**Software design document**

**Proof of concept**

**PON cars**

**Author**: Stela Trencheva

**Student number**: 4520095

**Date**: 09.01.2023

Contents

[**1.** **Introduction** 3](#_Toc124421327)

[**2.** **Technical overview** 5](#_Toc124421328)

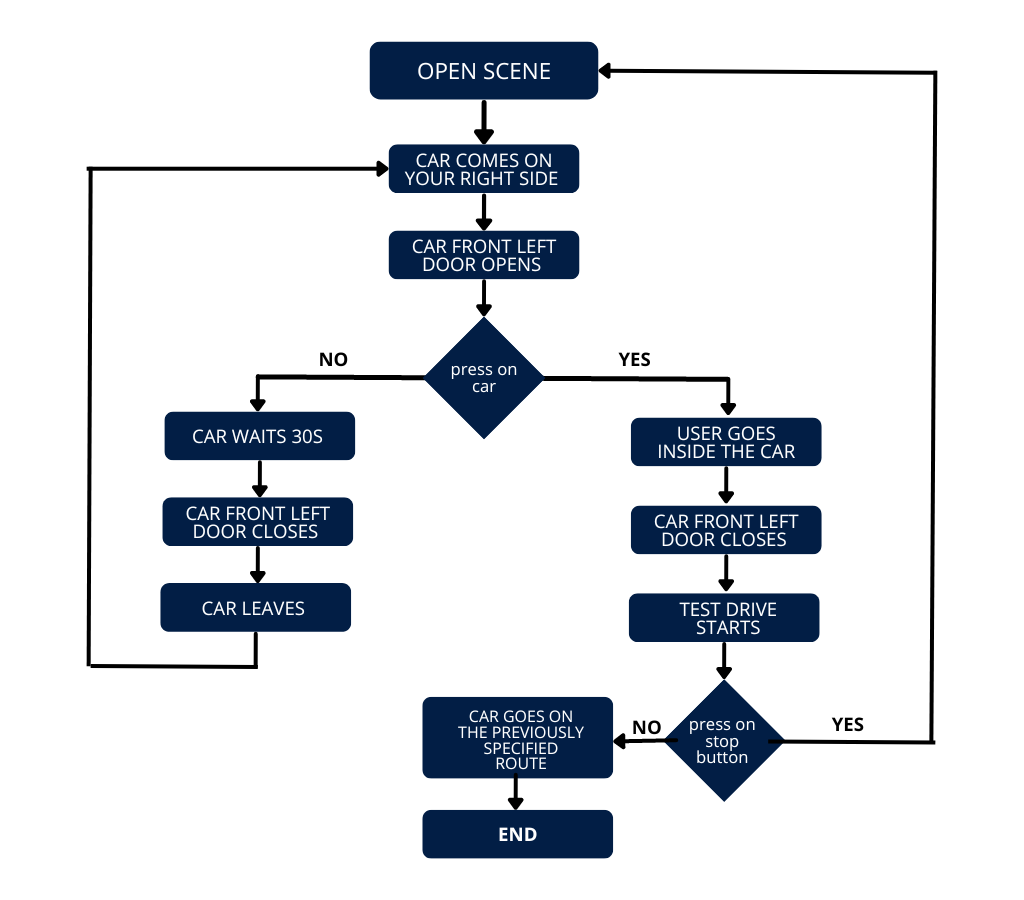
[**3.** **Code structure** 6](#_Toc124421329)

# **Introduction**

PON car is a VR application that acts as a proof of concept showcasing the idea of how takeover advertisement will look like in the metaverse in the near future. It is an application built with Unity 3D and working with Oculus Quest VR glasses. The application is branded after the car selling company PON.

A user is placed in a 3D futuristic environment and is facing a static menu that simulates the menu of an Oculus Quest 2 glasses. Everything around and behind the menu is a takeover advertisement. The background is a 3D futuristic city. The user is able to look around but not move. During the time that the user is in this environment, the takeover advertisement comes in action and offers an interaction to the user. It is up to the user whether they want to interact or not with the ad. The interaction in this case is a self-driven car that is moving on the road and comes on the right side of the user. The door of the car is opened and the timer of 10 seconds is started. During that time the user has the opportunity to press on the car. Then he will be transferred inside the car and the door will be closed. After that the self-driven car starts its journey on a previously set path and the user is enjoying a test-drive/ride of the car. Inside the car there is a screen on which the car is displayed from a different perspective. The user can have a look of the car from above on this screen. There is a stop button right under the screen in front of the user. By pressing on it, the user can stop the interaction with the advertisement and will be directly teleported to the starting position in front of the menu of the Oculus Quest. In case the user does not press on the car when its next to them and the door is opened, the timer of 10 seconds will expire, the door will be closed and the car will continue its journey without the user. Then the car will come again next to the user and they will have another opportunity to take part of the advertisement.

The users of the application are users of the Oculus Quest, companies / individuals that want to advertise and the Oculus Quest admins that are responsible for uploading and setting up the advertisements. In this POC only the role of the users of the Oculus Quest is tested and showcased. Due to the fact that this concept is at an early stage of development, these role was considered as the most important one and therefore that is why the scenario is building around it.

* **Process diagram**

# **Technical overview**

The application consists of front-end and back-end part.

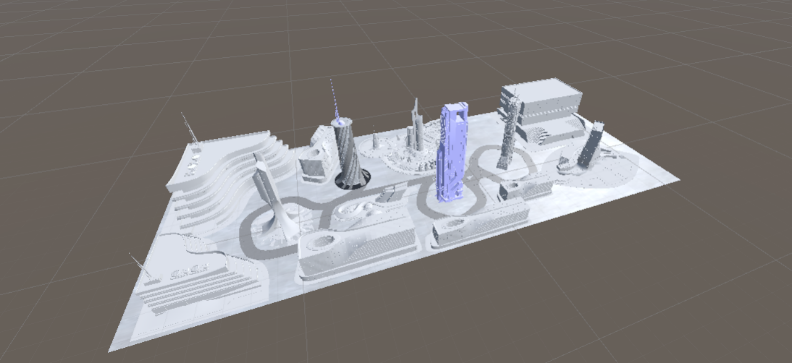
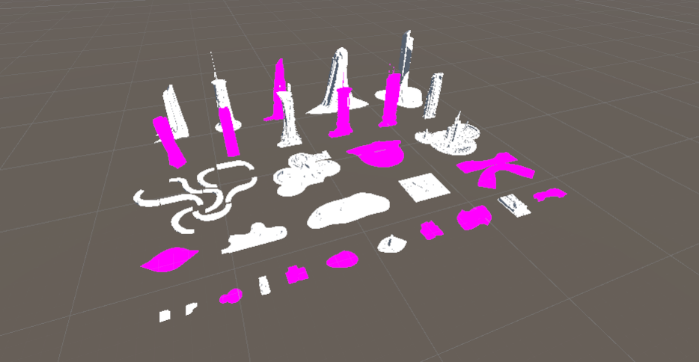
The frontend of Unity includes the Unity Editor, which was the primary tool used to create the application. The Unity Editor is a separate application that runs on top of the Unity engine and provided me with a wide range of features such as a visual scene editor, animation editor, game object inspector, and more. Furthermore, it also allowed me to extend its functionality by providing me a scripting API that I could use to automate common tasks and workflows.

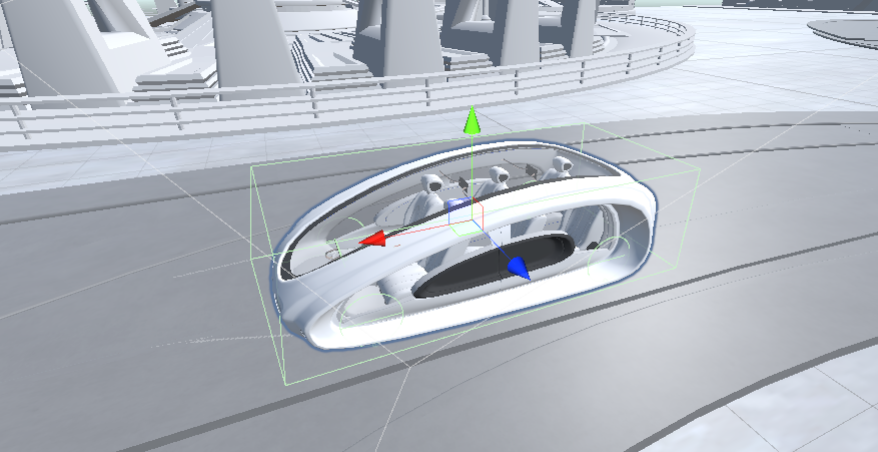
The frontend part of the project consists of all the 3D assets that are modelled together to form a futuristic city in which the user is placed. All of the assets in this project are downloaded from the Unity assets store due to the short time for development. The placement of the assets in the world/project in a good-looking way forms the frontend of the project.

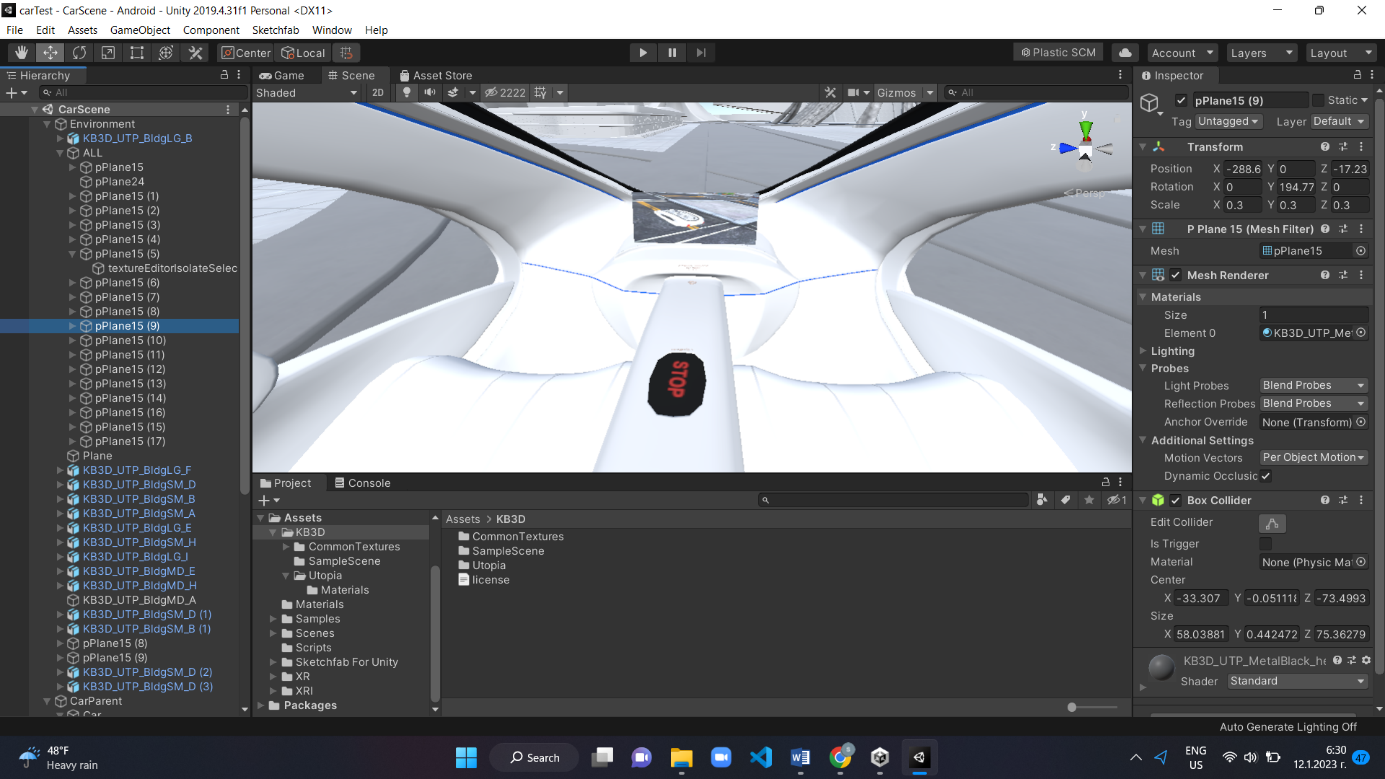
This backend was responsible for managing features such as rendering, physics, audio, and input. Additionally, it provided me with a set of APIs that allowed me to access and control these systems from my own C# scripts. Some classes were created with code that is executed when the program is started, others are attached to particular *GameObjects* to give them custom behaviour and are executed on a trigger.

The frontend and the backend are connected with the *UnityEngine* API and the *UnityEditor* API. *UnityEngine* API is the core Unity API that provides access to the engine's features and functionality. This API includes classes for managing things like physics, input, graphics, and audio, as well as classes for creating custom game objects and components. *UnityEditor* API provides access to the Unity Editor and its features. It allows developers to create custom tools and extensions for the editor, as well as to automate common tasks and workflows.

# **Code structure**

**Frontend –** The components in the frontend of this project are 3D assets downloaded from the Unity assets store. The city was created by modelling an environment and ordering the buildings, roads, etc. in a nice looking way in a 3D software – Blender. Then it was saved as *FBX* 3D object and imported into Unity. *Filmbox* (*FBX*) extension was used because it is more advanced for example in comparison with *OBJ* and can store a lot of information about the 3D object. It is recognised by a lot of 3D software and is used in game engine software. In Unity, basic materials coming with the downloaded package of buildings were added. They made the city look more like a draft, but due to the fact that the time was short and graphics were not a priority, it was not a problem to keep it like this for this POC.

A futuristic car was downloaded from a website for 3D objects – *Sketchfab* and imported into the scene. Some materials of the car were changed. For example, the windows were made transparent. In order to create the movement of the car (car AI) that is part of the backend, transparent spheres were added to the place where there were supposed to be the wheels of the car. Due to the fact that the car is autonomous, the spheres were aligned approximately. Also a *BoxCollider* was added around the car. The *BoxCollider* component allows objects to interact physically in a 3D space and serve as a collision detection shape. It is added to a *GameObject* to create a collision boundary in the shape of a box. When other objects with colliders come into contact with the object that has the *BoxCollider*, a collision will be detected.

Second camera was added to the screen and aligned to look from above the car. It was also made a child of the car so that when the movement is implemented, the camera is moving together with the car. Inside the car there is a screen. The view of the camera was added to the screen. Under the screen a “stop” button was placed which the user can use to go outside of the car. Furthermore, a transparent sphere was place inside the car which is used a placeholder of the user when they are transferred inside the car.

Directional Light was added to the screen. A *DirectionalLight* in Unity is a simulation of light rays that come from a single point and move in a specific direction. It is commonly used to create an effect of sunlight or light coming from a particular direction and illuminating everything in the environment.

Transparent empty *GameObjects* were added to different parts on the road. These objects together form the path that the car will follow.

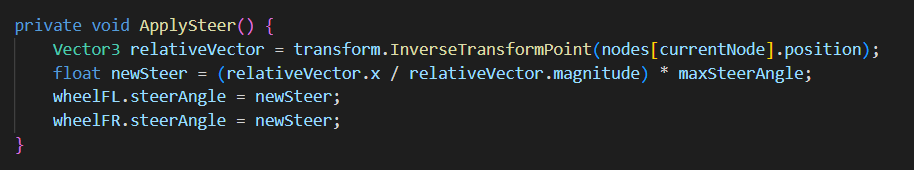
**Backend –** The backend of this project are all the actions and interactions. From the movement of the XR Rig and the ability of the user to look around the city to the movement of the car and the interactions of the user with the car.

The XR rig is a crucial component of this project. It serves as the user's virtual eyes, ears, and hands. There are two types of XR rigs: stationary and room-scale. For this project, since the user is not able to move around using teleportation or continuous movement, the stationary XR rig was chosen. It utilizes the user's physical space as a tracking position and enables them to experience a virtual environment that is the same size as their physical surroundings. As the user is not able to move around in the physical space, the stationary XR rig is suitable for this project. In this setup, the user's head and body remain stationary, while the virtual environment and objects move around them. A stationary XR rig is used for seated experiences such as flight simulators, racing simulators, and others where the user's physical movements are not a primary focus.

A UI interaction controller was used to manage the interactions between a user and a UI. It is responsible for processing user input, updating the UI to reflect the current state of the application, and responding to user interactions. The interaction controller receives input events such as VR controller press, and maps it to actions within the application. It also updates the UI to reflect the current state of the application, such as displaying new data or hiding or showing certain elements of the UI. In this case, it is used when the user presses on the car. The UI controller detects this press and links it to a method that is created to change the location of the user to the menu. Or when the user presses the “stop” button inside the car to call a method that change their location to in front of the menu. It is not a mandatory component but it is a good design practice to have one, as it separates the UI logic from the other parts of the application. It makes the code cleaner and more maintainable, and also make it easier to test.

The most complex part of the backend of this POC is the car AI or the functionality that makes the car move on a previously set path like a real one using real-world physics. The integration of Unity to develop AI for vehicles in a simulated setting is known as Car AI. It encompasses the implementation of autonomy in cars, the ability for them to take decisions, react to the surroundings and other cars.

Before the car starts moving, there is a script - *CreatePath.cs* that is executed and its purpose is to create the path itself. It has an empty list of nodes that is filled with the empty *GameObjects* from the road in the frontend when the program is started. Creating a path of nodes in Unity involves using the UnityEngine.AI namespace, which provides tools for creating navigation meshes and pathfinding. The path can be created using a *NavMesh* or *A\* algorithm* (a search algorithm that is used to find the shortest path between two nodes in a graph) with custom scripts. The first one *NavMesh* is easier for implementation and automatically generates a navigation mesh but since I already had programming experience there was no problem to use the *A\* algorithm* that gives more flexibility and better performance.

The rotation of the car to the according direction was implemented by applying steer to the transparent wheels’ colliders. In the *CarEngine.cs*, a method ApplySteer() was used. ApplySteer() is a car AI technology that enables autonomous vehicles to navigate safely and on the road. It uses a combination of sensor data, such as lidar, radar, and cameras, to perceive the environment around the vehicle, and machine learning algorithms to make decisions about how the vehicle should move. One of the key components of ApplySteer*()* is its ability to predict the behaviour of other road obstacles. This is done by analysing sensor data to identify and track other road users, and then using machine learning algorithms to predict their future actions. ApplySteer() also includes a number of safety features to ensure that the vehicle is always able to safely stop or avoid collisions if necessary. This includes a robust sensor fusion system that can detect and respond to unexpected obstacles, as well as a fail-safe braking system that can bring the vehicle to a stop if necessary. Due to the fact that the car does not have any visible wheels, the rotation of the wheels didn’t have to be smooth.

In order for the car to move autonomously it needs power of the engine. This was created by making a Drive() method in the *CarEngine.cs*. In order for the wheels to rotate a build in function *motorTorque* was applied to them. *MotorTorque* is a measure of the twisting force that a motor can generate. In a car, the *motorTorque* is used to turn the wheels and propel the vehicle forward. In order to make the car move like in real world, the *maxSpeed* was set at the beginning of the program and in the Drive() function, it is checked whether the current speed is less than the max speed. Only in this case, the *motorTorque* is applied to the wheels of the car so that it goes faster.

In order to navigate the car towards each of the nodes of the path a function CalculateDistanceToNextPoint() was created in the *CarEngine.cs*. This function checks what is the current node and which is the next one. It also calculates the distance to the next node and if this distance is pretty small the current node is updated to the next one. This goes in a loop and the car goes from one node to another. CalculateDistanceToNextPoint() function is called in the Update() function of the C# script.

After the car has reached the node that is next to the user, it has to stop. Break() function is called from the *CarEngine.cs* and there a *breakTorque* is applied to the wheels. Brake torque refers to the power generated by a car's braking system to decrease its movement. Through the integration of AI, the brake torque can be optimized for improved performance and safety. An AI system in the vehicle can gauge speed, weight and road conditions and adjust brake torque accordingly to guarantee the most efficient and safest stop. Next, a method called OpenDoor() from the *Car.cs* class is executed. This method first checks if the car has completely stopped by checking the value of the current speed. Then, it changes the position and rotation of the door so that it looks open. A timer of 10 seconds is started after the door is being opened. During that time the user can use the controllers to press on the car and if the car is pressed, a method called GoInsideCar() is executed. This method is changing the location of the user (XR Rig) to the location of the transparent sphere UserPositionInsideCar. After the timer has expired, a method CloseDoor() is bringing the door back in its initial position and the car continues on the way to the next node with or without the user in it. This repeats again and again until the program is stopped.