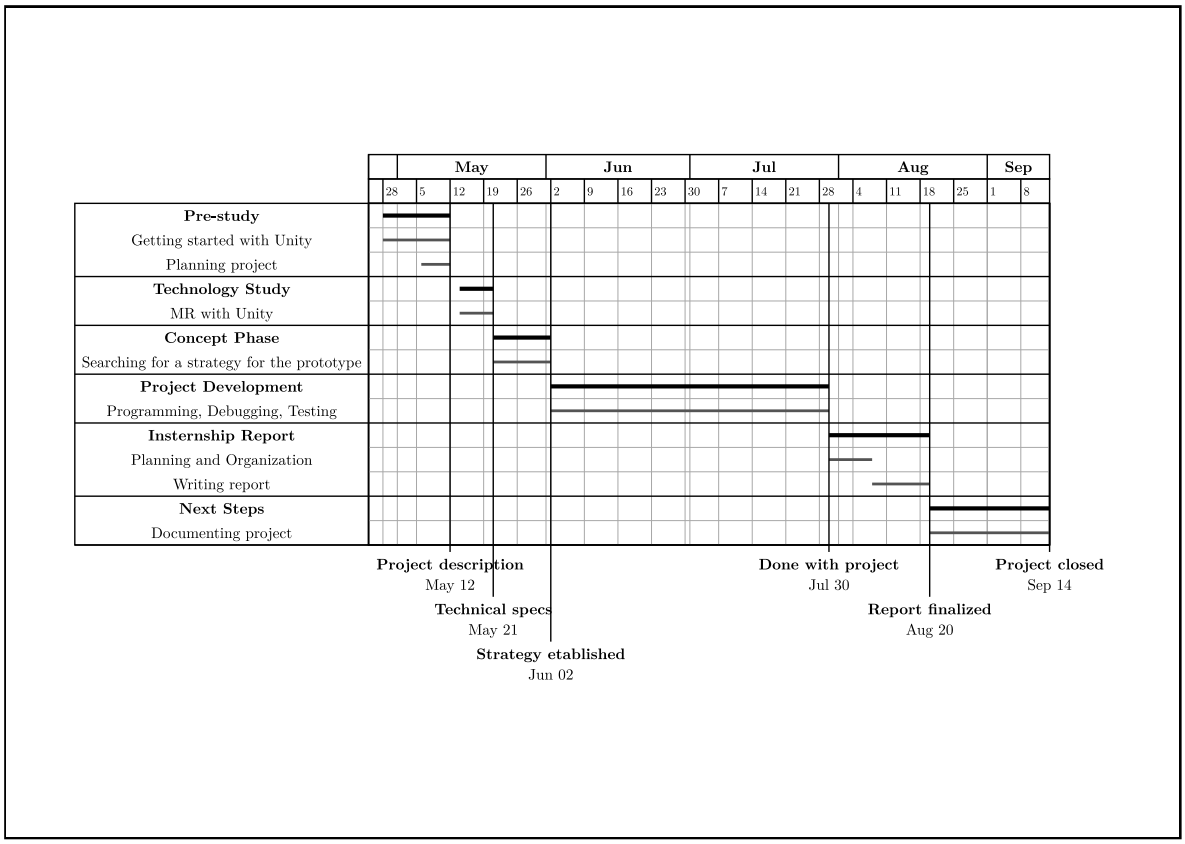

Figure 25: Actual Gantt Chart

At the beginning of the internship, I took some time to learn Unity so that I would not lose time later during the project development. The `Technology Study` section did not change much compared to what was planned. It included learning how to create a Unity MR project with the right packages, understanding the role of these packages, and how to use them.

For the `Concept Phase`, in fact, as our research went on, we discovered more and more limitations about the Meta Quest 3, as well as the level of development of the different libraries and packages, leading us to a situation where we had to do research and tests on how to make the project possible regarding the plane detections of the headset and also the recognition of doors, rooms, etc.

We can also see that there is no longer a `Design` phase. This is because the scope of the project was redefined when we realized it was much more ambitious than we first thought.

After that, the project development was done without a real organization: we programmed, tested, and debugged step by step, based on the ideas we had during the process.

Once we were satisfied with our project, we organized ourselves to write the report.

As feedback, I would say that this internship has been extremely valuable for me, especially regarding the development of an application with Unity. This report speaks for itself: before this internship, I had never used Unity for anything, and now I have a much better understanding of it.

Concerning how the internship went, overall it went well. Looking at the Gantt chart, it could appear that it lacked organization. In my opinion, this was partly due to the fact that we were given full autonomy during the internship, with only a few meetings to share our progress, and no clear guidelines provided at the beginning regarding the project topic. With less autonomy, we could have received more guidance, which would have allowed for a clearer definition of the different steps of the internship, even if only at a basic level.

We also faced some technical issues during the setup of the project, which took a lot of time to fix. Of course, this is part of the research process, but I feel it might have been better to receive some help during the very first setup steps. This would have saved time and provided better development conditions. Still, this is not a major issue, just a personal impression.

Finally, regarding the project topic itself, it might have been more appropriate to choose a subject with a difficulty level and scope better adapted to the students' experience and the internship duration. Our topic was chosen rather quickly, which may also explain why the scope had to be redefined later.

4.2 Next steps

During the development of the project, everything was done without progressive documentation, logs, or any real written trace of the process. In addition, the code and some of the naming conventions were not always well designed or clean. The idea now is to improve this by adding documentation to our project, which is publicly available on GitHub, so that any external person can better understand it — and maybe future interns at UiT who will continue our work.

This documentation will include context about the project, explanations of its current functioning, and guides on how to use and deploy the application in our Unity project. When possible, the project should also be improved in its current state by fixing inconsistent or unclear naming conventions, and by adding comments in the scripts to make them easier to understand.

Conclusion

In conclusion, this internship first allowed us to become familiar with the Unity game engine. We learned the basics, the vocabulary, and gained some practice in order to better prepare ourselves for the project. After this initial phase, we were able to propose an ambitious project: developing a mixed reality application with Unity to simulate an evacuation scenario, guiding a person towards the nearest emergency exit.

During the development of the project, we encountered several technical limitations with the tools available to us. One of the main issues was that the headset could not dynamically detect planes during the execution of the application. Instead, it relies on spatial data extracted beforehand. These spatial data are limited to a maximum of 15 rooms recorded on the headset, which created a major barrier to completing our project at the scale of the entire university building.

As a result, we explored alternative solutions. The most promising one was to create virtual floors and doors representing the real-world environment. This approach presented a challenge, especially since we did not have access to a digital model of the university building. The scope of the project was therefore redefined: our objective became to prototype a smaller-scale version using manually placed virtual elements.

This report specifically focuses on the work related to aligning the virtual elements with reality through several steps. We started by retrieving the floor of the room recorded in the headset, which we used as a spatial reference point for all the virtual elements in the Unity scene. Then, we implemented a method to create a virtual floor similar to the real one, allowing us to place virtual doors manually, using real-world measurements to ensure proper positioning relative to the virtual floor. Finally, we presented the last step, which consisted of adjusting the virtual elements based on the correspondence between the virtual floor and the floor imported from the headset.

For someone who had never used Unity before, this project was both ambitious and educational. Thanks to the collaboration with my teammate, we managed to create an application that allows the user to navigate through doors by following a path towards the nearest emergency exit. Even though this was achieved only at a small scale, it still represents an accomplishment we are proud of.

A | Glossary

1.a Hardware

Apple Vision Pro

The Apple Vision Pro is a mixed reality headset developed by Apple, designed to provide immersive augmented and virtual reality experiences with high-resolution displays and advanced sensors.

Meta Quest 3

The Meta Quest 3 is a virtual reality headset developed by Meta Platforms, designed to provide immersive experiences with advanced features like mixed reality capabilities and wireless controllers.

1.b Programming Terms

C#

C# is a modern, object-oriented programming language developed by Microsoft, widely used for developing applications on the .NET framework, including games with Unity.

Encapsulation

Encapsulation is a fundamental concept in object-oriented programming that restricts direct access to an object's data and methods, allowing interaction through a defined interface, enhancing security and modularity.

Object-Oriented Programming

Object-Oriented Programming is a programming paradigm based on the concept of 'objects', which can contain data and code, promoting organized and reusable code structures.

1.c Technical Terms

APK - Android Package

An APK (Android Package) is the file format used by the Android operating system for the distribution and installation of mobile applications.

AR - Augmented Reality

Augmented Reality (AR) is a technology that overlays digital information, such as images or sounds, onto the real world, enhancing the user's perception and interaction with their environment.

MR - Mixed Reality

Mixed Reality (MR) is a blend of physical and digital worlds, allowing real and virtual elements to coexist and interact in real-time, enhancing the user experience beyond traditional virtual reality.

Unity

Unity is a cross-platform game engine developed by Unity Technologies, widely used for creating both two-dimensional and three-dimensional video games and simulations for computers, consoles, and mobile devices.

Unreal Engine

Unreal Engine is a powerful game engine developed by Epic Games, known for its high-fidelity graphics and versatility in creating games and simulations across various platforms.

VR - Virtual Reality

Virtual Reality (VR) is a simulated experience that can be similar to or completely different from the real world, often involving the use of headsets and motion tracking to create an immersive environment.

XR - Extended Reality

Extended Reality (XR) is an umbrella term that encompasses all immersive technologies, including virtual reality (VR), augmented reality (AR), and mixed reality (MR), providing a spectrum of experiences that blend the physical and digital worlds.

B | Bibliography

[1] UiT The Arctic University of Norway, "About UiT The Arctic University of Norway." [Online]. Available: https://en.uit.no/om#engelsk

[2] Meta Platforms, Inc., "Meta Quest 3: Next-Gen Mixed Reality Headset." [Online]. Available: https://www.meta.com/quest/quest-3/

[3] M. Frąckiewicz, "Apple Vision Pro vs Meta Quest 3: In-Depth Comparison and Review 2025." [Online]. Available: https://ts2.tech/en/apple-vision-pro-vs-meta-quest-3-in-depth-comparison-and-review-2025/

[4] Unity Technologies, "Unity Manual." 2025. [Online]. Available: https://docs.unity3d.com/530/Documentation/Manual/

[5] LudicWorlds, [Online]. Available: https://www.youtube.com/@LudicWorlds

[6] Unity Technologies, "AR Foundation Manual (version 6.1)." 2025. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@6.1/manual

[7] Meta Platforms, Inc., "Mobile Device Setup for Meta Quest." [Online]. Available: https://developers.meta.com/horizon/documentation/native/android/mobile-device-setup/

[8] R. Boudrie and S. Abdul-Salam, "GitHub repository - Emergency Evacuation." [Online]. Available: https://github.com/SamiLumine/Emergency-Evacuation

[9] LudicWorlds, "Quest 3 Mixed Reality Course Assets." [Online]. Available: https://github.com/LudicWorlds/quest3-mr-course-assets