Università di Genova | **DIBRIS** DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

A Framework based on Large Language Models for Creating Customized Virtual Environments

by

Riccardo Caprile

DIBRIS, Università di Genova
Via Opera Pia, 13 16145 Genova, Italy
http://www.dibris.unige.it/

Master Thesis

UNIVERSITÀ
DEGLI STUDI
DI GENOVA

Laurea Magistrale in Computer Science
MSc in Computer Science
Data Science and Engineering Curriculum

A Framework for Creating Customized Virtual Environments

Riccardo Caprile

Advisor: Manuela Chessa

Examiner: Nicoletta Noceti

July 2024

Abstract

This thesis aims to exploit generative language models to create customized virtual reality environments.

In many fields of application, such as exergames for cognitive or physical training, one of the main challenges is to create a wide range of simulated scenarios in order to avoid the onset of boredom and habituation. However, creating many simulated situations can be a time-consuming and tedious task.

This thesis seeks to address this obstacle by creating a framework for generating different virtual reality scenarios described in natural language. The framework will give the possibility to unskilled individuals to add virtual scenarios and simulations to existing sw, such as cognitive exergames. We will analyse the use of generative language models (e.g. ChatGPT) and their application in creating virtual environments (e.g., in Unity).

Table of Contents

CHAPTER 1 INTRODUCTION 6

1.1 Motivations and Goals 6

1.2 Virtual Reality (VR) 7
  1.2.1 Applications 9

CHAPTER 2 LARGE LANGUAGE MODEL 12

2.1 Definition 12

2.2 Use Cases 13

2.3 GPT 14
  2.3.1 GPT-3 14
  2.3.2 GPT-4 15
  2.3.3 ChatGPT 16
  2.3.4 Prompt Engineering 16

2.4 LLM and VR 17

CHAPTER 3 METHODS 20

3.1 Software 20
  3.1.1 Unity 20
  3.1.2 API OpenAI 21
  3.1.3 Roslyn C# Runtime Compiler 26
  3.1.4 Meta XR SDKs 28
  3.1.5 Vocal Commands (Hugging Face API) 28

3.2 Hardware 29
  3.2.1 Meta Quest 2 29
  3.2.2 Workstations 31

CHAPTER 4 DESIGN OF THE SYSTEM 33

4.1 Introduction Scene and “How to” Scene 33

4.2 Developer Mode 35

4.3 User Mode 39

4.4 Environments 42
4.4.1 Pre-Built Environments 42
4.4.2 Free Models 44

4.5 Log File Generation 46

CHAPTER 5 IMPLEMENTATION 50

5.1 Chat.cs 50
5.1.1 Start() 50
5.1.2 ReadStringInput() – Prebuilt Environment 51
5.1.3 ReadStringInput() – Free Models 52
5.1.4 Meta Language 56

5.2 Domain.cs 57

5.3 Vocal_Commands.cs 58

5.4 Teleport.cs and Change_Scene.cs 59

5.5 Graph_Maker.ipynb 59

CHAPTER 6 EXPERIMENTS AND RESULTS 61

6.1 System Testing 61
6.1.1 Input Formation 61
6.1.2 Extra checks for the acceptability of the C# script 66
6.1.3 Script Generation: Execution Time and Error Rate 67

6.2 User Testing 72
6.2.1 Testers Description 72
6.2.2 System Usability Scale (SUS) 72
6.2.3 User Experience Questionnaire (UEQ) 72
6.2.4 Questionnaires Results 73

CHAPTER 7 CONCLUSIONS AND FUTURE WORK 77

BIBLIOGRAPHY 80

Chapter 1 Introduction

The first chapter provides a brief summary of the topics covered in the subsequent chapters, outlines the principal objectives of this thesis, and presents an introduction to Virtual Reality (VR).

1.1 Motivations and Goals

Large Language Models (LLMs) trained for code generation can be used to transform a natural language request for a virtual world into reality. The main objective of this thesis is to utilize LLMs to generate custom virtual environments tailored to user requests. In recent years, AI models have achieved remarkable levels of reliability when responding to inquiries. Therefore, we intend to exploit these models and provide users with a framework that can overcome the problem of creating different environments for various situations with minimal effort. Our goal is to enable the construction of environments using natural language, making the framework accessible to individuals outside the realm of computer science. The possibilities for custom-designed VRs are virtually limitless. For instance, a physiotherapist could request a virtual environment that is tuned to the needs of a patient with an unusual injury. Students in a laboratory could inspect, analyse, and become familiar with materials or machines that they might not have access to because of budget constraints, safety concerns, or physical limitations. Furthermore, without the support of a framework like this, building the environment would be a boring and tedious task that must be repeated hundreds of times. Firstly, we want to define and explain Large Language Models and what is the relationship between them and Virtual Reality as the framework is based on them. Then the combination of these tools will produce the desired result. Without LLMs we would not have the outputs requested for creating the customized world.

Subsequently, we will describe the hardware and, especially, the software required for the realization of the framework, providing the technical aspects of those methods. Then, we will analyse the design of the system, detailing how the previously described tools are combined and how the user can effectively utilize the framework.

At this point, we will delve deeper into the design and the implementation of the system, analysing all the C# scripts and all the building blocks that compose the framework.

In the last part of the thesis, we will present the results obtained from the framework. The “users’ analysis” will be based on their feedback through a couple of questionnaires (the System Usability Scale and the User Experience Questionnaire). In addition to this, a technical analysis will be conducted to evaluate the system’s performance in different scenarios (e.g. Error rate and execution time).

To conclude this project, we will offer some suggestions on how the framework could be improved in future works.

1.2 Virtual Reality (VR)

Virtual Reality (VR) is a technology which allows a user to interact with a computer-simulated environment that can be a representation of the real world or an imaginary world. The main objective is to provide an immersive experience of a virtual setting. The environment is typically accessed via a display, usually a wearable display (e.g. a Head mounted display (HMD), gloves or body suits).
The illusion of "being there" is simulated by motion sensors that pick up the user's movements and adjust the view on the screen accordingly in real time.

VR can be divided into 3 categories:

- **Non-Immersive:** This refers to a virtual experience through a computer where you can control some characters within the software, but the environment is not directly interacting with you. Basically, you are dealing with a virtual world, but you are not at the centre of attention in the game.
  All basic forms of gaming devices such as: PlayStation, Xbox, Computers, etc, provide us with a non-immersive virtual reality experience.

- **Fully Immersive:** This type of experience is what most people envision when they come across Virtual Reality. It relies on wearable displays that track the users' movements and present VR information depending on the position of the users**,** allowing for a 360-degree experience of the virtual environment. It creates the sensation of being physically present within the virtual world physically.

- **Semi-Immersive:** As the name suggests, this is a mix of non-immersive and immersive virtual reality. It involves a 3D space or virtual environment where you can navigate on your own, either through a headset or a computer screen. All the activities are centred around you, but you have no real physical movements other than your visual experience.

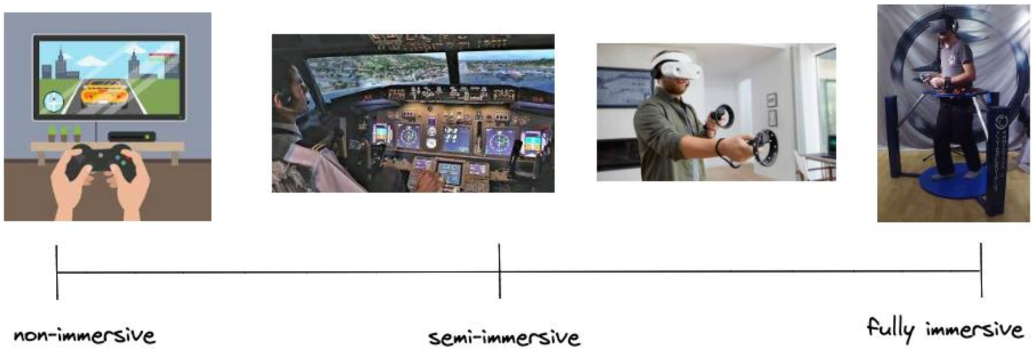


Figure 1: The three categories of Virtual Reality

In addition to the immersive and non-immersive virtual reality, previously discussed, there is also an extra category:

- **Augmented Reality:** It combines the real world and computer-generated content. It is a technology that enhances the real-world environment with digital information and objects, creating a more immersive and interactive experience. Users can access digital information in a more engaging and interactive way.
  An example of augmented reality are interactive displays; for instance, when we scan an object in a shop, we can visualize some extra information about that article through the screen of the smartphone.

One of the first appearances of Virtual Reality devices similar to Head Mounted displays (HMD) was in the first half of the 1800s with the stereoscope. It was built based on research that demonstrated that the brain combines two photographs of the same reference object taken from different points in order to make an image appear to have a sense of depth and immersion.
In 1956, cinematographer Morton Heilig created Sensorama [1] the first VR machine. It combined full colour 3D video, audio, vibrations, smell, and atmospheric effects.
His second invention was the so called Telesphere Mask which was the first HMD. It provided stereoscopic 3D images with wide vision and stereo sound.
Following this creation, Ivan Sutherland in 1965 presented his concept of the "Ultimate Display". Basically, his idea was based on a virtual world viewed through an HMD which replicated the reality so the user would not be able to distinguish between the real world and the virtual world.

"The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be good enough to sit in. Handcuffs displayed in such a room would be confining, and a bullet displayed in such a room would be fatal." [2]

In 1968, Sutherland actually created the first virtual reality HMD, called "The Sword of Damocles". It was connected to a computer and could display only simple virtual wireframe shapes. These 3D models changed perspective when the user moved his head thanks to the tracking system.

In more recent years, in 1991, NASA scientist Antonio Medina designed a VR system aimed at driving a Mars robot from Earth in real-time. This system was called "Computer Simulation Teleoperation."
Following this development, companies like SEGA and Nintendo started to create Virtual Reality headsets for the general public.
In 2012, this revolution exploded thanks to a Kickstarter campaign for the Oculus Rift raising a huge amount of money.
In 2014, with the acquisition of Oculus by Facebook, many companies started developing their own Headsets.
Sony announced that they were working on the Project Morpheus for PlayStation 4.
Google released the "Cardboard" – a low-cost stereoscopic viewer designed for smartphones.
Similarly, Samsung announced the "Samsung Gear VR", a headset that uses the smartphone Samsung Galaxy as a viewer.
By 2016 hundred of companies started developing VR products: HTC, Google, Apple, Amazon, Sony, Samsung, Microsoft.
In conclusion, Oculus has demonstrated that Virtual Reality has incredibly progressed. It is now used in a wide range of applications, from gaming experiences to teaching new skills and providing brand new virtual journeys.

1.2.1 Applications

So now, we would like to introduce some of the applications of Virtual Reality.

Education

The first compelling application for Virtual Reality is within the educational sector. While its current usage may not be extensive, there are many promising examples and studies that demonstrate how beneficial VR could be in education.
Firstly, the usage of VR has the capacity to enhance students' attention to activities within the VR environment, thereby maintaining their motivation and focus.
Because of this technology, they are extremely more interested and engaged in what they are learning at school, mitigating distractions commonly encountered in traditional classroom settings.

Moreover, VR also has the potential to transport students to different environments, allowing them to explore new cities or places, which are otherwise inaccessible or impractical for human visitation.

Medical Training

One of the most significant and impactful applications of Virtual Reality is its utilization in enabling professional doctors to practice operations on “virtual” patients. This advancement has resulted in improved outcomes and reduced the number of mistakes during medical procedures. With better preparation, the success rate of surgeries is notably higher.
Technically speaking, students can train in an interactive virtual environment that can be programmed with different scenarios and parameters. In this way, they have the opportunity to explore different real-life settings and test their skills.
The simulation could incorporate instructional videos demonstrating how the operation should be carried out or how a tool should be used. A lot of different information could be displayed during the test.
Furthermore, it is possible to program patient-specific environments which can simulate hundreds of possible combinations of training, ultimately contributing to better-prepared medical professionals and improved patient care.

Exergaming, Fitness and Sports

Virtual Reality can also be used for practising sports and maintaining physical fitness. This has been made possible thanks to a genre of games called exergames, where players participate in physical activities to reach goals within the game. The main objective of this type of exercises is to counteract the usual sedentary activity that is at the base of traditional games by incorporating body movement as input, rather than solely relying on pressing buttons on a keyboard, mouse, or controller.
VR technology can also be used in sports when athletes need to improve their skills or facilitate recovery from an injury with the help of therapy and rehabilitation programs.

Therapy

One interesting usage of Virtual Reality is therapy, particularly for mental health purposes. VR can be used by people who suffer from post-traumatic stress disorder, as a means to confront traumatic events but in a safe and controlled environment

under the guidance of a therapist. By creating the traumatic event in a virtual setting, patients can receive help in processing this event.
In addition to this, VR can serve as a valuable tool when a patient needs a safe and tranquil place to alleviate stress and anxiety. Through VR individuals can create an environment where they can destress and detach themselves from the external world, providing a therapeutic escape and promoting emotional well-being.

Entertainment

The last application is the most well-known and widely used ~~one~~ among the general public. Nowadays, Virtual Reality is primarily used ~~only~~ for entertainment purposes such as watching movies and TV shows as well as playing simple video games just for fun or trying different experiences that go beyond the everyday traditional computer and monitors.

In conclusion, we can say that Virtual Reality has an infinite number of applications and uses, if we consider the vast range of environments that can be created with the help of this technology. This is exactly what we have done: provide users with the possibility of creating customized virtual environments in a simple way.

Chapter 2 Large Language Model

Because the framework implemented ~~it~~ is based on a large language model such as ChatGPT, developed by OpenAI, we want to provide information about these models in this chapter and their connection with the Virtual Reality development world.

2.1 Definition

A large language model (LLM) is a deep learning algorithm that can perform multiple Natural Language Processing (NLP) tasks, such as: text generation, code generation and information retrieval.

LLMs and Generative IA are obviously linked to each other, because LLMs are a type of generative AI that has been built for the generation of text-based content.

Basically, LLMs are designed to understand and generate text, like a human being. They base their knowledge on the vast amount of data available used for their training. They have the ability to infer from context and generate coherent and relevant responses comprehensible to humans.

These kinds of models are based on a transformer architecture, which is particularly well-suited for handling sequential data such as text input.

The transformer architecture was presented to the public in 2017 in a research paper by Google called “Attention is All You Need” [3].

Transformers are neural networks that learn and understand context, through sequential data analysis, and generate new data based on it. They use a technique called “Attention” which enables them to identify how distant data elements influence and depend on each another.

Transformers took inspiration from the encoder-decoder architecture. The encoder takes the input and generates a matrix representation of it, whereas the decoder takes as the input the output generated by the encoder earlier and it will output the desired answer.

This architecture allows transformers to effectively capture and utilize contextual information, making them highly effective for tasks involving sequential data, such as language understanding and generation.

One of the key strengths of this algorithm is that it does not perform data processing in sequential order, but it parallelizes all the operations, enabling faster data processing.

Now, we know a little bit more what is behind an LLM, but how do they work?
As we said before, LLMs are based on a transformer model and they function by receiving an input, encoding it with an encoder, and, in the end, decoding it to produce the desired output. But before all these operations, we need to *train* them, so they can fulfil general functions, and *fine-tune* them so that they are able to perform more specific operations.

Training: During this step, models are exposed to a wide amount of unlabelled textual datasets taken from websites such as: Wikipedia, GitHub, or others.
The models must learn patterns, structures, the meaning of the words, relationships networks and semantic knowledge present in the text corpus. At this point the datasets are processed without specific instructions, unsupervised learning.

Fine-Tuning: A language model needs to be trained to perform a specific task or domain.
We train the LLM on a labelled dataset (Supervised Learning) for the specific task or domain wanted.
Thanks to Fine-Tuning, the performance of the LLM can be improved for the specific task or domain by adjusting the weights of the model to better fit the data.
In this case, the labelled data consists of pairs of input and output data.

2.2 Use Cases

Large language models can be used for several reason**s:**

- Information Retrieval: Whenever you use a search engine and their feature, you are relying on a large language model to provide information in response to a query written by the user.

- Sentiment Analysis: LLMs enable users to analyse the sentiment of textual data.

- Text Generation: LLMs are able to generate text based on inputs. They can answer a question through a text-based response.

- **Code Generation**: This is very similar to Text Generation. For example, in our framework, The LLM will generate a C# script, according to our requests, which will be executed and will create the virtual environment as specified by the user.

- Chatbots and conversational IA: Large language models enable customer service chatbots to have a conversation with customers, interpret the meaning of their queries or responses, and offer them responses to clarify their doubts.

Furthermore, large language models can be found in a huge number of fields. In Healthcare and Science, they assist in understanding proteins, molecules and DNA or are a helpful tool in the development of vaccines. In Marketing they employ sentiment analysis to generate targeted marketing campaigns. In the Legal sector they translate documents written in basic language to more complex and legalese language.

2.3 GPT

2.3.1 GPT-3

GPT stands for Generative Pre-Trained Transformer 3 which is a large language model released by OpenAI [4] in 2020.
It belongs to a family of AI models.
The latest GPT model is GPT-4 which belongs to the fourth generation, although various versions of GPT-3 are still widely used and available. These models are the ones we will use in the testing phase of our virtual environment experiments.
There are many models in the GPT-3 family, some of which are employed for different purposes. In the published presentation paper [5], a table, Table 1, was presented describing 8 different GPT-3 models and their sizes:

Model Name	Parameters	Number of layers	API Name
GPT-3 Small	125M	12	n/a
GPT-3 Medium	350M	24	ada
GPT-3 Large	760M	24	n/a
GPT-3 XL	1.3B	24	babbage
GPT-3 2.7B	2.7B	32	n/a
GPT-3 6.7B	6.7B	32	curie
GPT-3 13B	13.0B	40	n/a
GPT-3 175 or GPT-3	175.0B	96	davinci

Table 1: GPT-3 Family

The parameters in GPT-3, like any neural network, consist of the weight and the biases of the layers. The sizes are determined by the number of layers. Therefore, versions with more layers have a higher number of parameters.

Weights are the most important parameters through which the model learns by exploiting the training data. For example, if a model sees the word “Dog” followed by the word “bark”, it will assign a higher weight to this pair of words. Next time, the model will be able to predict more likely the word “bark” after “dog”.
Bias is a parameter used as an adjusting factor to Bias is a parameter used as an adjusting factor to fine-tune the predictions of an entire layer, making them more accurate. Basically, it is employed to adjust the output values and influences how easily a node can be activated.

Most of the models mentioned above are accessible through APIs. Later, we will explore which models are available and their respective purposes.

2.3.2 GPT-4

With the technical report [9], OpenAI launched on March 14, 2023, GPT-4. A multimodal large language model, successor of GPT-3 and GPT-3.5. As stated in the presentation paper the main objective of this new version is to “improve the ability of such models to understand and generate natural language text, particularly more complex in nuanced scenarios”.
The model is trained on a large amount of multimodal data, including text and images from different sources and datasets. After training, the model is aligned with manually labelled datasets containing verifiable facts and desired behaviours.

There are many differences between GPT-4 and its predecessor.
The first one is the number of parameters used for the training step; as we have seen previously, GPT-3 has 175 billion parameters, on the other hand GPT4 is estimated to have 1.76 trillion, a noticeable difference.
The second difference between them is that GPT-3 accepts as input only textual input while GPT-4 accepts obviously textual input, but also images. For example, it has the ability to recreate websites just by providing a basic mock-up. A second example ~~it~~ is explained in the paper [9] where a set of 3 images is given as input to the model and asked to determine what is funny about them. The model was able to clearly describe what those images were representing and was able to answer correctly to the questions asked.
Another difference is that GPT-4 has been trained with information up to 2023, while GPT-3 ~~it~~ has information up to 2022, as a result that information is outdated, and it might not be 100% precise.
Lastly, the output quality of GPT-4 is more accurate, precise and advanced compared to the previous models.

2.3.3 ChatGPT

ChatGPT [6] is a chatbot developed and released by OpenAI in November 2022. As the name suggests, it is based on the large language model GPT, previously described.
It gives the user the possibility to have a conversation with a desired length, style and language.
ChatGPT is available for everybody in two versions, one built with GPT-3.5 and the other with GPT-4.
This chatbot was trained on a massive corpus of text data, around 570GB of datasets, including software manual pages, web pages, books, and other kind of sources.
The main objective of ChatGPT is to engage in humanlike conversations, trying to answer users' questions as ~~much~~ precisely as possible.

2.3.4 Prompt Engineering

Prompt engineering is an artificial intelligence engineering technique that comprehends the process of refining LLMs with specific prompts and recommended output, and, at the same time, the process of refining input to get the best output from an LLM.
Prompt engineering is fundamental for creating better AI-powered services and getting better results from AI generative tools.
For example, we will face this problem in 6.1.1 where we will see what kind of sentences have been used as input and the expected and unexpected results we have obtained.

There are four types of prompts:

- Zero-shot: It is the simplest and most direct type of prompt. A direct instruction or question without adding extra information about is given to the generative IA. This type of prompt technique tests the model's ability to produce relevant output without relying on prior examples (e.g. "List me some Computer Science projects that I can develop")

- One-shot: In this case, a single example guides the model's output. This example can be a question-answer pair or a specific template. This method has the objective to align the model's output more closely with the user's desired format of response.

- Few-shot: This technique, similar to One-shot, involves providing the generative IA with some examples to guide it to what you would like it to imitate. This helps the models to understand the user's requirements better, generating output that closely follow the same pattern of the given examples.

- Chain-of-Thought (CoT): This method is designed to guide the AI through a logical reasoning process. The idea is not just asking for an output, but it is a strategy that incorporates demonstrations that tell the model how to arrive at the correct answer step by step. Basically, it encourages the large language model to explain its reasoning. With this kind of prompt structure, we get the solution and understand how the AI reasoned, by producing a detailed output.

- Zero-shot Chain-of-Thought (CoT): It is a technique similar to Zero-shot prompting previously explained, but this approach adds an instruction at the end of the request: "Let's think step by step".
  The model will be able to generate a chain of thoughts from this simple instruction, and of course the result will be more accurate as well.

In addition to this, there are some ideas for refining prompts whenever you get stuck, and most of them were used during the conversations we had with ChatGPT when a certain script had been required.
For example, it is recommended to repeat key words and use caps to stress important points or specific details desired. Another idea is to use the verb "must" frequently**.**
Try to write long requests repeating the same statement many times but in different places in the sentence.

2.4 LLM and VR

In this subchapter, we would like to talk about the relationship between large language models and virtual reality nowadays.
It **is** possible to find just two papers [7][8] about this topic, meaning that this relationship has not been explored and exploited so much at the moment.
It is useful to delve into the first paper, as it provided inspiration on how we could overcome some challenges we had to face during the design process of the system.
When the implementation of our framework was completed, we had access to a second paper [8], that will be analysed later in this chapter.

The Microsoft developers created a VR game with non-deterministic game mechanics, called Codex VR Pong, powered by OpenAI's text generative models which were integrated with Unity game engine.
This is a sort of Ping Pong game where the players can transform both the paddles and the ball into any 3D objects and these transformed objects interact in semantically

sensible ways; for instance, a ball transformed into an egg colliding with a pan result in a fried egg.
To implement Codex VR Pong, the developers used an integration of Codex for Unity, which is a descendant of GPT-3. Codex's training data contains both natural language and billions of lines of source code from publicly available sources.

The most important part of this scientific document is the method used to overcome the issue that C# code needs to be compiled at runtime, while Unity does not have a built-in compiler to evaluate code at runtime. The solution to this problem is an open-source compiler called Roslyn, which can be downloaded from the Unity Asset Store. We will discuss this in detail in section 3.1.3.

When the implementation of our application was completed, paper [8] was released and it is interesting to analyse this in this subchapter, as we explore the integration of large language models with Virtual Reality."
The developers' objective was to create a system based on Unity that bridges the gap between static content and dynamic behaviour generation in VR environments by using LLMs for the generation of run-time code.
At the moment with Unity, the only possibility for executing generated code at runtime is through the C# Compiler Roslyn [12].
They aimed to exploit LLMs because they possess the potential to enhance the productivity of software developers by automating various task such as implementing new features or translating natural language program descriptions into an actual program. On the other hand, we focused on non-skilled users who are not developers, with the objective to eliminate boring and redundant tasks just for creating an environment.
Regarding the implementation, they decided to use in addition to Unity, Ubiq [13], a library for research with networking features including message passing, room management and matchmaking. Their objective was similar to ours, but they took another path for reaching their goal.

Of course there are differences with our framework. We will now see some of them.
Firstly, the main objectives of their research. They exploited large language models for the code generation of objects' behaviour. They required the user to ask ~~to~~ the AI to create a certain behaviour (e.g. make a circular movement) for a selected object already inside the Unity scene. Instead, we wanted to let the user ~~to~~ create his own environment and, in a "game modality", positioning models wherever he pleased.
Secondly, they decided to use only basic Unity assets such as cubes, spheres and cones. Our aim was to offer users a "real world" experience by providing them with objects that can be found in the real world, ensuring a faithful representation, and giving them a VR experience that simulates what they see every day.

Like us, they had to test the LLM in order to obtain the best code structure and style to get an acceptable and with no compilation errors script.

They made a code generation comparison with different large language models analysing the accuracy and the time needed in order to receive an answer from the AI. In the end, they analysed the type of errors that the models can make with the generated output and how to deal with them.

All of these problems and all of the testing phases will be sorted out and analysed in the Chapter 6

LLMs in VR, basically, demonstrate that they are a fundamental and interesting tool when we want to develop an application that can be changed drastically at runtime, considering the code structure that generate**s** the game world, by providing an interface with which the user can interact with and create completely new environments, worlds, behaviours and an infinite number of objects and models, in the same session.
LLMs in VR basically demonstrate that they are a fundamental and intriguing tool for developing applications that can undergo drastic changes at runtime. They accomplish this by altering the code structure that generates the game world, by providing an interface with which the user can interact and create entirely new environments, worlds, behaviours, and an infinite number of objects and models, in the same session.

To conclude this chapter, we would like to spend few words about a framework mentioned in the paper [12], Ubiq-Genie [14]. This framework enables you to build server-assisted and collaborative mixed reality applications with Unity using the Ubiq framework [13]. In the paper [15] every aspect of this framework is explained but the most important aspects of it, are the services provided starting from Speech-to-text, moving on with Text-to-speech, Image Synthesis (generating images based on text input), Text-Synthesis (generating text that simulates a conversation based on text-based prompts).
We did not use this framework, because we wanted to create a useful application from scratch and exploited the OpenAI API as much as possible (see chapter 3.1.2).

Chapter 3 Methods

In this chapter, we will describe all the software and hardware tools used during the implementation and the testing of the framework. For each of them, we will give an idea of their working principles plus some technical aspects, specifications and examples.

3.1 Software

3.1.1 Unity

Unity [35] is a game engine developed by Unity Technologies and released in 2005. It is a cross-platform solution available for Windows, macOS and Linux; in addition to them, it supports the development of a large set of other platforms for which it is possible to build applications like iOS, Android and Virtual Reality platforms.
Nowadays, its main field of application is the video games development; the most famous video games developed with Unity are: *Pokemon Go*, *Call of Duty: Mobile* and *Cuphead*. It has also been used in the cinematographic industry by Disney to create backgrounds for the 2019 film *The Lion King*.

Unity is generally contrasted with another game engine used worldwide which is Unreal Engine developed by Epic Games. The capabilities of both engines are extremely similar, so the choice of one over the other depends only on the developer's preferences. Both systems are written in C++, but Unity default scripting language is C#. Unity has a powerful scripting API that offers a quick access to the most needed features such as: general game features or specific features for the engine. For instance, you can access and modify the position, rotation, materials etc.., of a particular object in the scene through the API provided.
Since our framework is VR-based, we required a game engine that supported VR and AR technologies, and Unity was the perfect fit. For VR, there are numerous packages available that support all VR headsets currently in use, and they are constantly updated. We used the XR SDKs package 3.1.4."
A strength of Unity is the great amount of documentation, provided by Unity Technologies, to help you out whenever there is a problem with a component of the game engine.
As most of the game engines in the market, even Unity has its own Asset Store with thousands paid or free graphical assets, specific game genre templates, audio, textures, animations. Thanks to this we could have downloaded different type of 3D models and textures for our environments. The most important developed package

downloaded from the store was the Roslyn C# Runtime Compiler API, which will be analysed in detail later in this master thesis.

In addition to all these features, Unity offers more. For example, several built-in options for the render pipelines; this allows the developer to choose the best render pipeline that best fits his project, because rendering graphics to the screen is a fundamental step for having good performances in a game.
To sum up this brief introduction, Unity serves as a beginner-friendly game engine because it is built with an architecture which is very easy to understand and appreciate. Moreover, it offers a robust set of tools that supports any kind of game or project. But the most important aspect for this project is that there are few engines that support VR like Unity does. Unity supports this technology in every possible way. Of course it will require improvements in the future, but the amount of support, tools, and integration with the engine make it the best game engine to choose if you want to start a VR-based project.

3.1.2 API OpenAI

OpenAI API provides programmatic access to the OpenAI models and services. What we need for our purposes is a package that integrates OpenAI API inside Unity, and we have found a package called OpenAI-DotNet [16] realised for C#, the programming language used inside Unity scripting which is easily installable through the Unity package manager through the option *add a package from a git URL.*
It must be said that this not an official unity package but with this package it is possible to easily communicate with OpenAI models. OpenAI is aware of this package and other packages for other environments and because of this, it is possible to find it in the Community Libraries section of the OpenAI official website.
To activate the package, an OpenAI key is required, which is possible to create only if you have an official OpenAI account.
This package offers multiple functions and features. For example, you can list the various models available in the API, also checkout model endpoint compatibility to understand which models work with which endpoints. Then, you can retrieve information about a certain model such as: the owner and permissions.
The most important feature, for us, which this package offers, is the Chat feature. Basically, as ChatGPT, if you provide an input to the model, you receive a text output that answers the question you previously asked.
What we used is called Chat Completions. Chat models take a list of messages (in our case, the list will contain just one message) as input and return a model-generated message as output. Although the chat format is designed to make multi-turn conversations easy, it is also useful for single-turn tasks without any conversations, and that’s what we are aiming for, because we just need a simple answer, and the conversation can be closed immediately.

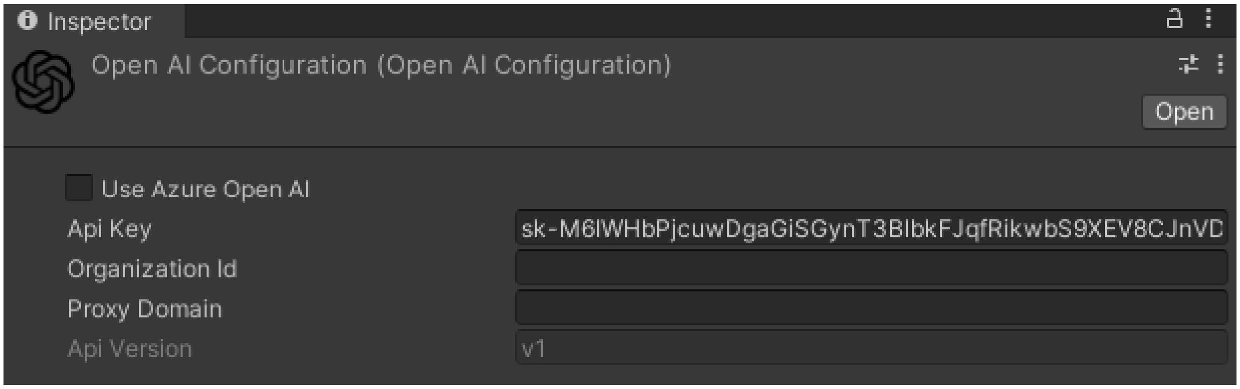
We would like, now, to let you visualize a short code snippet about how we used this API inside our project:

```
1 var messages = new List<Message>
2
3  {
4    new Message(Role.User, input)
5  };
6
7 var api = new OpenAIClient();
8 var chatRequest = new ChatRequest(messages, Model.GPT3_5_Turbo);
9 result = await api.ChatEndpoint.GetCompletionAsync(chatRequest);
```

Code Snippet 1: How the OpenAI API is used inside the framework

This is al~~l~~ the code needed to establish a communication with the model GPT3_5_Turbo (we will see which models can be accessible through this API in 3.1.2.1).
In line 1 and 4, we define a simple list of messages (in this case there is only one message). In the Message we have to specify the role and the content of the message which is the string variable *input.*
Then in lines 8 and 9, the real request is sent to the selected models and inside the result variable, we wait for an answer to our question, and now the final result will be saved as a string.
In line 7, we have the typical usage with an internal HttpClient. We have already uploaded all the Client information in the OpenAI Configuration asset.

It should be noted that at the beginning of the implementation, we used another function instead of *GetCompletionAsync(),* called *CreateCompletionAsync().* This function when given a prompt, let the model return one or more predicted completions. This function was deprecated in January 2024, and so we had to change what function to use for our purposes.



☐ Use Azure Open AI
Api Key: sk-M6lWHbPjcuwDgaGiSGynT3BlbkFJqfRikwbS9XEV8CJnVD
Organization Id:
Proxy Domain:
Api Version: v1

Screenshot 1: OpenAI API Configuration Asset

Getting back to the OpenAI API Configuration asset, Screenshot 1, is provided in the exact moment the package is installed. It is useful if you want to save all the client information and use them whenever is possible without writing repetitive lines of code in every project that includes this API.
The fundamental parameter is the *Api Key*, generated by OpenAI in your account set up panel.
Then, there is the *Organization Id*, used for specifying an organization.
API that makes requests to OpenAI on behalf of your front-end application.
The last parameter is the *Proxy Domain*. Exposing your API keys and other sensitive information is a risk. To mitigate this risk, it is possible to set up an intermediate API that makes requests to OpenAI on behalf of your front-end application."

3.1.2.1 Models

The OpenAI API is powered by a wide range of models with different capabilities and, especially, prices.
They are, firstly, divided into a sub-group depending on the task they carry out; then every sub-group contains different type of models depending on the Context window and how updated they are.
The table below lists all the models and their corresponding tasks**.**

Model Name Groups	**Description**
GPT-4	A set of models that improve GPT-3.5 and can understand generated natural language or code.
GPT-3.5 Turbo	A set of models that improve GPT-3.5 and can understand as well as generate natural language or code
DALL-E	It is a model that can generate and edit images with a given natural language prompt
TTS	It stands for “Text to Speech” and can convert text into a natural sounding spoken audio
Whisper	A model that can convert audio into text
Embeddings	A set of models that can convert text into a numerical form
Moderation	A fine-tuned model that can detect whether a text may be sensitive or dangerous
GPT base	A set of models without instruction that can understand as well as generate natural language or code

Table 2: Set of models inside the OpenAI API

There is also a list of deprecated models along with the suggested replacement. During the implementation of the framework, we were using a model belonging to the GPT base models, called *Davinci*, that was deprecated by OpenAI on the 4th of January 2024. Because of this, we had to change the selected models from the previous one to GPT-4 groups and GPT-3.5 Turbo groups.

We are interested in three sets of models: GPT-4, GPT-3.5 Turbo and GPT base. These groups are available for the testing phase of our framework in the OpenAI API for Unity.
Every model obviously has a price to pay whenever it is used. The price depends on the different capabilities of each model and the number of tokens written by the user. A model that is more powerful and advanced than an older one, will be more expensive, and this is a detail that must not be underestimated.
In the table below, we present the models available in the API with name, description, execution prices (the combination of the prompt sent to OpenAI, and the response given by OpenAI) and the Context Window (the maximum number of tokens that the model can consider at any one time when it generates a response):

Specific Model Name	Description	Prices for 1 Execution	Context Window
GPT-4	Large multimodal model that can solve different problems accurately thanks to its broader general knowledge and reasoning capabilities	0.00045$ for 10 tokens 4.5$ for 1M tokens	8192 tokens
GPT-4-32k	It is an extended version of GPT-4, with the same capabilities but quadrupled the Context Window, allowing for processing up longer input.	0.00090$ for 10 tokens 9$ for 1M tokens	32768 tokens
GPT-3.5-Turbo	It can understand and generate natural language or code and has been optimized for chat. It is also suitable for non-chat work tasks as well.	0.00001$ for 10 tokens 1$ for 1M tokens	16385 tokens

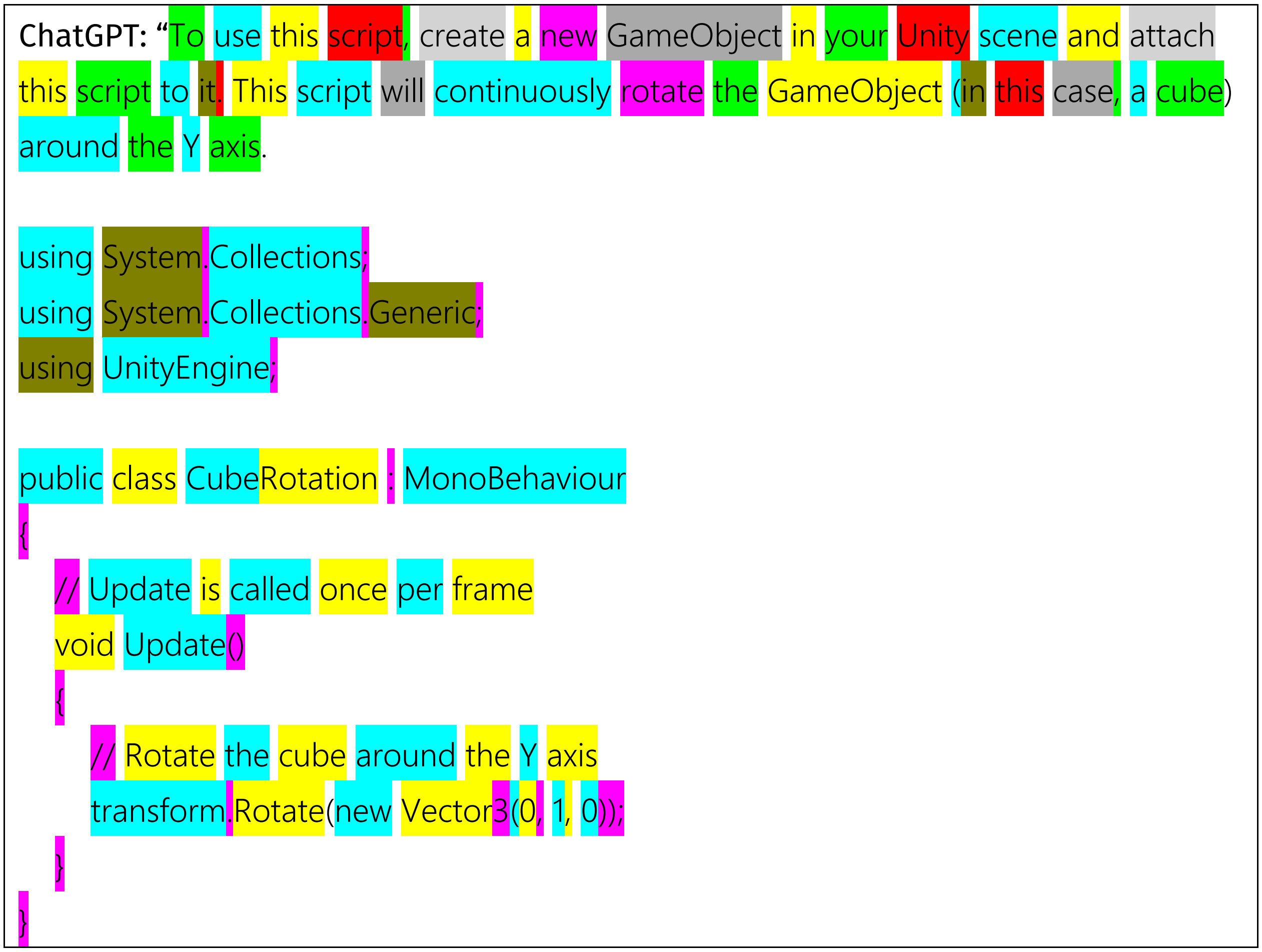
GPT-3.5-Turbo-16k	It has the same capabilities and standards of GPT-3.5-Turbo, and the same Context Window. But it has been trained with a larger quantity of data, and can generate more accurate output	0.00003$ for 10 tokens 3$ for 1M tokens	16385 tokens
Text-Davinci-002	GPT base models that can understand and generate natural language and code, but they are not trained with instruction following. It is better to use GPT-3.5 or GPT-4.	0.00002$ for 10 tokens 2$ for 1M tokens	16384 tokens
Text.Babbage-002		0.000004$ for 10 tokens. 0.40$ for 1M tokens	

Table 3: Models available in the OpenAI API

3.1.2.2 Tokens

In previous sentences, we spoke about tokens, but what are these *tokens?*
They are the key metric of OpenAI request, and they represent a part of a word. Short and simple words, usually, are formed by one token. On the other hand, longer and more complex words could be formed by two or three tokens or more.
OpenAI bases all the price ranges on this metric.
When you write something, you have to consider that the pricing metrics are considering the Input Tokens (all the words you write in the console) and the Output Tokens (the words generated by the model).
It is important to underline that the exact tokenization process varies between models.
Just to be clear, here is a useful and easy example, where input and output are subdivided in tokens already:

Prompt : “Hi, create a Unity C# program”



ChatGPT: “To use this script, create a new GameObject in your Unity scene and attach this script to it. This script will continuously rotate the GameObject (in this case, a cube) around the Y axis.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CubeRotation : MonoBehaviour
{
    // Update is called once per frame
    void Update()
    {
        // Rotate the cube around the Y axis
        transform.Rotate(new Vector3(0, 1, 0));
    }
}
```

Code Snippet 1: Example of code divided into tokens.

In the example above, we wanted to show how OpenAI separates input and output into tokens by using a tool provided by OpenAI [17]. In the input, there 27 characters which, in this case, correspond to 8 tokens , whereas in the output there are 477 characters that are divided into 103 tokens.
One token should correspond to ~4 characters of text for common English text, roughly 100 tokens comprehend 75 words.
For completion, a price prediction for this operation, using GPT-3.5 Turbo, could be:

$$(103 + 8 \text{ tokens}) * 0.00001\$ = 0.00111\$$$

The price is small, of course, but if we increase the complexity of the question the answer will be longer and so the price will be higher.

3.1.3 Roslyn C# Runtime Compiler

Roslyn C# [12] is a compiler that allows runtime loading of assemblies and C# scripts. It was released on August 11th, 2023, at the price of 20$ on the Unity Asset Store.

It was a tool we were exactly looking for. Our purpose is to compile C# scripts while the application is running in order to build environments whenever a script is provided by the AI.
We will now see the building blocks that compose Roslyn and an example about how we have used the tool inside our application.
Roslyn is based on the definition of three different concepts: Script Domain, Script Type and Script Proxy.

- Script Domain: It is considered as a container for any externally loaded code. You must create a Script Domain before you can load any external code and all the subsequent loading will be handled by that domain.
  Other than being a code container, the Script Domain is considered responsible for the loading and the compiling of C# code; in addition to this, it ensures that any loaded code does not use dangerous assemblies or namespace.

- Script Type: It acts like a wrapper class for “System.Type” and has a number of specific methods that make it easier to manage external code. The principal advantage of the Script Type class is that it provides a number of methods for type specific construction meaning that the type will always be created using the correct method. For example, types which inherit from MonoBehaviour, the Script Type will require a GameObject to be passed and will use the *AddComponent()* method to create an instance of the type.

- Script Proxy: It is used to represent an instance of a Script Type that has been created using one of the *CreateInstance()* methods.

We would like to show an example of how Roslyn has been used inside our project in order to clarify those three concepts that compose this runtime compiler.

```
1         domain = ScriptDomain.CreateDomain("Example Domain");
2
3
4
5         ScriptType type = domain.CompileAndLoadMainSource(sourceCode,
6                           ScriptSecurityMode.UseSettings);
7
8
9         ScriptProxy proxy = type.CreateInstance(gameObject);
10
11
12        proxy.SafeCall(sourceCode)
```

Code Snippet 2: Roslyn Compiler usage example.

The first thing to do (line 1 of Code Snippet 2) is to create a Script Domain where all our external code will be loaded into. Everything is done by the method *CreateDomain()* and the only argument provided to this method is just the Domain’s name.

Now that we have created a domain, we can compile and load any external C# code (lines 4 and 5) using the method *CompileAndLoadMainSource()*; as an argument , we have passed the string *sourcecode* , which contains the code generated by the LLM, that must be executed at runtime.
This method will invoke the Roslyn compiler which will generate a managed assembly (the name of the class of the external code). The main type is automatically selected and returned as a Script Type.
With the Script Type loaded, the next step is to create an instance of that type (as shown in line 9). The easiest way to do this is to use the basic method *CreateInstance*(). The argument is a GameObject because the new class inherits from Monobehaviour.
All of the expected Monobehaviour events will be called "Start" and "Update".
The script is now constructed but we need to execute it. So, the last step, line 12 , is the call for actually executing the external code. The method used is called *SafeCall()* that will catch any exceptions thrown by the target method, and as an argument we provide it with the code that must be executed.
These are the easy step for the right usage of Roslyn C# Runtime Compiler.

3.1.4 Meta XR SDKs

When you want to develop a Virtual Reality application in Unity, you need an SDK that provides you with functionalities for Extended Reality(XR) applications built with virtual reality or mixed reality components.
For the Meta Quest devices, there is the Meta XR SDKs available inside the Unity Package Manager. [18]
The SDK gives you the ability to create immersive applications, including advanced rendering, social and community building.
There are many packages of the Meta XR SDK, but we have used two of them: Core SDK and Interaction SDK.
The first one contains many essential functionalities which Meta Quest 2 headset provides such as hand tracking, spatial anchors, scene management whereas Interaction is the package that has allowed us to make possible interactions between the scene and the controllers.
With all of these packages we could walk around the environment, interact with all the XR UI Canvas and buttons, teleport ourselves from a point of the world to another one and visualize in detail every model inside the custom environment.

3.1.5 Vocal Commands (Hugging Face API)

In the framework, there is the possibility for the user to use vocal commands to ask which environment he desires or which object he wants inside his customized world.

We have implemented the vocal commands thank to the API : Hugging Face API for Unity [19].
Hugging Face API allows developers to access and use Hugging Face API models within the Unity projects and provides a good number of AI services such as: conversation, text generation, text to image, text classification, summarization, translation; but we are interested in speech recognition. By implementing this feature, the user can speak through the headset's microphone and ask for the environment he wishes.

During the testing phase, vocal commands did not understand the user's request on the first try; most of the time, they required more than one attempt to accurately transcribe what the user was saying. Of course, users can use their own keyboard to write the desired customized environment, which is the suggested mode of use.

The actual implementation of the vocal commands will be demonstrated in section 5.3.

3.2 Hardware

3.2.1 Meta Quest 2

A fundamental step during the development of a Virtual Reality application is the choice of the head mounted display that will be used. An HMD (head mounted display) is a wearable display device, that has a small display optic in front of one (monocular HMD) or each eye (binocular HMD). In addition to this categorization, HMD can be divided into two more groups *video see-through* (VST) and optical *see through (OST)*, basing these categories on how the images are presented to the user and the way the device can be tracked in space.

VST devices mount one or more cameras which capture what is in front of the user and show it on the screen in front of his eyes; of course, you don't see what's around you directly, but you see a video of it. With this type of technology, you can try more immersive experiences by mixing real-world objects and virtual elements. Developers have the possibility to change the aspects of the real world such as where the user is located, changing textures, lighting, removing or adding virtual objects. The cons of using this kind of HMD include potential delays between the real world and its display, and the field of view is limited by the camera, which may not accurately simulate the human one.

The second category is OST. Imagine wearing a special pair of glasses that allows you to see through them directly, like a usual pair of sunglasses. If you wear these glasses, you can visualize additional information or images on top of what you are already **see.** Basically, you are watching the real world but with the addition of more interesting and useful information. You can have a more natural interaction with the physical environment because all the new information is projected onto the real world through the display with less latency.

In conclusion, both OST and VST technologies offer unique advantages, the choice of one category over the other just depends on what kind of application the developer has to create.

For our project, we have used the Meta Quest 2, a VST headset, developed by Reality Labs (a division of Meta Platforms [10]) and released on October 13, 2020. The display of the headset is an LCD panel with a resolution of 1832x1920 per eye and a refresh rate of 120 Hz. It uses the processor, Snapdragon XR2, designed for VR and augmented reality devices.

Included with this device, there are two touch controllers used for navigating inside the virtual world as "real hands".

The headset accurately tracks the user's head and body movement without the need of external devices thanks to four cameras distributed around the headset and offers hands tracking features for navigating menus without the touch of controllers.

With this headset there is no necessity to wear headphones because there are built-in speakers for left and right positional audio. But the user can still use his own headphones via the 3.5 mm audio port. This device includes a built-in microphone that will be very useful for our application's vocals commands.

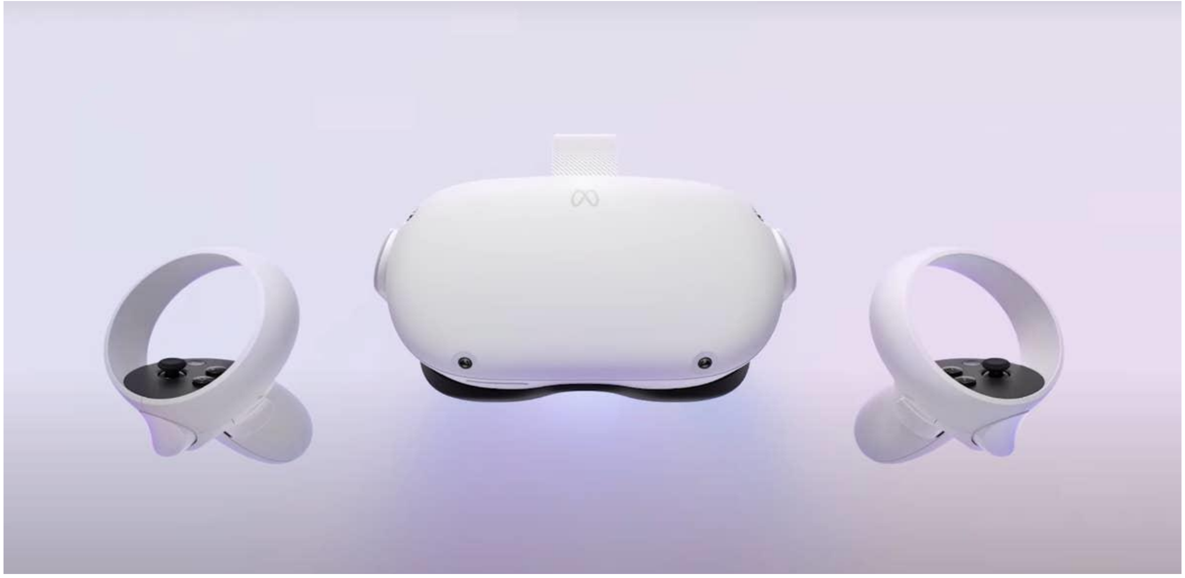


Figure 2: Meta Quest 2 with controllers.

When you turn on your Meta Quest 2, you will jump into the Meta Hub where you can access different settings and activities. We are interested in the *Link* connection that will give us the possibility to connect the headset to our computer.

The Meta Quest Link is a piece of software that Meta has integrated with the Meta Quest 2 and allows you to connect your Meta Quest to your PC.
If the connection is granted you will be redirected to the Oculus Rift Dashboard where you could also visualize the screen of your PC and interact with it by wearing the headset.

If you want to use the Oculus Link, you first need to check if your PC is compatible with Meta Quest Link. The requirements are a processor that is an Intel i5-4590/AMD Ryzen 5 1500X or better, memory which is at least 8GB minimum, a Windows 10 Operating System and a GPU from a long list of compatible GPUs [11]. If your PC is compatible with the previous prerequisites you have to follow some steps:

1. Download the Meta PC software, which must be activated every time you want to use the *Link*.
2. Plug a compatible Meta Quest Link cable to your PC (Meta offers the possibility to use the Link wirelessly, but we have only used the headset through the cable connection, because it is more stable and has less latency)
3. A message in the headset's display will pop up, if accepted the link will be completed.

This connection has been fundamental for us in order to launch Unity projects and test them on the headsets.

3.2.2 Workstations

The entire project has been developed on a machine with the following hardware components:

- CPU: Intel Core i9-9900k 8 core/16 threads
- GPU: NVIDIA GeForce RTX 2070 SUPER
- RAM: 32GB DDR4
- Operative System: Microsoft Windows 11 Home

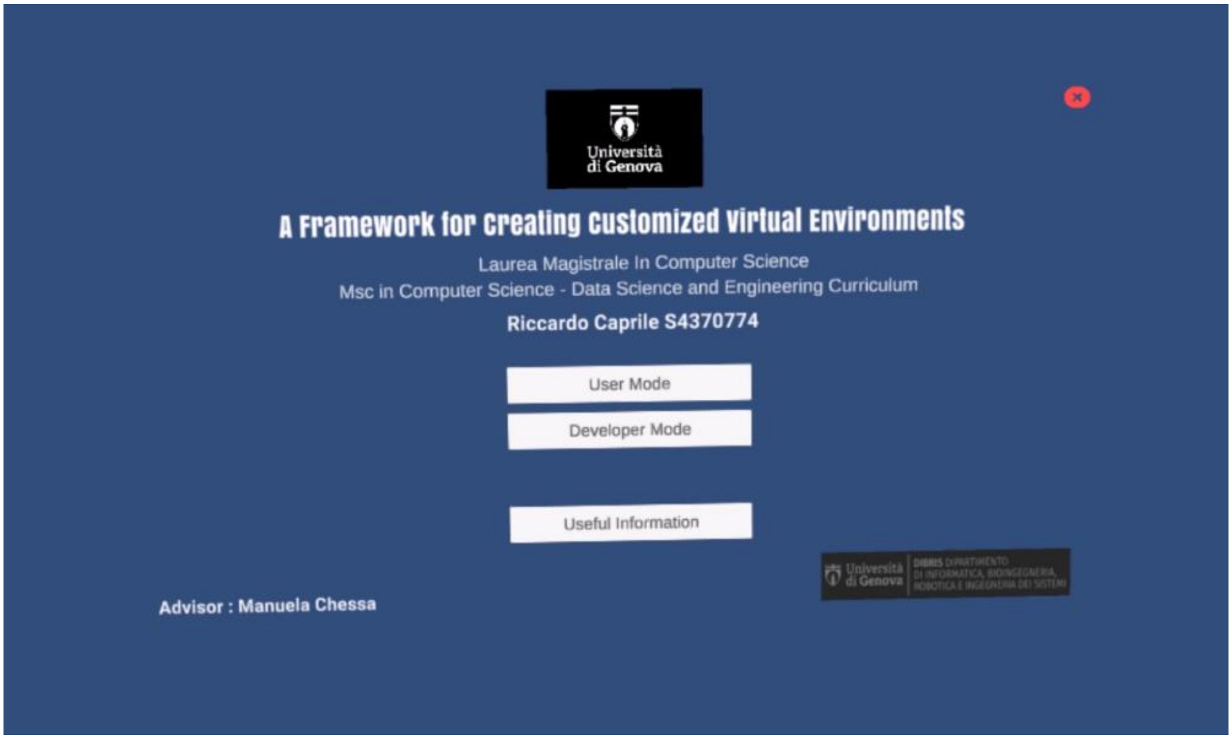
The User Testing has been conducted on a different machine, an ASUS Zenbook Pro Duo UX582, with the following hardware specifications:

- CPU: Intel Core i9-10980HK 8 core/16 threads
- GPU: NVIDIA GeForce RTX 3070
- RAM: 32GB DDR4
- Operative System: Microsoft Windows 11 Pro

Chapter 4 Design of the System

In this chapter we would like to present how the system is designed, with all the Unity scenes, and how we can interact with it through different UI Canvas and buttons. Firstly, we will introduce all the scenes of the framework and then how the user can exploit the application correctly.

4.1 Introduction Scene and "How to" Scene



A Framework for Creating Customized Virtual Environments

Laurea Magistrale In Computer Science

Msc in Computer Science - Data Science and Engineering Curriculum

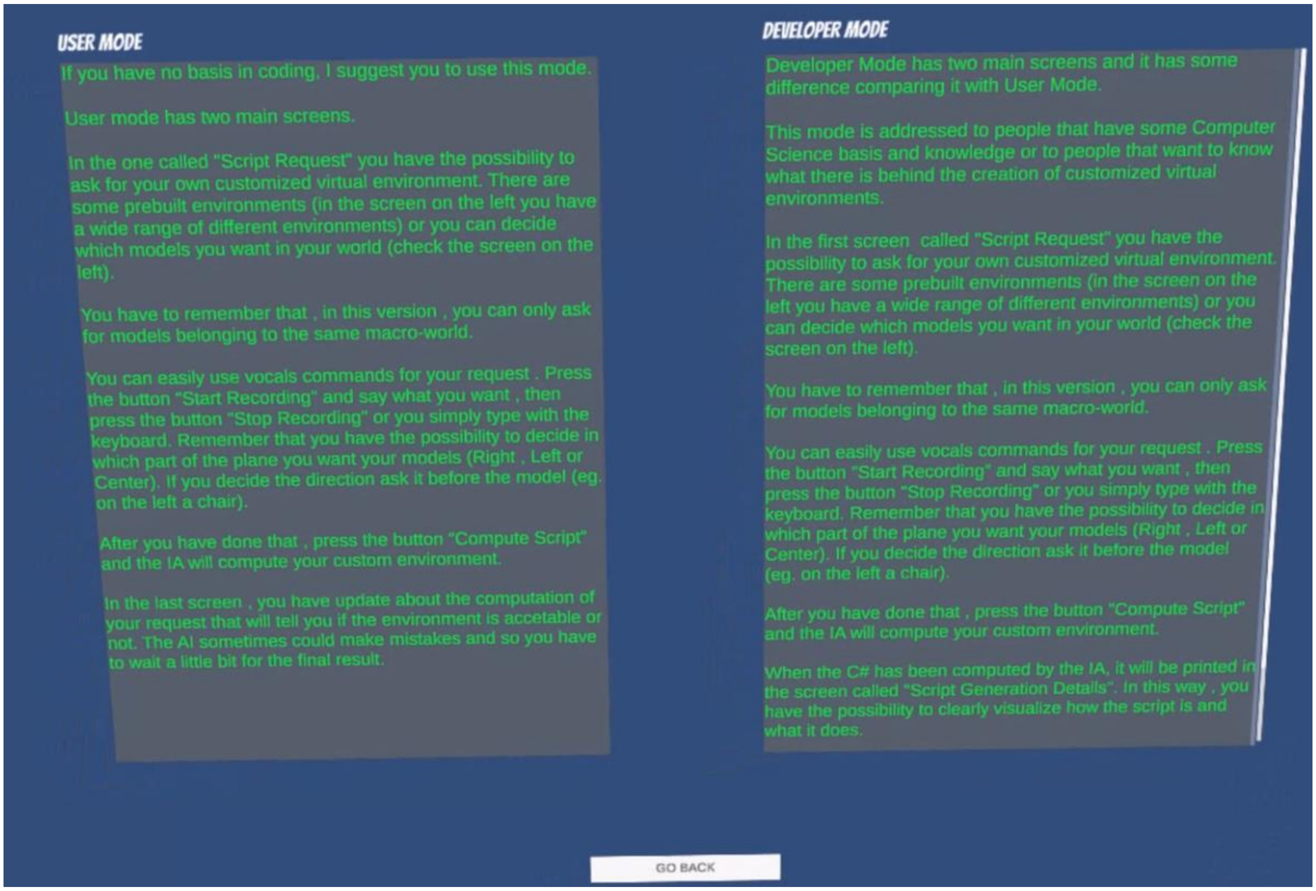
Riccardo Caprile S4370774

Advisor : Manuela Chessa

Screenshot 2: Introduction Scene Screenshot

The first scene that the user visualizes is the "Introduction Scene" (Screenshot 2 shows how the scene appears with the headset worn). It shows the title of the framework "A framework for creating customized virtual environments" which gives the user a clue about what he can do with this program as well as the creators of the framework. In addition to this, there are 3 different buttons "User Mode" 4.3, "Developer Mode" 4.2 and "Useful Information". At the top right, there is a button that will let the user quit the application and go back to the desktop.

The buttons “User Mode” and “Developer Mode” will redirect you to the specifics scenes where you can build you own world, while “Useful information” will bring the user to a scene where there is a general description of the two modalities of the framework (e.g. how you can interact with the GUI).



USER MODE

If you have no basis in coding, I suggest you to use this mode.

User mode has two main screens.

In the one called "Script Request" you have the possibility to ask for your own customized virtual environment. There are some prebuilt environments (in the screen on the left you have a wide range of different environments) or you can decide which models you want in your world (check the screen on the left).

You have to remember that , in this version , you can only ask for models belonging to the same macro-world.

You can easily use vocals commands for your request . Press the button "Start Recording" and say what you want , then press the button "Stop Recording" or you simply type with the keyboard. Remember that you have the possibility to decide in which part of the plane you want your models (Right , Left or Center). If you decide the direction ask it before the model (eg. on the left a chair).

After you have done that , press the button "Compute Script" and the IA will compute your custom environment.

In the last screen , you have update about the computation of your request that will tell you if the environment is accetable or not. The AI sometimes could make mistakes and so you have to wait a little bit for the final result.

DEVELOPER MODE

Developer Mode has two main screens and it has some difference comparing it with User Mode.

This mode is addressed to people that have some Computer Science basis and knowledge or to people that want to know what there is behind the creation of customized virtual environments.

In the first screen called "Script Request" you have the possibility to ask for your own customized virtual environment. There are some prebuilt environments (in the screen on the left you have a wide range of different environments) or you can decide which models you want in your world (check the screen on the left).

You have to remember that , in this version , you can only ask for models belonging to the same macro-world.

You can easily use vocals commands for your request . Press the button "Start Recording" and say what you want , then press the button "Stop Recording" or you simply type with the keyboard. Remember that you have the possibility to decide in which part of the plane you want your models (Right , Left or Center). If you decide the direction ask it before the model (eg. on the left a chair).

After you have done that , press the button "Compute Script" and the IA will compute your custom environment.

When the C# has been computed by the IA, it will be printed in the screen called "Script Generation Details". In this way , you have the possibility to clearly visualize how the script is and what it does.

Screenshot 3: Useful Information Scene Screenshot

As you can see in the screenshot above, there is a brief description for both modalities that will explain **to** the user how he can use the framework in the right way with the help of usage examples.

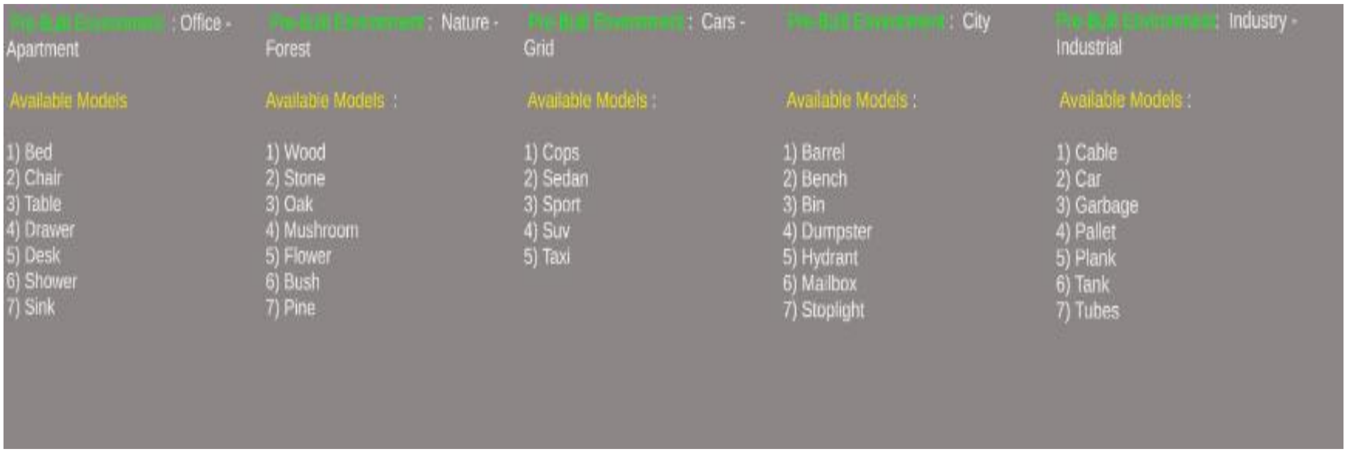
The “Go Back” button, at the bottom of the screen will redirect the user to the “Introduction Scene”.

4.2 Developer Mode

This is the first of the two modalities available inside the framework, and it has been built for the user familiar with the computer science world. It includes some technical aspects which may not be comprehensible to every kind of user.

As we have said in the chapter presentation, there are Unity Canvas and Buttons that must be explained in order to understand how the framework is supposed to be used, and this is what we will do in this subchapter concerning "Developer Mode" modality. Thus, we will analyse all the windows step by step and, in the end, we will show a complete usage example.

The first window that the user sees is the list of models and pre-built environments implemented inside the framework.



Pre-Built Environment : Office - Apartment

Available Models

1) Bed
2) Chair
3) Table
4) Drawer
5) Desk
6) Shower
7) Sink

Pre-Built Environment : Nature - Forest

Available Models :

1) Wood
2) Stone
3) Oak
4) Mushroom
5) Flower
6) Bush
7) Pine

Pre-Built Environment : Cars - Grid

Available Models :

1) Cops
2) Sedan
3) Sport
4) Suv
5) Taxi

Pre-Built Environment : City

Available Models :

1) Barrel
2) Bench
3) Bin
4) Dumpster
5) Hydrant
6) Mailbox
7) Stoplight

Pre-Built Environment: Industry - Industrial

Available Models :

1) Cable
2) Car
3) Garbage
4) Pallet
5) Plank
6) Tank
7) Tubes

Screenshot 4: List of environments and models available.

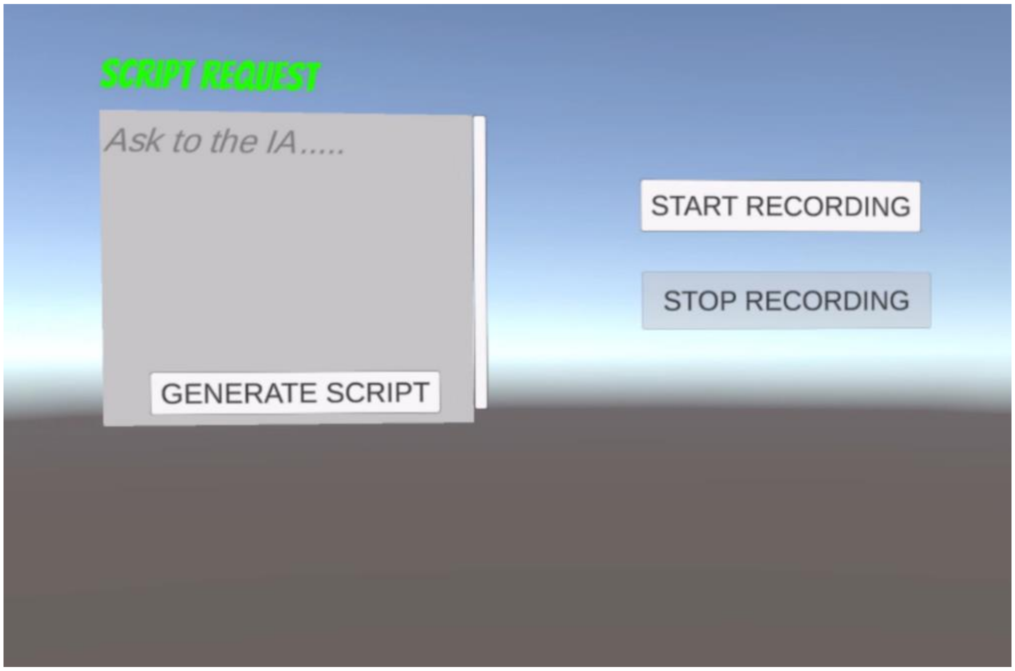
In this way, the user can clearly visualize which keywords the system will recognize as acceptable words.
The "Pre-Built Environment" are pre-implemented environments by us, that simulate some real-world situations and they are built with models available in the "Available Models" list, downloaded from the Unity Asset Store. Additionally, there is the possibility to select the desired number of specifics models to include inside the environment, as long as they belong to the same category; but we will see more details about these differences in 4.4.1 and 4.4.2.

The models/objects are divided into 5 different categories:

- **Furniture** : Bed, Chair, Table, Drawer, Desk, Shower, Sink [21]
- **Nature** : Wood, Rock, Oak, Mushroom, Flower, Bush, Pine [20]

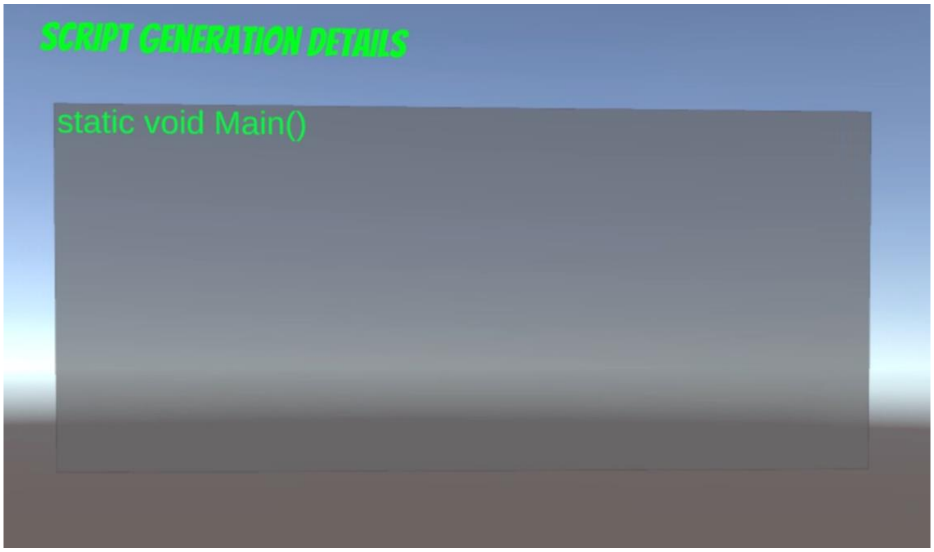
- **Cars** : Cops, Sedan, Sport, Suv, Taxi [22]
- **City** : Barrel, Bench, Bin, Dumpster, Hydrant, Mailbox [23]
- **Industry** : Cable, Car, Garbage, Plank, Tank, Tubes [24]

Moving on with the framework's windows, we introduce the so called "Script Request" window, where the user can ask for the desired Pre-Built Environment or for the specific models.



Screenshot 5: Script Request Window with Vocal Commands Buttons

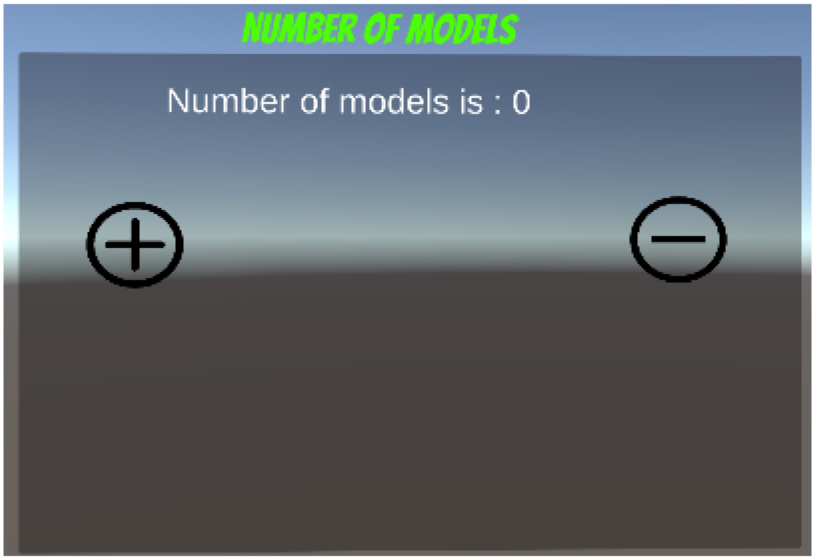
In the window on the left, the user can write the environment he wishes by choosing it from the pre-built environments; for example, "Please build an office for me, thanks". After typing the sentence, he has to click the button "Generate Script" and the LLM will create an appropriate C# script for the generation of a customized environment.
There are two more buttons in Screenshot 5, "Start Recording" and "Stop Recording". If the user clicks with the Meta Controller "Start Recording", he can make a request just by using his own voice, thanks to a microphone installed in the Meta Quest 2 and 3.1.5. When his sentence ~~it~~ is finished, he has to click "Stop Recording", wait few seconds, and his sentence will be inserted inside the window "Script Request"; then, as before, he has to click the button "Generate Script" for completing the operation and generating the environment.



```
static void Main()
```

Screenshot 6: Script Generation Details Screenshot

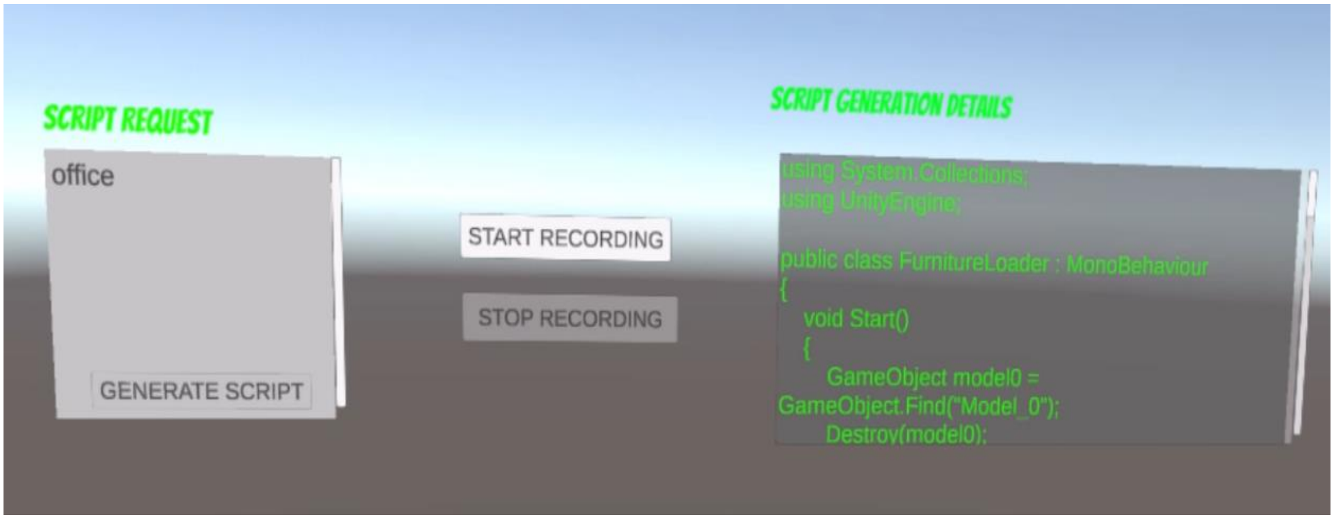
The third window that we will describe is the “Script Generation Details”, Screenshot 6. This window allows the user to visualize all the C# code generated by the large language model and in this way the user can understand exactly what the script is doing and understand a little bit more how the system is designed.
In Screenshot 6 we can see how the window appears, inside the headset, when the scene is empty (the window contains just a simple string “static void Main()”).



Number of models is : 0

Screenshot 7: Number of Models inside the environment.

The last Unity Canvas inside the framework is the one that tracks the number of models present in the environment, called “Number of Models”. The number of the base case is 0 models, as shown in Screenshot 7.
For example, if the prebuilt environment selected is “Office”, the counter will be equal to 5 and so the string in the window will declare “Number of models is: 5”.
However, if the user would like to create his own world by using the Free Models game mode, he can directly decide the number of models in the scene by pressing the button “+” if he wants to increase the number of models, or he can press the button “-“ if he wants to decrease the number of models.
If he wants to use this modality, he must remember to ask for the exact number of models, otherwise an error will be raised in the “Script Generation Details” window. For instance, if the number of models in the window is 10, the user must insert, 10 models belonging to the same category in the “Script Request” window.



```
using System.Collections;
using UnityEngine;

public class FurnitureLoader : MonoBehaviour
{
    void Start()
    {
        GameObject model0 =
GameObject.Find("Model_0");
        Destroy(model0);
```

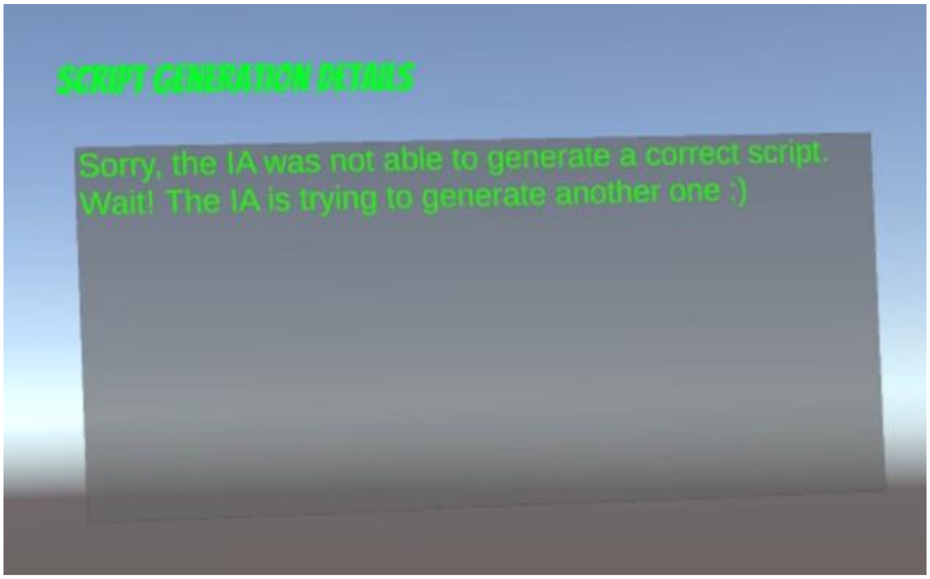
Screenshot 8: Requesting an Environment Example

In this last screenshot Screenshot 8, taken from the headset view, we would like to present a complete example of what happens when the user requests a prebuilt environment (“office”, in this case).
In the “Script Request” window, the requested environment is asked, and the “Generate Script” button clicked. It is possible to notice that the button is deactivated until the environment is actually created (the world is not yet ready).
In the other window, we can see that the C# script has been generated correctly by the IA and it is possible to check it using the scrollbar.

In case the C# script generated is not acceptable (we will discuss this in Chapter 5 and 6.1.2), an error message will appear in the “Script Generation Details” window, but all the user needs to do is wait. Another request will be sent to the LLM until a satisfactory result will be provided and executed.
In Screenshot 9, it is showed this type of error.

There two more error messages that could be displayed in the “Script Generation Details” window.
The first can appear if the user asked for a prebuilt environment which does not still exists inside the framework, the system will raise the message “Error: The environment you asked is not implemented yet, sorry”).

The second one could be raised if the user asks for a certain number of models but the total number of them do not correspond to number displayed in the “Number of Models” window , Screenshot 7. In this case the error message is going to be “Error: you have to ask for the exactly number of models requested for this simulation”.


SCRIPT GENERATION DETAILS

Sorry, the IA was not able to generate a correct script.
Wait! The IA is trying to generate another one :)

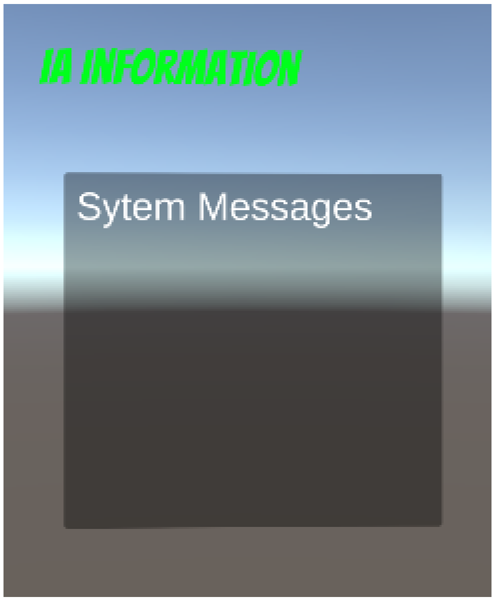
Screenshot 9: Error-Script Unacceptable

4.3 User Mode

The second modality of usage is called "User Mode." Basically, it is similar to the previous mode but removes technical information that a typical user without a background in computer science might not understand. For instance, the window with all the C# code information is substituted with a simpler one, which we will demonstrate later.
The important aspect of this modality is that we do not reveal the details about the large language model and C# code to the user. Instead, the focus is on the final result: creating a satisfactory virtual custom environment.

To explore further the user experiences when he wears the Meta Quest 2 headset, it is important to note that there are some windows which are completely identical to those

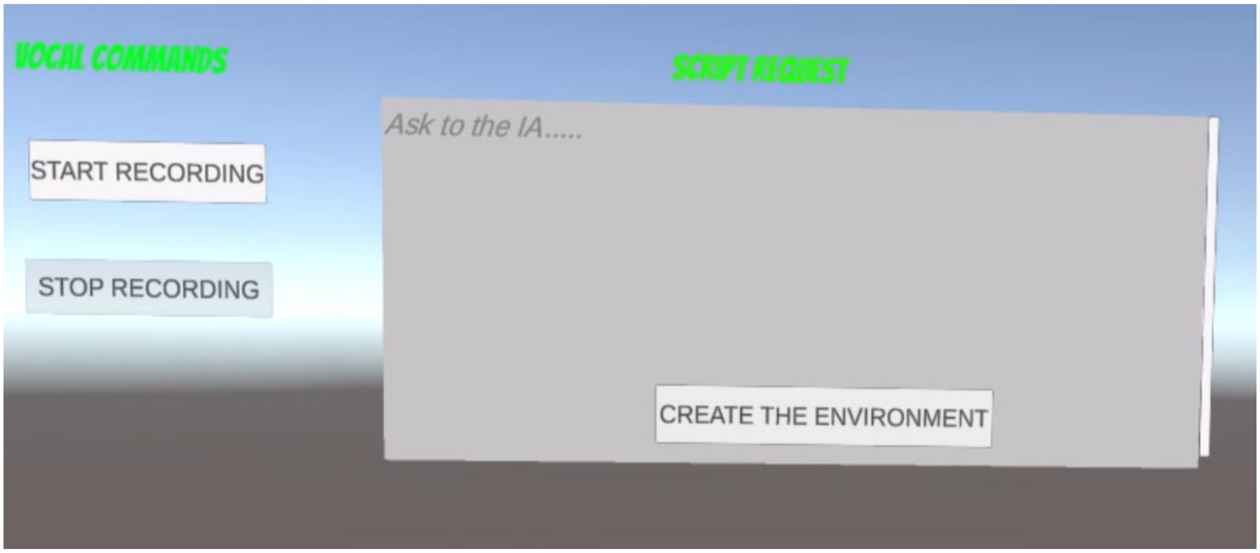
in Developer Mode. These include the "Number of Models" window (Screenshot 7) and the "List of Models and Environments" window (Screenshot 4).
However, the remaining two windows on the left differ slightly from those in Developer Mode, which we will now explain.



Screenshot 10: AI Information Window (User Mode)

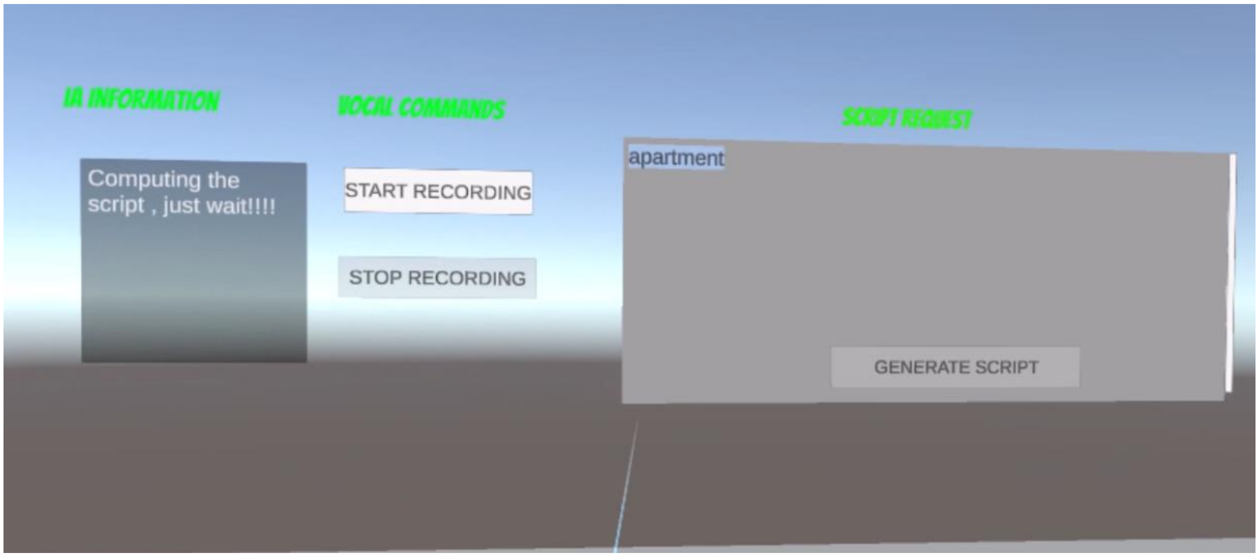
The first window is called "AI Information", and it displays all the system messages: the error messages previously seen and other system status messages such as:

- "Computing the script, just wait!!!": this tells the user that the system is taking care of his request.
- "Sorry, the IA was not able to generate a correct script. Wait! The IA is trying to generate another one": this tells the user that the LLM did not generate an acceptable C# script and so it needs to repeat the operation.
- "Executing……": The last message that pops up when everything is fine is. This means that everything is correct and C# script can be executed providing the customized environment



Screenshot 11 :Script Request Window (User Mode)

In this modality, the Script Request window takes place in the Script Generation Details in the Developer Mode (Screenshot 11). The purpose of this is equal to the one in the other scene; the user writes the request for a customized virtual environment and presses the button “Create the Environment”. He has, as before, the possibility to use the vocal commands by pressing the buttons “Start Recording” and “Stop Recording” in order to avoid the usage of the keyboard.



Computing the script , just wait!!!!

apartment

Screenshot 12: Usage Example of the User Mode

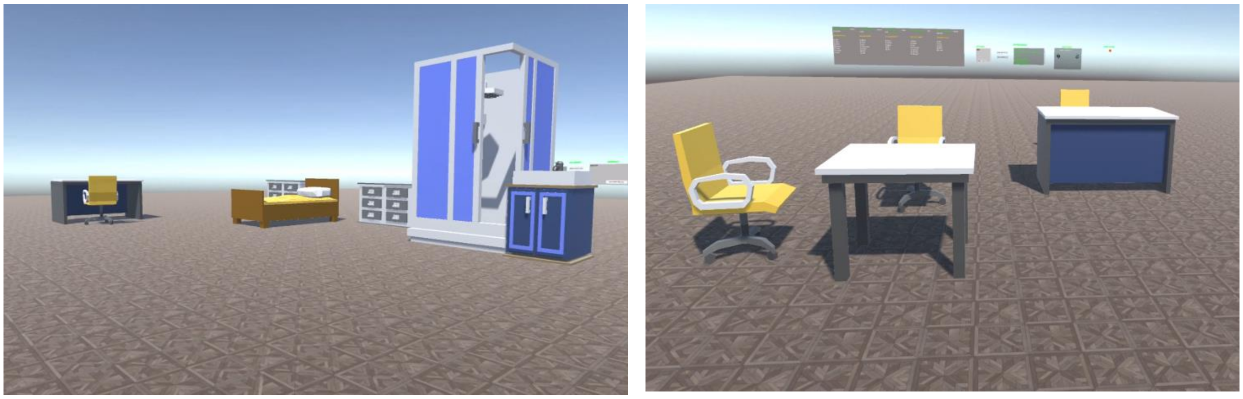
In the example above (Screenshot 12), the user has requested an apartment as **the** environment, and we can see that the button “Generate Script” is unclickable, and the IA Information window is telling us that it is handling the user’s request.

4.4 Environments

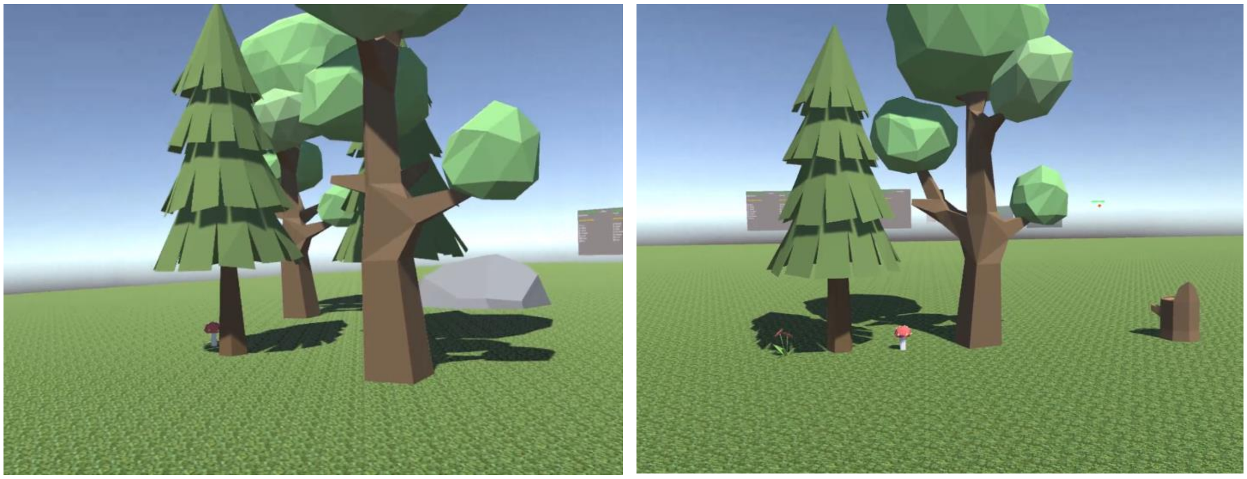
4.4.1 Pre-Built Environments

Inside the framework there are some pre-built environments that should simulate real-world rooms or habitats. We have implemented 7 different environments (we will analyse how all of this has been made possible in the next chapter, Chapter 5).
Every prebuilt environment is made of models that are also available on their own when the user does not want to use environments already implemented by us.
In addition to this, we have to consider that there is an extra model that must be taken into account, which is the pavement.
The pavement for every category of models changes texture with a particular one; for instance, if the user asks for a natural environment the pavement's texture will be changed into a grass-like texture.
Let's see which environments are available and which models/objects they contain :

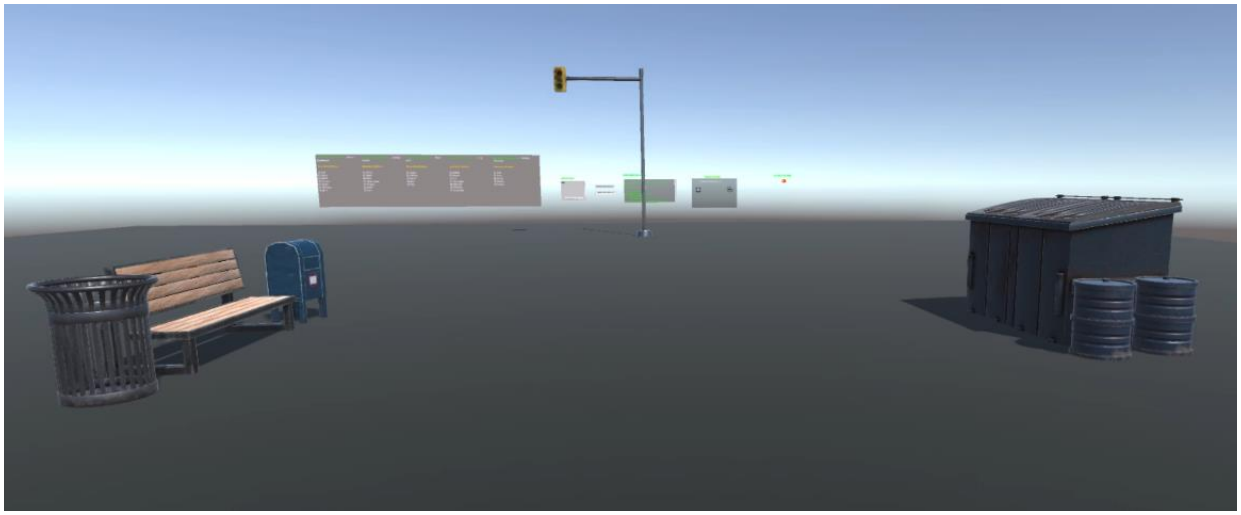
- **Apartment** : Table, Chair, Bed, Drawer x2, Shower, Sink (7 models total and parquet texture [25]).
- **Office**: Chair x3, Table, Desk (5 models total and parquet texture [25]).
- **City** : Barrel x2, Bench, Dumpster, Bin, Mailbox, Stoplight (7 models total and asphalt texture [26]).
- **Nature**: Oak, Mushroom, Wood, Pine, Flower (5 models total and grass texture [27]).
- **Forest** : Oak x2, Pine x2, Mushroom , Stone (6 models total and grass texture [27]).
- **Industry** : Garbage, Pallet x2, Plank, Tube, PalletCar (6 models total asphalt texture [26]).
- **Grid** : Cops, Sedan, Sport, Taxi, Suv (5 models total road asphalt texture [26]).



Screenshot 13: Apartment and Office prebuilt Environments.



Screenshot 14: Nature and Forest prebuilt Environments.

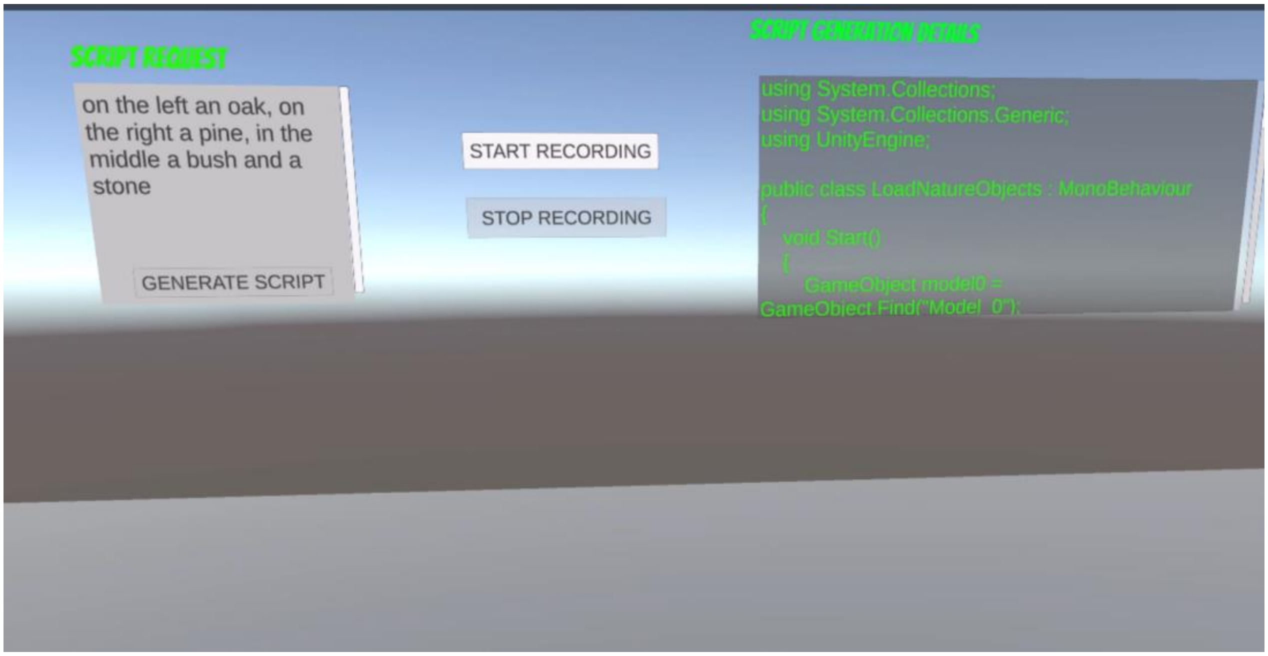


Screenshot 15: City prebuilt Environment.

Screenshot 16: Industrial and Cars prebuilt Environments.

4.4.2 Free Models

There is a second game mode that can be used, when we intend to create a customized virtual reality environment.
The best way to explain how this works is through a simple example (Screenshot 17).



on the left an oak, on the right a pine, in the middle a bush and a stone

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class LoadNatureObjects : MonoBehaviour
{
   void Start()
   {
      GameObject model0 =
GameObject.Find("Model_0");
```

Screenshot 17: Example for Free Models Game Mode

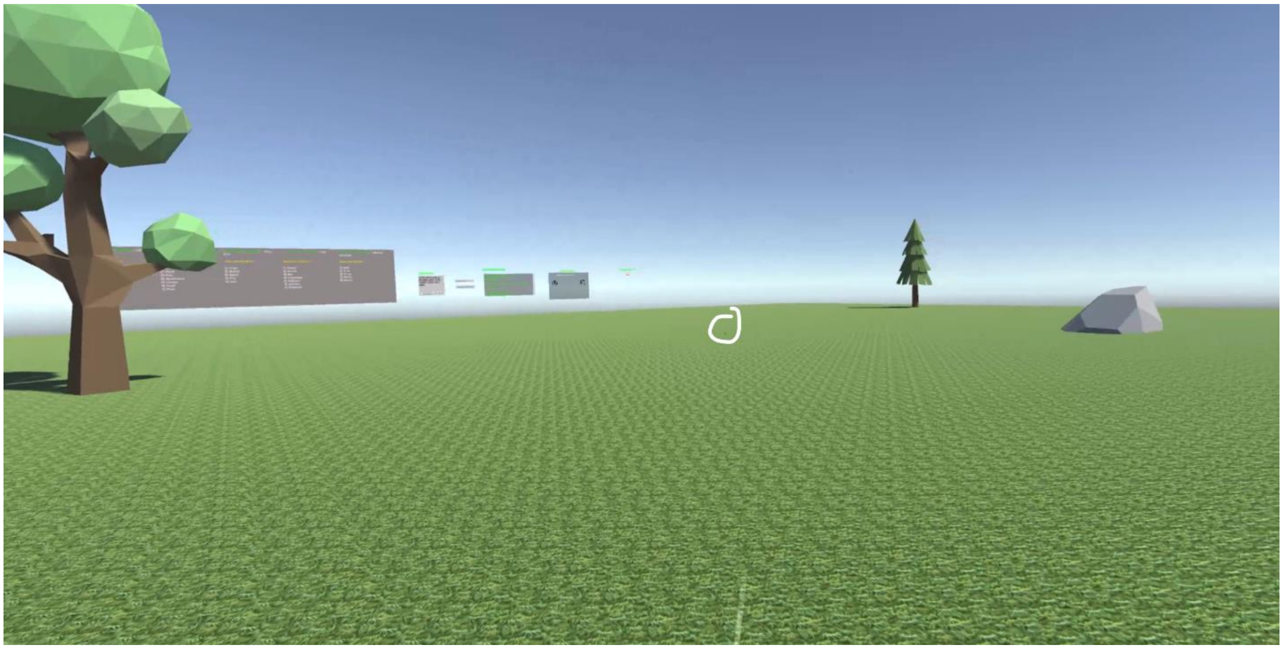
In this example we are in "Developer Mode", and we are asking the system the placements of 4 models (an oak, a pine, a bush, and a stone) in the environment. The

interesting part of this modality is the possibility to let the user choose the position of a certain model.
There are 4 different positions: left, right, middle/center, and random.
Basically, the model will be positioned on the left/right/middle (center) available part of the pavement of the environment; otherwise, if the user does not specify the position, the model will be placed randomly in the world.
In this modality there are two strict rules that must be followed.

1)The position of the object in the Script Request window must be written before the name of the model, for example "Please I would like on the right a pine, on the left an oak and in the middle a stone".

2)The user can just select models belonging to the same macro category. An acceptable example is "position a pine, an oak, another pine and a stone". An unacceptable example is "place a pine, a mushroom, a chair, a dumpster".

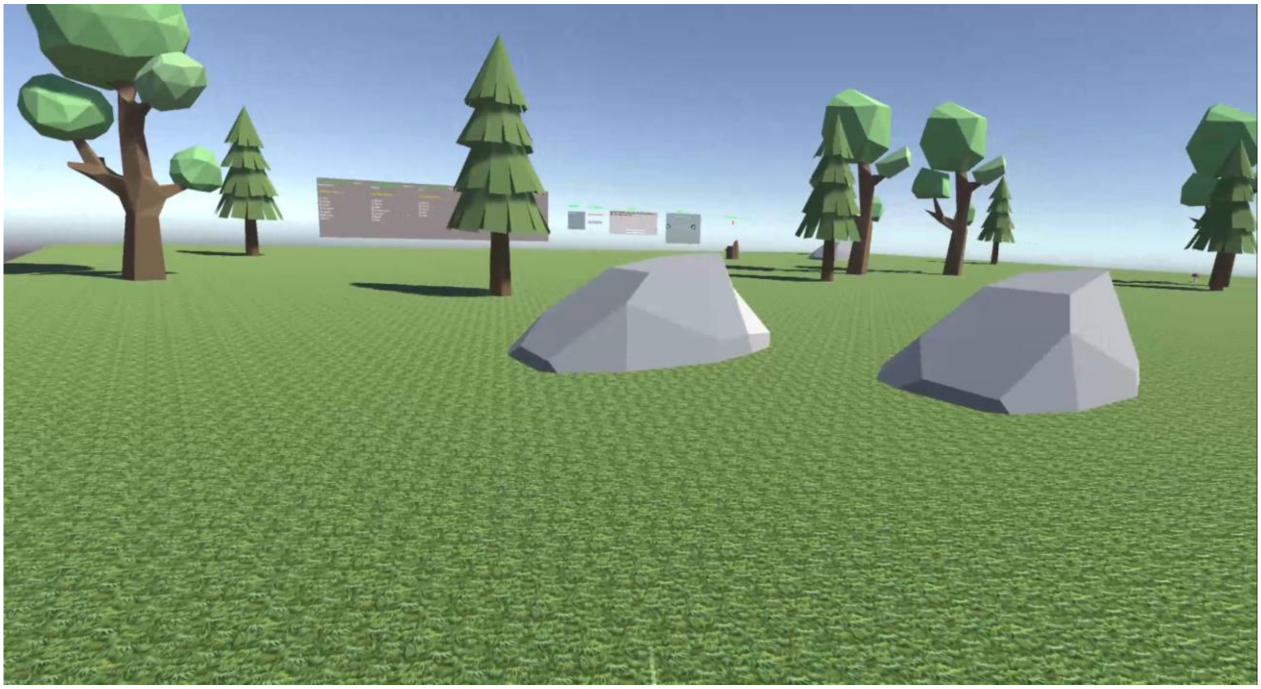
In the example above, the user wants on the left an oak, on the right a pine, in the middle a bush and a stone (no position inserted).
Here is the result:



Screenshot 18: Free Models Example

As we can see in Screenshot 18, the Oak is positioned on the left part of the environment, the bush in the middle (the model in the picture is circled because it is

too small to be clearly visualized), the pine is on the right part , and the stone is randomly positioned.
The pavement's texture has been changed into grass because we have selected models belonging to the same macro category which is nature.
This macro category is the one that works better with this modality, considering that the real world/nature environment includes hundreds of trees, plants and rocks positioned randomly. Here is an example of an environment with 15 objects chosen by the user.



Screenshot 19: Example of Free Models with 15 models Environment

We must not forget that in this case the window "Number of Models" Fig. 9 is set for having 4 models in the environment. If the user would like to increase or decrease the number of models in the scene, he just has to press the buttons "+" or "-". Of course, if the user inserts too few models or more than expected models, the system will raise an error message.

4.5 Log File Generation

Because we wanted to track every script generated for every single session and all the information related to it, we implemented a useful extra feature for both modalities "Developer Mode" and "User Mode".

Basically, when the script is correctly generated and accepted by the system, all the script details will be recorded inside a text file in order to track all the useful information that the user could read afterwards.
The file is saved inside a folder called “Logs” which is inside the Unity folder called “Assets”. The files have a unique name in order to avoid any type of name conflicts. For instance, the name of the file below, Code Snippet 3, is 10-05-05-44-13, has been generated May 10th at 5.44 and 13 seconds.
If in one session the user asks for more than one environment, all the information of the next environment will still be contained in the same file; in Code Snippet 3 the requested environment is only one.

The information printed in the file includes:

- The model used to generate the script.
- The total number of models inside the virtual environment.
- The complete sentence written by the user in the “Script Request” window.
- The C# code generated by the model.
- The amount of time taken to generate the C# script expressed in seconds.
- The number of tries required by the model to generate an acceptable script.

Having a log file like this would help the user to perfectly understand what is going on with the script generation and what is behind the framework because the complete C# script perfectly highlights how the system is designed.

After some sessions, it is suggested to delete the old unnecessaries log files in order to avoid wasting of space in the computer.

```
LOG GENERATED FOR THE SESSION
Model - gpt-3.5-turbo
Number of models in the scene ~ 5
You wrote the following  sentence : office

The script generated by the AI is the following:
 using UnityEngine;

public class FurnitureLoader : MonoBehaviour
{
    void Start()
    {
        GameObject model0 = GameObject.Find("Model_0");
        Destroy(model0);

GameObject newModel0 = Instantiate(Resources.Load<GameObject>("Furniture/Desk"), new
Vector3(0.08f, -0.47f, 7.13f), Quaternion.identity);
        newModel0.name = "Model_0";
        newModel0.AddComponent<BoxCollider>();

        GameObject model1 = GameObject.Find("Model_1");
        Destroy(model1);
        GameObject newModel1 = Instantiate(Resources.Load<GameObject>("Furniture/Chair"),
new Vector3(0.13f, -0.47f, 9.25f), Quaternion.identity);
        newModel1.name = "Model_1";
        newModel1.AddComponent<BoxCollider>();

        GameObject model2 = GameObject.Find("Model_2");
        Destroy(model2);
        GameObject newModel2 = Instantiate(Resources.Load<GameObject>("Furniture/Table"),
new Vector3(-2.64f, -0.47f, 4.62f), Quaternion.identity);
        newModel2.name = "Model_2";
        newModel2.AddComponent<BoxCollider>();

        GameObject model3 = GameObject.Find("Model_3");
        Destroy(model3);
        GameObject newModel3 = Instantiate(Resources.Load<GameObject>("Furniture/Chair"),
new Vector3(-2.76f, -0.47f, 6.28f), Quaternion.identity);
        newModel3.name = "Model_3";
        newModel3.AddComponent<BoxCollider>();

        GameObject model4 = GameObject.Find("Model_4");
        Destroy(model4);
        GameObject newModel4 = Instantiate(Resources.Load<GameObject>("Furniture/Chair"),
new Vector3(-4.37f, -0.47f, 4.81f), Quaternion.Euler(0, -97.34f, 0));
        newModel4.name = "Model_4";
        newModel4.AddComponent<BoxCollider>();



        GameObject plane = GameObject.Find("Plane");
        Renderer planeRenderer = plane.GetComponent<Renderer>();
        Material newMaterial = Resources.Load<Material>("Furniture/Material");
        planeRenderer.material = newMaterial;
    }
}




Elapsed time for the generation of the script | 8,443235 seconds
The IA required 1 tries , for obtaining an accetable script
```

Code Snippet 3: Log File Example

In 5.5, we will also analyse a Python script, based on these kinds of files, which speeded up the testing phase in order to obtain noticeable graphs for the analysis of accuracy and errors rate.

Chapter 5 Implementation

After the definition of the design of the system, we will analyse and further explore the implementation of the framework. In this chapter, we will provide a fully technical description, including code snippets, of all the blocks needed to build the final framework, paying more attention to the implementation of the key points of the project.

The complete source code can be found at the GitHub project page:
https://github.com/Riz97/Master-Thesis

5.1 Chat.cs

This first script is the heart of the framework because it is responsible for executing multiple tasks and achieving different goals. The script has many methods, implemented for all the different tasks supported by auxiliary methods in order to avoid repetitive code.
The main object of this script is to build a request, in the form of a string, that will be sent to the LLM responsible for creating an acceptable script , through the OpenAI API [16], of all the different environments and of the Free Model game mode.
The definition of "acceptable" is implemented in Chat.cs too as a sort of meta language, 5.1.4.
The implementation of the free model modality, for its complexity, presents more methods than the prebuilt environment modality.
So, we will divide the description of the script into four different small subchapters : Start() 5.1.1, ReadStringInput() 5.1.2 for Prebuilt Environments, ReadStringInput() 5.1.3 for Free Models and Meta Language 5.1.4.

5.1.1 Start()

This is the main method of the script, and it is executed every time the user asks for a new virtual environment or when the script generated is not acceptable and then not executable by the runtime compiler.
The method starts by resetting the material of the object "Plane", which is the pavement of the environment. Initially, the pavement's material is the standard material but subsequently, it must be reset in order to avoid having a pavement which is not related to the category of models considered.

Then we check if the input string (the real request) is empty or not. If it is not, the request is sent to the OpenAI API with the code that we have seen before in Code Snippet 2, and when the script is ready , it is saved inside the string *result*.

During this process, when the request is sent, a timer is started and then stopped once the system interprets the result as a good script.

In addition to this, a variable called *tries* is incremented by 1 every time the method is executed. Those two variables will monitor the amount of time needed for obtaining a script and the amount of tries the system requires for generating an acceptable script.

On the contrary, if the script it is not satisfying, the method is executed again until the LLM output a good result.

If the script has a correct format (we will see why in 5.1.4 and in Chapter 6), in the "Script Generation Details" window for the Developer Mode the C# code will be printed and in the "AI Information" window for the User Mode the message "Executing..." will be displayed.

Now we will explain how the input request is built.

5.1.2 ReadStringInput() – Prebuilt Environment

This method is attached to the "Generate Script"/" Generate the Environment" button, and it is executed every time the button is clicked.

We must outline that, at the beginning, the scene has no models inside of it, the only one is the pavement which is the same for every environment.

Therefore, the first necessary operation is the creation of the reference objects; they are, basically, invisible objects with no mesh renderer. They will be substituted with the real models that form a prebuilt environment and they have to correspond to the exact number of references otherwise when the script is executed by the runtime compiler, it will try to modify models that do not exist in the scene, then **display** an error. Consequently, the script will not work correctly.

The methos responsible for this task is *createModels(int Number).* It creates *Number* models, and it renames them as "Model_*i*" (with *i* = 0 and *i* < *Number*).

The operation of renaming is fundamental because the system will always be able to find the objects and change them whenever a new script is executed. Otherwise, after some shots in the same session, we would have an exaggerated number of models with different names in the Unity hierarchy and the framework would not be able to locate them and go on correctly.

Afterwards, we set the global variable "Number of Objects" equal to the number of models in the selected prebuilt environment (e.g., Office has 5 models).

This variable is very important when a new environment is requested by the user, because we need to delete all the models belonging to the previous environment and set the scene as it was.

All the models inside the scene are found, thanks to their name (Model_0, Model_1 etc..), through a for loop and, thanks to the global variable, the exact number of models

in the scene is deleted, without exceeding the limits, otherwise we would get a compilation error.
At this point, it is possible to write the request in detail and save it inside the string variable *input*, seen in the Start() method.
Finding the right way to write the string was quite challenging but we will delve deeper into this aspect and explain how the request was designed in sections 6.1.1 and 6.1.2.
Here, we will show the best request we have formed for the environment called “Office”.

```
IT IS MANDATORY TO WRITE ONLY C# CODE NOT ANY OTHER COMMENT PLEASE NO APOSTROPHES
AT THE BEGINNING OR AT THE ENDING  Write a C# unity script with libraries
inclusion that the first thing to do must be find using the Find() method the
gameobjects  called 'Model_0', 'Model_1', 'Model_2' 'Model_3 'Model_4  and
destroy them and YOU MUST  substitute them  with the gameobjects THAT YOU MUST
load  from the folder named 'Furniture' inside the folder  'Resources' called
'Desk',
"Chair' 'Table' 'Chair' 'Chair'  , You MUST RENAME THEM AS 'Model_0', 'Model_1',
'Model_2' 'Model_3 'Model_4  in the unity hierarchy MANDATORY
'Model_0' (Desk) at Y position equals to -0.47, at X position 0.08 and Z position
7.13 , 'Model_1' (Chair) at Y position equals to -0.47, at X position 0.13 and Z
position 9.25 'Model_2' (Table) at Y position equals to -0.47, at X position -2.64
and Z position 4.62 'Model_3' (Chair) at Y position equals to -0.47, at X position
-2.76 and Z position 6.28  'Model_4' (Chair) at Y position equals to -0.47, at X
position -4.37 and Z position 4.81 and Y rotation equals -97.34"
and add just one collider per gameobject, find the gameobject named Plane and
change its  material with the material   called 'Material' THAT MUST BE LOADED
inside the 'Furniture' folder which is inside the folder Resources and do not
destroy it, using a method called Start , avoid any type of comments , you must
write only code
```

LLM Request 1: Environment "Office”.

The requests for the other environments follow the same pattern, the only elements that change are the coordinates and obviously the objects. Then, *Start()* is executed and *input* will be used by the system and sent to the chosen LLM.

If the environment requested does not exist, an error message will be displayed, the execution will be interrupted, and the “Generate Script” button will be interactable again.

5.1.3 ReadStringInput() – Free Models

This modality, because of its more complex characteristics, is implemented with the support of more auxiliary methods.
The objective is still the same, to create a request string which will be used as input for the OpenAI API, but, in this case, the user must be free to select the specific models he likes, the number of models and the position, without particular restrictions.

The beginning is almost equal to the previous case, but here the number of models necessary for the scene are set in the “Number of Models” window.

Through the methods *Add()* and *Subtract()* , attached to the buttons “+” and “-“, the user can increment or decrement the global variable *Number_of_Objects* , that will be used as parameter in the function *CreateModels(int Number_of_Objects)*. Thus, the system will know how many models need to be created.

Now, that the models are inside the scene, we need to build the input string. This has been made thanks to the method implemented below, *Input_Request(string input, int Number_of_Objects, List <string> list, string Material, List<string> list_directions)*.

How we handled the two parameters *list* and *list* directions and how they work will be explained in detail in 5.1.4

Here it is the code of *Input_Request()* :

```
1   public string Input_Request(string input, int Number_of_Objects, List<string>
2   list, string Material, List<string> list_Directions)
3
4 {
5
6 input = "the first thing to do must be find using the Find() method the
7 gameobjects  called ";
8
9 input = Define_Models(Number_of_Objects, input) + "and  destroy them and YOU
10 MUST   substitute them  with the gameobjects THAT YOU MUST   load  from the
11 folder named " + Material +  " inside the folder  'Resources' called ";
12
13 input = Enum_Objects(list, Number_of_Objects, input) + "You MUST RENAME THEM AS
14 ";
15
16 input = Define_Models(Number_of_Objects, input) + " in the unity hierarchy
17 MANDATORY";
18
19
20 input = Define_Models_Coordinates(list,Number_of_Objects,input,list_Directions)
21 + " and add just one collider per gameobject, find the gameobject named Plane
22 and change its" + "material with the material   called 'Material'THAT MUST BE
23 LOADED inside the" +  Material + " folder which is inside the folder
24 Resources," +
25 "using a method called Start , avoid any type of comments , you must write only
26 code";
27
28
29       return input;
30   }
```

Code Snippet 4: Input Request Method

The code is structured in different steps and the main objective is to obtain a string which is similar to the one in LLM Request 1, but in this case, we do not know how many models the user will ask for in advance, as before.

In lines 6 and 7, we have the first sentence that form the request which is equal to the one showed in the LLM Request 1 (the first three lines of the request are already inserted, they are constant for every request) and must be concatenated with the others.

In line 9, there is a new method that must be introduced: *Define_Models(int Number_of_Objects, string input).*

With this method we would like to form the sentence: “Model_0, Model_1, Model_2 ….”. The AI must previously know the number of models to find in the Unity Hierarchy, and in this way , thanks to the global counter *Number_of_Objects*, we have no problem with that.

In line 11, we specify the texture material for the pavement, which is passed as parameter.

At this point the string is the following (we are considering a 3-model environment, taken from the macro category, Nature) :

```
“the first thing to do must be find using the Find() method the  gameobjects  called
'Model_0', 'Model_1', 'Model_2' and  destroy them and YOU MUST  substitute them  with
the gameobjects THAT YOU MUST   load  from the folder named 'Nature' inside the folder
'Resources' called”
```

Next, we need to specify the names of the objects which we do in line 14. They are loaded inside the folder Resources/Nature in order to ensure a correct uploading code lines by the AI (Model_0, Model_1….).

We achieved this by defining a new method called *Enum_Objects(List<string> objects, int Number_of_Objects, string input).*

It concatenates the models’ names that are inside the string list *objects* with the string *input*.

After this operation the input string will look like this:

```
“the first thing to do must be find using the Find() method the  gameobjects  called
'Model_0', 'Model_1', 'Model_2' and  destroy them and YOU MUST  substitute them  with
the gameobjects THAT YOU MUST   load  from the folder named 'Nature' inside the folder
'Resources' called 'Oak', "Pine' 'Mushroom' , Youm MUST RENAME THEM AS”
```

Now, that the real models’ names are uploaded, we must rename them in order to let them be recognizable in the Unity hierarchy.

In lines 16 and 17, we do this operation by using the method we have previously seen, *Define_Models(int Number_of_Objects, string input).*

Thus, the new input string will look like this:

```
“the first thing to do must be find using the Find() method the  gameobjects  called
'Model_0', 'Model_1', 'Model_2' and  destroy them and YOU MUST  substitute them  with
the gameobjects THAT YOU MUST   load  from the folder named 'Nature' inside the folder
'Resources' called 'Oak', "Pine' 'Mushroom' , Youm MUST RENAME THEM AS Model_0 Model_1
Model_2” in the unity hierarchy MANDATORY”
```

All the models are now correctly found in the folders and named in the hierarchy.

The next and last step, in line 20, is the creation of the input string in order to associate all the new models “Model_0, Model_1, Model_2” to their objects’ meshes, and to give those objects a correct position in the environment.

We have sorted this out by developing another method called *Define_Models_Coordinates(List<string> objects, int Number_of_Objects, string input, List<string> list_Directions).*

Firstly, it specifies the object name for a certain model (e.g. Model_0 is a Chair); then it states the position of that object, starting from the Y coordinate which is the same for every object. Next, through two different float methods: *Random_PositionX(List<string> list, int i)* and *float Random_PositionZ(List<string> list, int i)*, the system will position the model randomly in the part of the environment requested by the user (Left,Right,Center). This will be analysed in detail in the following subchapter.

We also ask the LLM to create a collider for every object. This ensures that when we wear the headset and roam around the environment, we cannot pass through the objects, but they will act like realistic things.

```
“the first thing to do must be find using the Find() method the  gameobjects  called
'Model_0', 'Model_1', 'Model_2' and  destroy them and YOU MUST  substitute them
with the gameobjects THAT YOU MUST   load  from the folder named 'Nature' inside the
folder  'Resources' called 'Oak', "Pine' 'Mushroom' , Youm MUST RENAME THEM AS
Model_0 Model_1 Model_2” in the unity hierarchy MANDATORY 'Model_0' (Oak) at Y
position equals to -0.47, at X position 0.34 and Z position 2.13 , 'Model_1' (Pine)
at Y position equals to -0.47, at X position 0.45 and Z position 7.25 'Model_2'
(Table) at Y position equals to -0.47, at X position -2.64 and Z position 4.62
'Model_3' (Chair) at Y position equals to -0.47, at X position  -2.76 and Z position
6.28  'Model_4' (Chair) at Y position equals to -0.47, at X position -4.37 and Z
position 4.81 and Y rotation equals -97.34"
and add just one collider per gameobject, find the gameobject named Plane and change
its  material with the material   called 'Material' THAT MUST BE LOADED inside the
'Furniture' folder which is inside the folder Resources and do not destroy it, using
a method called Start , avoid any type of comments , you must write only code”
```

The string is now complete and *Start()* can be executed by using the variable *input* which can be used as input for the LLM in order to generate the necessary C# script.

```
else if (words_Furniture.Count() == Number_of_Objects)
{

   createModels(Number_of_Objects);

    input = Input_Request(input, Number_of_Objects, words_Furniture,
  "Furniture",list_Directions);

    Start();


 }
```

Code Snippet 5: Example of Free models , Furniture.

The code above is the same for every other macro category developed inside the framework; in this case the selected category is “Furniture”.

5.1.4 Meta Language

We have previously explained how the input string has been formed for communicating with the LLM, but we now must analyse another block of the implementation: Meta Language.

We did not want to oblige the user to write a request, like the one in LLM Request 1 inside the “Script Request” window (Screenshot 5 and Screenshot 11) because it would be impossible for a basic user to write something like that; so, we had to implement a language that would overcome a problem like that.

The idea behind this meta-language is not difficult, basically we have selected key words for prebuilt environments and, the object meshes and their position for the Free Models game mode; then, when the user request is submitted the system recognizes them and it starts the right environment request to send to the LLM.

The last thing that must be said is the fact that the system is case-insensitive, it will do not care, for example, if the user writes “Office” or “office”.

Lastly, it's important to note that the system is case-insensitive and when the user enters "Office" and "office" it will not differentiate between them.

As before, we analyse the meta-language for the prebuilt environments, which is more straightforward.

First of all, the user writes the request as he wishes, (e.g. “Please build an office for me); at this point the string written is compared with a list of accepted prebuilt environment key words. If the user’s input string contains an acceptable prebuilt environment, the real request will be sent to the LLM as explained in chapter 5.1.2.

This was made possible thanks to the method called *ContainsAny(string Input, List<string> substrings);* if inside the string *s*, there is at least one string of the *substrings* set, it will come back true.

If the user mistakenly writes more than one prebuilt environment in the window, the system will only consider the first one written.

On the other hand, the implementation for the Free Models modality is different and harder than the previous one.

Everything starts with the string written by the user in “Script Request” window, which is extracted and saved in the variable *input*.

In this case, the string is split up into tokens and all the tokens are compared with the related list that contains all the objects’ names belonging to a macro-category. Then, a list with only the models asked is returned.

For example: “Please I would like an oak, then a pine, another oak and a mushroom”.

The list will be the following: “oak” | “pine” | “oak” | “mushroom”.

The method *List<string> isIn(string s,List<string> Vocab)* is responsible for this task. The returned list is the first parameter list that is used in the *Define_Models_Coordinates()* and in the *Input_Request()* method.

We have to remember that in this modality, there is the possibility to select the part of the environment where you want to position your models.

We need to create another list of strings that will contain the directions of the objects. Firstly, we must create an auxiliary list through the method *List<string> isIn_Direction(string s, List<string> Direction, List<string> allwords)*, where *Direction* are the acceptable directions (Right, Left, Center | Middle) and *allwords* are all the models accepted by the system.

A possible example could be: "Please put on the right an oak , on the left a pine , and a mushroom". The result list is: "right" | "oak" | "left" | "pine" | "mushroom".

The next step is to populate another list but only with directions and the indexes will represent the objects, so we have to eliminate the objects' names.

If in the auxiliary list the object is preceded by a direction, we are going to remove the object from the list, otherwise (the user did not specify a position) we fill the list with a blank; the result is then saved in a new list called *List_directions*. So, if we consider the previous example the final list will be: "right" | "left" | " "

List_directions will be passed as parameter in the method *efine_Models_Coordinates()* and used in the methods *Random_PositionX()* and *Random_PositionY()*; if the directions are acceptable the objects will have a random position within limits (e.g. Right, means that the model will be positioned randomly in the right part of the environment), otherwise if no direction is specified, the object will be just positioned randomly.

It is very important that the position of the object **is** written firstly.

In the end, if the user asks for more objects than expected the error "Error: you have to ask for the exactly number of models requested for this simulation", will be displayed.

5.2 Domain.cs

This script is responsible for the correct runtime execution through the runtime C# compiler Roslyn [12], of the newly generated script by the LLM and for the generation of the Log file explained in 4.5.

The script will be attached to an empty gameobject that will execute the new code extracted from the "Script Generation Details" window (Screenshot 6) in the Developer Mode and from an invisible window in the User Mode, but the code cannot be executed immediately because the LLM needs time for the script generation and this consideration is very important for the system to work correctly.

So, we must check through the method *Start()* that in the window's text there is only C# code, otherwise the system will stop.

When the request is made, the system is put in a waiting state for 20 seconds if the number of models asked is less than 10, waiting for the output of the LLM.

If the number of models is greater, the waiting state will last longer.

Those seconds are enough in order to obtain a C# script, because the LLM is able to generate it in a time between 8 and 17 seconds.
The code then is printed in the "Script Generation Details" window, It is saved inside the string variable *sourcecode* and executed, using the code explained in 3.1.3, through the Roslyn runtime compiler.
However, we have to consider the possibility that the C# script might not be accepted by the system.
If it is not accepted, an error message like this will appear in the window "Sorry, the IA was not able to generate a correct script. Wait! The IA is trying to generate another one :)". If this message is detected, the system will go back into the waiting state and this waiting state loop will continue until an acceptable script is generated.
In addition to this, when a prebuilt environment is requested and executed, a flag called *Chat.Bases* is set to true in order to let the system **know** the modality of usage chosen by the user. This ensures that the system can destroy the right number of models when the next environment is requested; the same thing happens **to** the free models modality with the flag *Chat.Custom.*
If we had not done this, the system would have deleted a wrong number of models causing a compilation error.

In the script there is one more method called *void CreateLogFile(string sourcecode, TMP_Text Input_Text).* It creates the log file Code Snippet 3, in the path selected in the session just closed by the user, taking most of the variables' values from the Chat.cs script such as *Number_of_Objects, elapsed_time* and *tries.*

5.3 Vocal_Commands.cs

In this C# script we have implemented a Speech Recognition tool through the Hugging Face Unity API [19]; a useful feature for giving vocal commands that convert spoken words to text.
The script has 3 important methods: *void StartRecording(), void StopRecording()* and *void SendRecording().*
StartRecording() records up to 100 seconds of audio at 44100 Hz, and in case the recording reaches its maximum, the recording is automatically stopped. *StopRecording()* truncates the recording and encodes it in WAV format, which is the audio format accepted by the API.
SendRecording() sends the recording to the Hugging Face Unity API and **sets** the text in the "Script Request" window (Screenshot 5 and Screenshot 11) with the words said by the user.

5.4 Teleport.cs and Change_Scene.cs

Teleport.cs allows the user to teleport himself in to two different locations. The first spot is in front of all the control windows, while the second is positioned in front of the prebuilt environments, which are situated farther away from the control panels. This feature aims to minimize unnecessary walking back and forth, thus optimizing user time.

Change_Scene.cs is responsible for the movement along the Unity Scenes such as "Opening Scene", "Developer Mode", "User Mode" and "Useful Information". All the methods inside the class are attached to the buttons for quitting the application or going back to the opening scene.

5.5 Graph_Maker.ipynb

This is a python notebook, extremely useful for making all the charts for the testing phase, 6.1.3.
The only thing to do is select a text file obtained after the conclusion of a playing session, save it inside the folder where the notebook is located and write the name of it in the variable *file*.
The notebook will extract useful information from the text file such as: the number of models used, the average number of tries required by the LLM to create an acceptable script, the LLM used and, most importantly, a line chart depicting the seconds taken by the LLM to create the C# script.

The parameters of the function are a list of timings, the average amount of tries for the entire session, the Large Language Model name, and the number of models used in the session
In the end, the chart is saved as a jpg file inside a subfolder called "Graphs", named as a concatenation of the LLM used and the number of models used.
We have to say that this notebook is useful when you want to test how the LLM behaves with the same environment or with an environment with the same number of models.
In Chapter 6, we will illustrate the charts created.

```
def Graph(Time_Execution,tries,LLM,models):

    fig = plt.figure(figsize = (25,5))

    ax1 = fig.add_subplot(1,2,1)

    aux_model = str(models)
    plt.plot(Time_execution,color='blue',  linestyle='dashed',  linewidth  =  3,
marker='o', markerfacecolor='red', markersize=12)

    plt.title("The number of models used is " +  aux_model,loc = "right")
    plt.title("LLM used: "+ LLM,loc = "left")
    plt.xticks([])
    plt.ylabel("Seconds")
    plt.xlabel("\n The average amount of tries is " + str(tries),fontsize = 20)
    plt.legend(["Script"])
    plt.grid()
    plt.savefig("Graphs/" + LLM + "_" + "NModels_" + aux_model + ".jpg")

    plt.show()
```

Code Snippet 6: Python function responsible for the creation of test charts.

Chapter 6 Experiments and Results

In this chapter, we wanted to show the results obtained during the testing phases.
The first one is the testing of the system where we analyse how we got an acceptable request to send to the LLMs and then how the LLMs react to this C# script request (Execution time and error rate) with the support of the charts created with the Python script seen in 5.5.
In the second part, we have selected some testers with different backgrounds : one set of testers are not related to the Computer Science world in no way, the second group is composed of PhD students of Computer Science department from Genoa University. These two groups have tried the framework and have answered to two different questionnaires : System Usability Scale (SUS) and the User Experience Questionnaire (UEQ).
In the end we are going to analyse the result obtained from the questionnaires.

6.1 System Testing

6.1.1 Input Formation

In LLM Request 1, we have shown the most optimal request in order to obtain an acceptable C# script most of the times, but we would also like to show all the issues encountered during the experiments. The request was suitable for all the LLMs used in the testing: GPT3_5_Turbo_16K, GPT3_5_Turbo, GPT4.
But how did we end up with that sentence?
We used an approach CoT (Chain of Thought), trying to help the model reason step by step in order to reach the best possible outcome.

```
IT IS MANDATORY TO WRITE ONLY C# CODE NOT ANY OTHER COMMENT AT THE BEGINNING OR A
AT ENDING, Write a C# unity script with libraries inclusion that the first thing
to do must be find using the Find() method the  gameobjects  called 'Model_0',
'Model_1', 'Model_2' 'Model_3 'Model_4  and  destroy them and YOU MUST  substitute
them  with the gameobjects THAT YOU MUST   load  from the folder named 'Furniture'
inside the folder  'Resources' called 'Desk',
"Chair' 'Table' 'Chair' 'Chair'   , You MUST RENAME THEM AS 'Model_0', 'Model_1',
'Model_2' 'Model_3 'Model_4  in the unity hierarchy MANDATORY,
'Model_0' (Desk) at Y position equals to -0.47, at X position 0.08 and Z position
7.13 , 'Model_1' (Chair) at Y position equals to -0.47, at X position 0.13 and Z
position 9.25 'Model_2' (Table) at Y position equals to -0.47, at X position -2.64
and Z position 4.62 'Model_3' (Chair) at Y position equals to -0.47, at X position
-2.76 and Z position 6.28  'Model_4' (Chair) at Y position equals to -0.47, at X
position -4.37 and Z position 4.81 and Y rotation equals -97.34"
```

```
and add just one collider per gameobject, find the gameobject named Plane and
change its  material with the material   called 'Material' THAT MUST BE LOADED
inside the 'Furniture' folder which is inside the folder Resources and do not
destroy it, using a method called Start , avoid any type of comments, you must
write only the code
```

LLM Request 2: Analysis

In the initial two lines of the request, we asked for a C# code without any type of comment at the beginning or at the ending of the result and we asked to insert all the libraries inclusion.

We used capital letters in order to help the LLM to understand that those parts are fundamental and cannot be avoided.

We started the reasoning by saying to the LLM that the first thing to do is to use the *Find()* method and search for all the gameobjects that would be changed later.

We had to expressly ask for the *Find()* method, because without it, the model would not have been able to find them correctly, or it would have used another method to find all the objects, which could have caused a compilation error in the execution.

Then, we analysed a crucial part of the request. We wanted the LLM to destroy those models, otherwise we would have had tons of objects in future usages, and we would have needed to substitute the objects just found with the ones that were inside the Resources/Furniture folder. In this part of the request, we used “YOU MUST” (with sentences in capital letters LLM is more careful) because it is something that must not be missed and, without this expression, the LLM would completely skip this request causing a compilation error in the execution.

In the next part of the request, we used the same expression to rename the new objects just uploaded from the folder. However, after some tests, we realized that the LLM did not rename the models correctly inside the Unity Hierarchy, an example is provided at Code Snippet 8; instead, only the variable names inside the script were renamed and this was a fundamental problem to overcome. We expressively instructed the LLM to rename the objects in the Unity hierarchy, emphasizing that this was “MANDATORY”. Unfortunately, the LLM sometimes did not include the renaming in the script itself (we will present a solution to this problem in 6.1.2).

The subsequent stage of the input formation was quite straightforward. We listed all the objects’ names inside the parenthesis in order to correctly link the objects’ names and the names in the hierarchy. We instructed the LLM to specify their X, Y and Z positions and to add a collider to every object.

Here we focused on the pavement, and we asked the LLM to change its material, which was uploaded from the Resources/Furniture folder, while ensuring that the gameobject “Plane”, representing the pavement, is not destroyed. This aspect of the request always worked well.

We inserted a reminder instructing to use a method called Start() and with no other names, because Roslyn needs a *Start()* method for the correct execution of the script. Unfortunately, the LLM sometimes used different names for the main method and so we had to insert this phrase.

In the final part of the request, we emphasized twice to the LLM to avoid any type of comments inside the C# script and to only focus on writing code. This requirement is crucial as we need the script to be compiled and executed without touching it.
Unfortunately, there were instances when the LLM completely ignored these requests and wrote all the codes with apices or inserted characters that could not be compiled as shown in Code Snippet 7 (such occurrences were resolved through an acceptability check)
Of course, the LLM cannot be aware that the code it generates will be used at runtime and so the LLM prints it as if the developer is going to modify it and then compile it which of course is comprehensible.

We would like to show two examples, highlighting the problem of the apices and the problem of the non-renaming:

~~~~
```c#
using UnityEngine;
public class ReplaceModels : MonoBehaviour
{
    void Start()
    {
        GameObject model0 = GameObject.Find("Model_0");
        Destroy(model0);
        GameObject desk = Instantiate(Resources.Load<GameObject>("Furniture/Desk"));
        desk.name = "Model_0";
        desk.transform.position = new Vector3(0.08f, -0.47f, 7.13f);
        desk.AddComponent<BoxCollider>();

        GameObject model1 = GameObject.Find("Model_1");
        Destroy(model1);
        GameObject chair1 = Instantiate(Resources.Load<GameObject>("Furniture/Chair"));
        chair1.name = "Model_1";
        chair1.transform.position = new Vector3(0.13f, -0.47f, 9.25f);
        chair1.AddComponent<BoxCollider>();

        GameObject model2 = GameObject.Find("Model_2");
        Destroy(model2);
        GameObject table = Instantiate(Resources.Load<GameObject>("Furniture/Table"));
        table.name = "Model_2";
        table.transform.position = new Vector3(-2.64f, -0.47f, 4.62f);
        table.AddComponent<BoxCollider>();

        GameObject model3 = GameObject.Find("Model_3");
        Destroy(model3);
        GameObject chair2 = Instantiate(Resources.Load<GameObject>("Furniture/Chair"));
        chair2.name = "Model_3";
        chair2.transform.position = new Vector3(-2.76f, -0.47f, 6.28f);
        chair2.AddComponent<BoxCollider>();

        GameObject model4 = GameObject.Find("Model_4");
        Destroy(model4);
        GameObject chair3 = Instantiate(Resources.Load<GameObject>("Furniture/Chair"));
        chair3.name = "Model_4";
        chair3.transform.position = new Vector3(-4.37f, -0.47f, 4.81f);
        chair3.transform.rotation = Quaternion.Euler(0f, -97.34f, 0f);
        chair3.AddComponent<BoxCollider>();

        GameObject plane = GameObject.Find("Plane");
        plane.GetComponent<Renderer>().material =
Resources.Load<Material>("Furniture/Material");
    }
}
```````````````````````````````````````````
~~~~

Code Snippet 7: Faulty Script (apices issue)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FurnitureLoader : MonoBehaviour
{
  void Start()
  {
    GameObject model0 = GameObject.Find("Model_0");
    GameObject model1 = GameObject.Find("Model_1");
    GameObject model2 = GameObject.Find("Model_2");
    GameObject model3 = GameObject.Find("Model_3");
    GameObject model4 = GameObject.Find("Model_4");

    Destroy(model0);
    Destroy(model1);
    Destroy(model2);
    Destroy(model3);
    Destroy(model4);

    GameObject newModel0 = Instantiate(Resources.Load("Furniture/Desk", typeof(GameObject)),
new Vector3(0.08f, -0.47f, 7.13f), Quaternion.identity) as GameObject;
      GameObject newModel1 = Instantiate(Resources.Load("Furniture/Chair",
typeof(GameObject)), new Vector3(0.13f, -0.47f, 9.25f), Quaternion.identity) as GameObject;
    GameObject newModel2 = Instantiate(Resources.Load("Furniture/Table", typeof(GameObject)),
new Vector3(-2.64f, -0.47f, 4.62f), Quaternion.identity) as GameObject;
    GameObject newModel3 = Instantiate(Resources.Load("Furniture/Chair", typeof(GameObject)),
new Vector3(-2.76f, -0.47f, 6.28f), Quaternion.identity) as GameObject;
    GameObject newModel4 = Instantiate(Resources.Load("Furniture/Chair", typeof(GameObject)),
new Vector3(-4.37f, -0.47f, 4.81f), Quaternion.Euler(0f, -97.34f, 0f)) as GameObject;

    Collider collider0 = newModel0.AddComponent<BoxCollider>();
    Collider collider1 = newModel1.AddComponent<BoxCollider>();
    Collider collider2 = newModel2.AddComponent<BoxCollider>();
    Collider collider3 = newModel3.AddComponent<BoxCollider>();
    Collider collider4 = newModel4.AddComponent<BoxCollider>();

    GameObject plane = GameObject.Find("Plane");
    Renderer renderer = plane.GetComponent<Renderer>();
    renderer.material = Resources.Load("Furniture/Material", typeof(Material)) as Material;
  }
}
```

Code Snippet 8: Faulty script (non-renaming issue)

Because the LLM cannot be 100% trustworthy, we developed some extra checks in order to reach our final objective and avoid faulty C# script that could block the correct execution of the algorithm.

6.1.2 Extra checks for the acceptability of the C# script

We cannot run the risk to execute a C# script with errors, because there is always the possibility that the LLM could output code with the problems previously explained. Thus, we have added four extra checks in order to get an acceptable script that will be executed.

- The first non-white space char of the new C# script must be a ‘u’ or ‘U’. We thought about this check because every Unity C# script usually begins with the libraries inclusion such as: *using System.Collections;* In addition to this, we replaced all the apices in the resulting script with empty spaces, because most of the time, the LLM inserts apices at the beginning and at the bottom of the script as it shows Code Snippet 7.

- The C# script must include some mandatory words. The first one is “*Find(“M””*, in this way the script will find the model just by using the *Find()* method and not any other similar method such as *FindGameObjectsWithTag()*. The second one is *.name*, which corresponds to the renaming code, in this way we have a script that correctly **rename**s the models and we avoid the situation in Code Snippet 8. The last word that must be present in the C# script is *Instantiate*, which is a method that creates a clone of a Prefab at runtime.

- This check is focused on the pavement’s material. We want to be sure that the texture’s material is uploaded from the correct folder, because this was a problem encountered during our perfect request research. Therefore, the code must contain *““Furniture”|”Cars”|”Nature”|”City”|”Industrial”/Material”*.

- The last check guarantees that the script contains at least one model’s name. This is necessary because sometimes the LLM completely forgot to upload the models from their respective folders.

It must be said that it was not possible to address every possible faulty script in the generated scripts because sometimes all the LLM can make some mistakes in the C# script even if it was accepted by the system, but we tried to cover most of the issues.

6.1.3 Script Generation: Execution Time and Error Rate

6.1.3.1 Number of models used -> 5.

The tests were conducted with 3 different LLMs : gpt-3.5-turbo, gpt-3.5-turbo16k and gpt-4. We aimed to measure the time and number of tries required to generate an acceptable C# script. The first test was conducted asking an environment composed of 5 models. In Chart 1, we can see that, for gpt-3.5-turbo, the peak of time required was around 11 seconds, whereas the lowest peak was at 6 seconds. The average amount of tries was also good, reaching 1.857 tries.

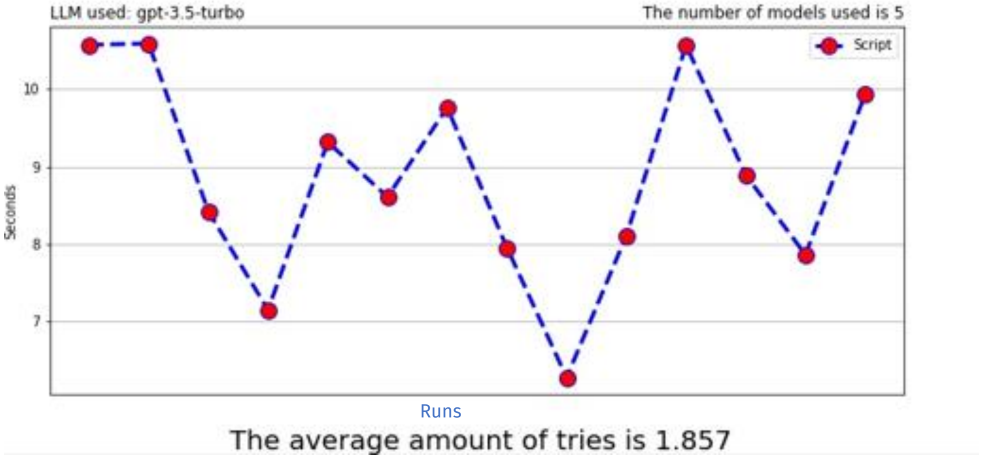


Chart 1: 5 Models using gpt-3.5-turbo.

For gpt-3.5-turbo-16k, the results were quite similar to the previous LLM. The main differences were that the lowest peak of time was 5 seconds, and the average number of tries was slightly higher, reaching 1.923 tries to generate an acceptable script.

It must be said that this LLM is trained with a higher quantity of data and the price for tokens is also higher, but its performance is almost identical compared to gpt-3.5-turbo.

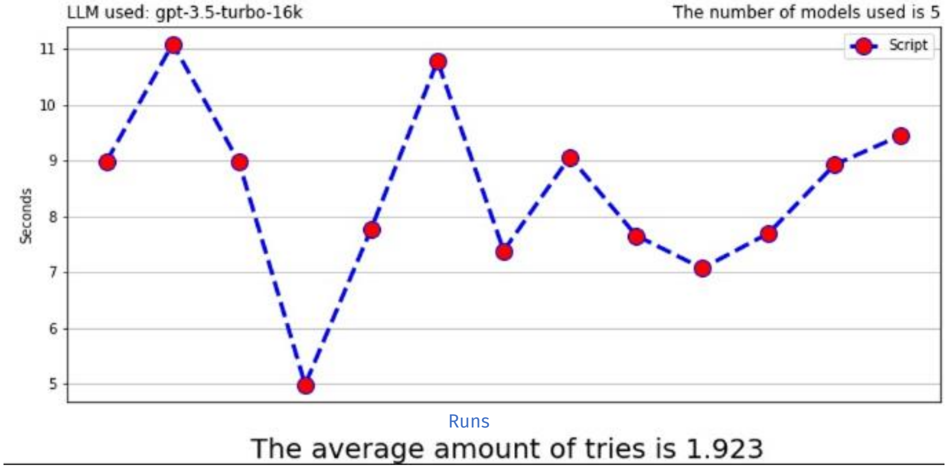


Chart 2: 5 Models gpt-3.5-turbo-16k.

In Chart 3, we can visualize that the results of gpt-4 are controversial compared to the previous one in Chart 1 and Chart 2.
The LLM's best generation time was around 19 seconds, and the worst was over 32 seconds, and it required an average of 2.692 attempts.
It is interesting to note that gpt-4 should have greater capabilities and broader knowledge but, at the same time, its price is 4.5 times higher compared to gpt-3.5-turbo.

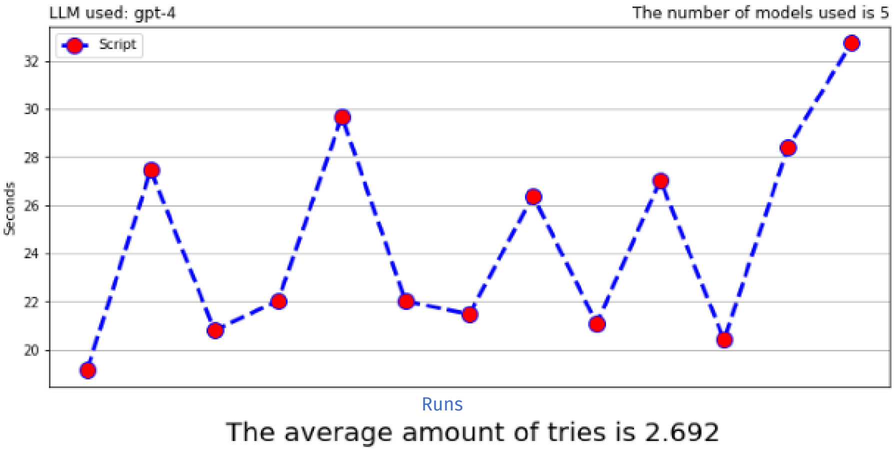


Chart 3: 5 Models using gpt-4.

6.1.3.2 Number of models used -> 15.

The second set of tests was conducted creating an environment composed of 15 models. As before, the first LLM tested was gpt-3.5-turbo (Chart 4). It is easy to guess that, because we increased the number of models, the time required to generate an acceptable script increased significantly, varying between 10 and 19 seconds. The average number of tries also rose substantially, reaching 4.643. With these results, it becomes evident that gpt-3.5-turbo encounters some difficulties in generating an acceptable script when handling a large number of objects in the environment.

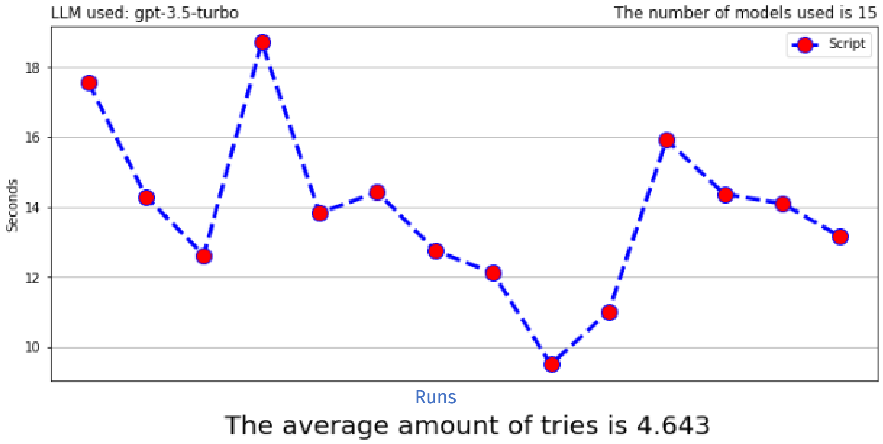


Chart 4: 15 Models using gpt.3.5-turbo.

Gpt-3.5-turbo-16k, as the previous LLM, took 12 to 19 seconds to generate a correct code (with one outlier that took 5 seconds). But, if we analyse the number of attempts, we can see that it requires only 2.615 tries, two less than gpt-3.5-turbo. With this data, we can understand the differences in performance between the two LLMs, illustrated in Table 3.

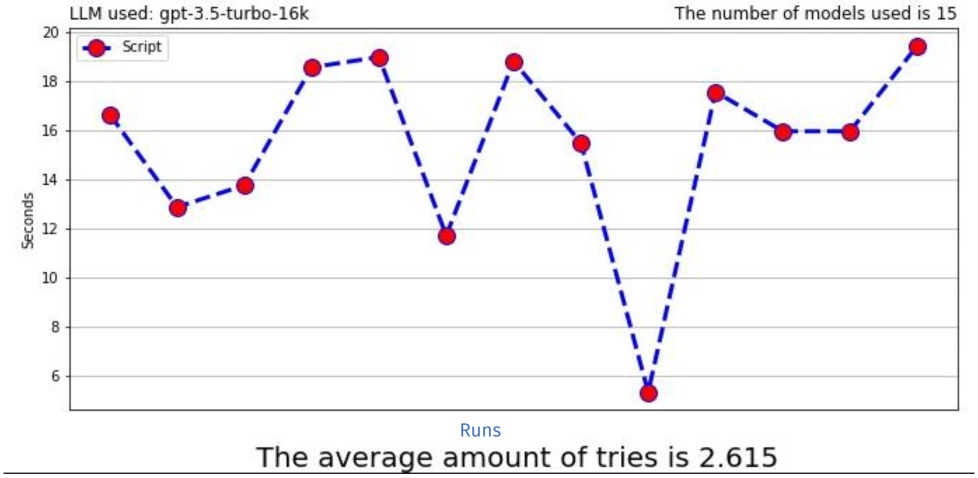


Chart 5: 15 Models using gpt-3.5-turbo-16k.

The results obtained by gpt-4 were completely different from those of the previous LLMs. The seconds required, as we have seen in the 5-model tests were extremely higher than the previous LLMs, ranging between 35 to 53 seconds, with an outlier at 18 seconds.

However, the most notable result was the average number of tries needed, which was just 1.333—half that of gpt-3.5-turbo-16k and a quarter of gpt-3.5-turbo. This test highlights the substantial differences in performance and in computing capabilities between gpt-3.5 and gpt-4.

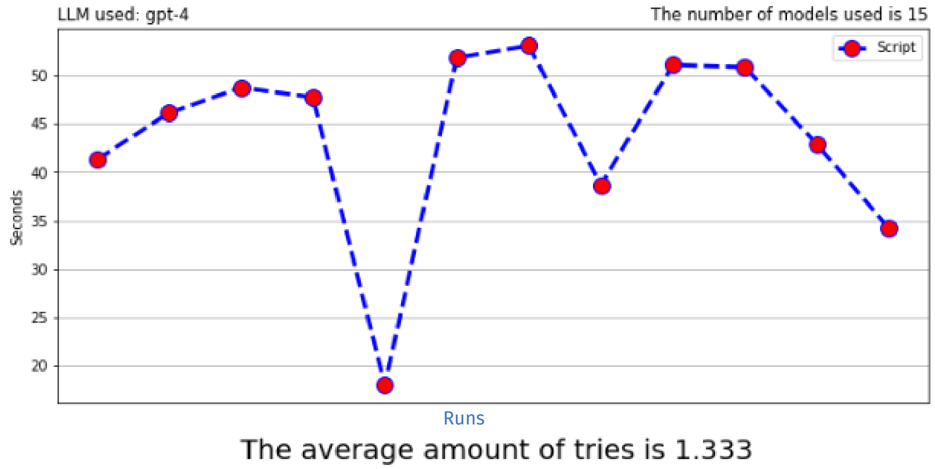


Chart 6: 15 Models using gpt-4.

6.1.3.3 Number of models used -> 30.

For this test, we incremented the number of models for just one environment to 30. For gpt-3.5-turbo (Chart 7: 30 Models using gpt-3.5-turbo. The situation was not different from the previous one, the seconds required still varied between 16 and 21 , with one outlier; the average amount of tries was slightly less than before.

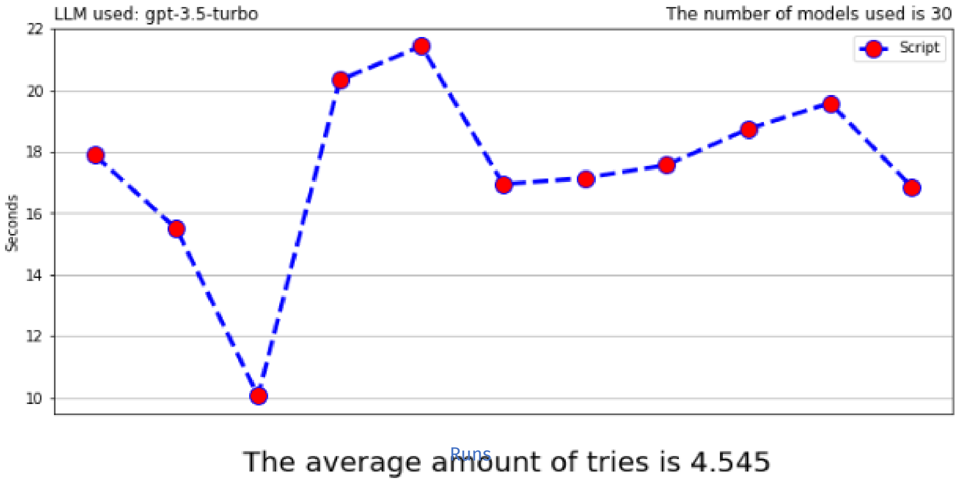


Chart 7: 30 Models using gpt-3.5-turbo.

In Chart 8, we can see the behaviour of gpt-3.5-turbo-16k with 30 models. The trend here was quite linear, varying mostly between 15 to 25 seconds, with some peaks at 32 and 35 seconds. Notably, the average number of attempts, as in the previous experiment, was lower than gpt-3.5-turbo.

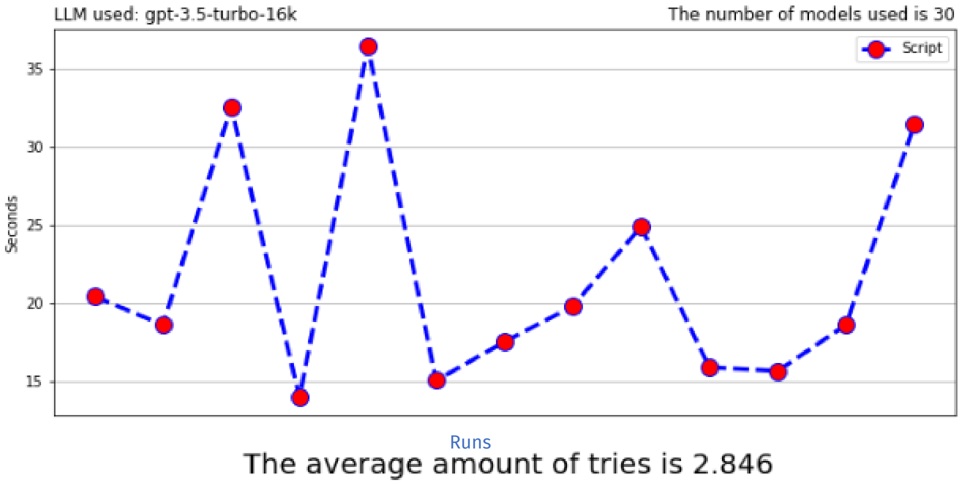


Chart 8: 30 Models using gpt-3.5-turbo-16k.

For gpt-4 the situation was not similar to the previous one which considered an environment with 15 models inside. With 30 models, the time spent to generate an acceptable C# script ranged from 62 seconds to 90 seconds, with one outlier at more than 100 seconds. Even the number of tries increased with respect to Chart 6; it almost doubled, but that was fine because we must consider that the script requested was more complex than the one generated before and so the results would be more complex.

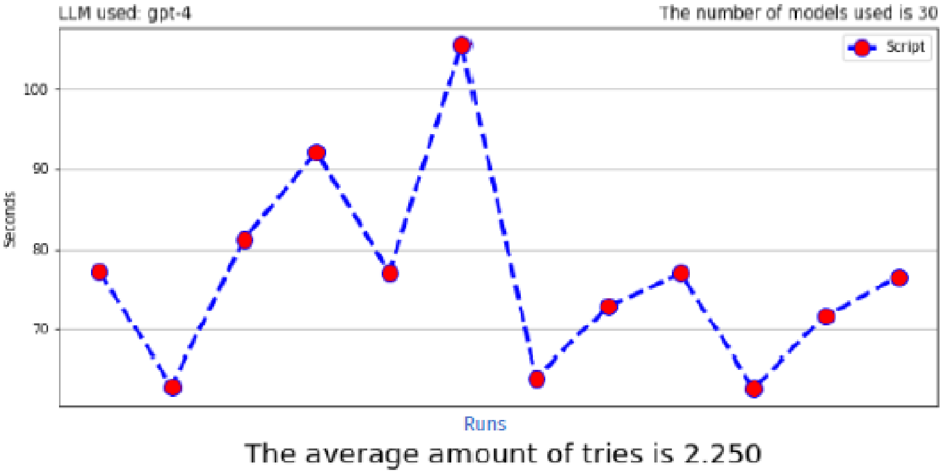


Chart 9: 30 Models using gpt-4.

6.2 User Testing

All the testing phase has been conducted by using gpt-3.5-turbo-16k.
The actual questionnaires can be found here:
https://github.com/Riz97/Master-Thesis/tree/main/Questionnaire

6.2.1 Testers Description

Firstly, we have selected 10 different users for the testing of the User Mode.
It must be said that all of these users are not related to the Computer Science world, they have jobs very different among them such as: a broker, three lawyers, a sales manager, a social media manager, a physiotherapist, a finance advisor and a sales administrator.
For the Developer Mode testing, we have selected 7 PhD students from Computer Science department of the University of Genoa and one software engineer.

6.2.2 System Usability Scale (SUS)

The System Usability Scale questionnaire was released by John Brooke in 1995 [28]. It is made up of 10 questions designed to provide a quantitative method to evaluate the usability of software and hardware.
The answers to the questions range between “I strongly agree” (5 points) to “I strongly disagree” (1 point).
The questionnaire has several advantages, starting with its quick processing time due to the concise 10-question format, its versatility for various software and the ease with which results can be calculated and applied to the system to make it better.

The survey’s statements are divided into odd and even numbered questions for computation reasons.
The formula for the SUS score is as follows: $x + y * 2.5$, where x is obtained by subtracting 5 from the sum of all points of the odd-numbered questions, and y is obtained by subtracting the sum of points of the even-numbered questions from 25.
The average SUS score is 68. A software score above this value can be considered above average and a value below 68 can be considered below average. The top score is considered to be 80.3 points, whereas scores below 51 indicate the software is placed at the bottom of the leaderboard.

6.2.3 User Experience Questionnaire (UEQ)

The User Experience Questionnaire that we are going to use, was created in 2005 [29] [34]. It is a reliable and simple questionnaire that measures the User Experience of interactive products.

Each question is represented by two terms with an opposite meaning; half of the questions start with a positive adjective and the other half with a negative adjective. The purpose of the questionnaire is to identify areas of improvement and make decisions to enhance the overall user experience.
The main topics covered are the ease of use, performance, and the design of the system.
The questionnaire considers 6 different aspects:

- **Attractiveness**: It measures the overall impression of the user; did he like it or not?
- **Perspicuity:** Was it easy for the user to become familiar with the software?
- **Efficiency:** The ability to complete tasks quickly and with minimal effort.
- **Dependability:** Does the user feel in control of what he was doing?
- **Stimulation:** Is the framework stimulating and thrilling?
- **Novelty:** Is the framework innovative and creative? Can it catch the user's attention?

The range of every scale is between -3 (horribly bad) and 3 (extremely positive).
If the value obtained, for a particular aspect, is between -0.8 and 0.8, this represents a neutral evaluation of the corresponding aspect; values greater than 0.8 represent a positive evaluation, whereas values lower than -0.8 represent a negative evaluation.

6.2.4 Questionnaires Results

6.2.4.1 User Mode

After the testing for User and Developer, we have collected all the questionnaires' answers in two different excel files for SUS and EUQ, the first one created by us, and the second one provided by UEQ creators [36].
These files can be found in the Github folders https://github.com/Riz97/Master-Thesis/tree/main/Questionnaire/EUQ and https://github.com/Riz97/Master-Thesis/tree/main/Questionnaire/SUS

Firstly, we analyse the results obtained by User Mode.
For the System Usability Scale the final score is equal to 89.75, found through the formula explained 6.2.2 and it can be evaluated as "Best Imaginable" score as it is shown in Figure 3.

This result is extremely exciting because it tells us that all of the users have well understood all the capabilities of the framework and have enjoyed their experience with it.

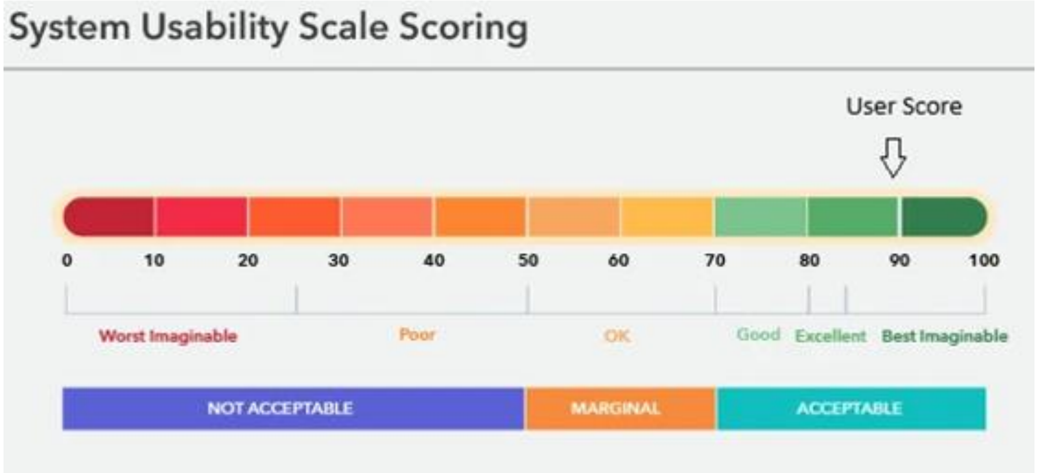


Figure 3: System Usability Scale Results for User Mode

Secondly, for User Experience Questionnaire, it is provided by the UEQ creators an excel spreadsheet where we can insert all the data obtained from the users and it outputs a useful graph for the evaluation of the result.

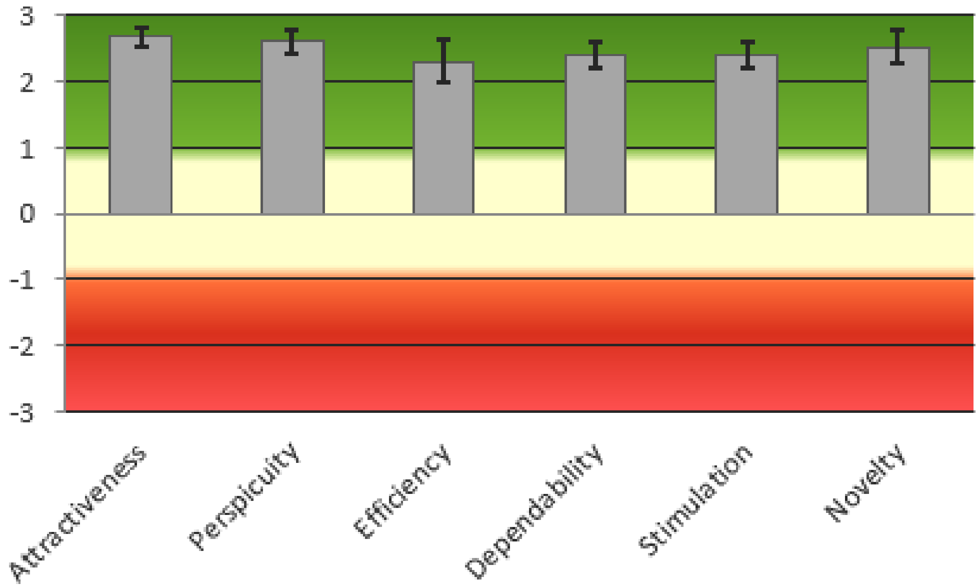


Figure 4: User Experience Questionnaire Results for User Mode

The graph evaluates perfectly all the 6 aspects considered in the questionnaire, exposed in 6.2.3.

We have to remember that if a value obtained is greater than 0.8 means that the evaluation is positive.

Therefore, for Attractiveness we got 2.683, for Perspicuity 2.600, Efficiency 2.300, Dependability 2.400, Stimulation 2.400 and Novelty 2.525.

It is not very surprising that the lowest score has been obtained in Efficiency, because the framework depends on the the time of generation for the C# Script by the gpt models and on the acceptability of the code (if a script is not well shaped for our purposes, the algorithm will continue to run until acceptable code is generated).

To sum up, we can say that the graph tells us that the user experience with the framework was convincing under every aspect.

6.2.4.2 Developer Mode

As we have done before for the User Mode, the first results that we will analyse is about the System Usability Scale questionnaire.

In this case the score got is equal to 87.85, which is lower than the previous one but still encouraging.

The high score represents the fact that the framework's usability is extremely high.

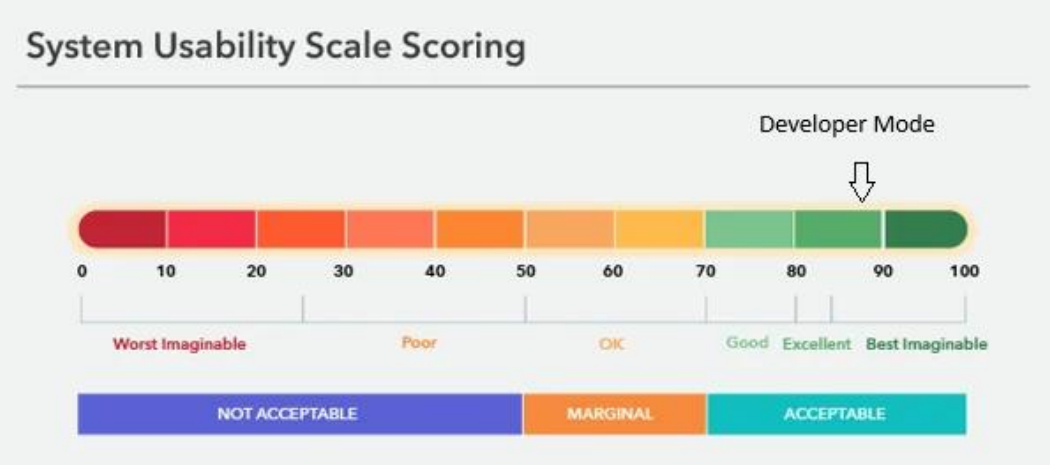


Figure 5: System Usability Scale Results for Developer Mode

On the other hand, the situation of the User Experience Questionnaire is changed, but it is still comforting.

In this case, the scores of every aspect are the following: Attractiveness 2.214 , Perspicuity 2.286, Efficiency 1.821, Dependability 2.071, Stimulation 2.321 and Novelty 2.321.

All the values are lower than before, but they still fall within the range of the positive results.

As before, the lowest record has been reached by the aspect Efficiency because the framework requires time to obtain acceptable C# code and to effectively execute it.

In the future we will work on this aspect to make the framework quicker.

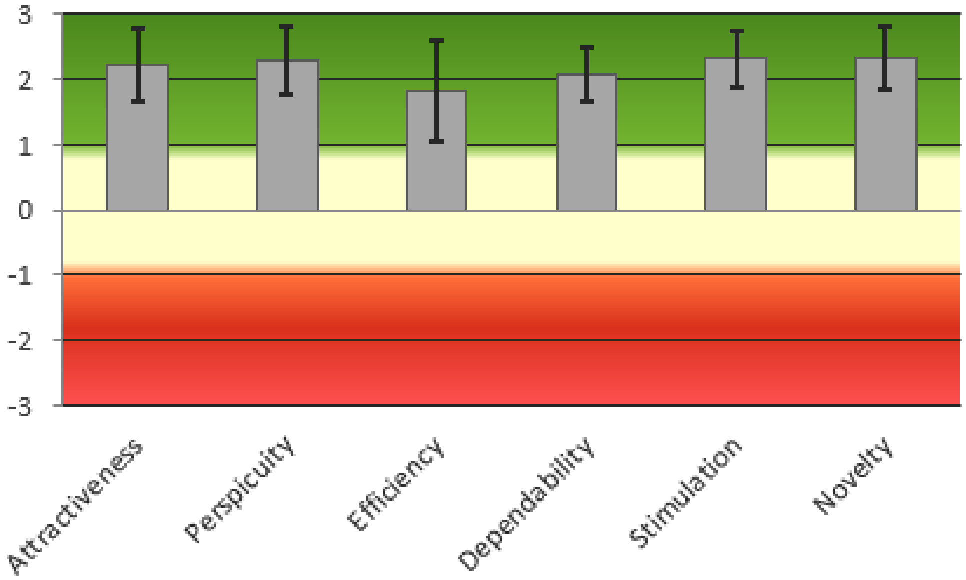


Figure 6: User Experience Questionnaire Results for Developer Mode

Chapter 7 Conclusions and Future work

In this thesis, we have presented a framework capable of creating custom virtual reality environments with the support of Large Language Models. The system was built using Unity, C#, and different APIs to connect to the LLM and the runtime compiler.
We have implemented two different modalities tailored to different types of users: “User Mode” and “Developer Mode”. Each modality offers two distinct “game modes” called Prebuilt-Environment and Free Models. With the former, the user can choose a list of different environments that simulate real world rooms or environments while with the latter, users can customize their environments by choosing how many models they want and positioning each object individually.

Then, we tested the framework with 3 different Large Language models: gpt-3.5-turbo, gpt-3.5-turbo-16k and gpt-4, considering 3 different sets of model quantities (5, 15, 30). We also conducted user testing with 20 participants: 10 without any computer science background and 10 with a background in the field.
The project, obviously, has room for improvement, and we plan to address various issues in the near future to enhance system stability, reduce errors, and add new features. To make the system more stable and avoid errors created by IA, we have to generate every type of possible C# faulty script, then list all the errors made by the LLM and then add new acceptability checks and filters in the code. We must point out that the number of different errors is not so low, thus this task will be time consuming and expensive (because of the costs of the API).

The first improvement which the system could greatly benefit from is the usage of a 3D asset database such as Sketchfab [30]. This would allow users to download any 3D models required for their session, without downloading and saving them previously in the Unity project.
Of course, implementing this feature would require formulating a new request text to send to the OpenAI API.
Another improvement for the system could be giving the users the option to choose which Large Language Model they wish to use. In addition to this, we would like to insert other LLMs that are not related to OpenAI, such as: Google’s Gemini [31], Meta’s LLaMA 3 [32], and NVIDIA’s NeMo [33].

This improvement is strictly linked to the possibility that exist APIs of the LLMs previously mentioned for Unity. Of course, before introducing new Large Language

models, we should conduct some tests like the ones done in 6.1.3 in order to guarantee efficiency and performance.
This improvement depends on the availability of APIs for these LLMs that are compatible with Unity. Before introducing new LLMs, we need to conduct tests similar to those described in section 6.1.3 in order to ensure efficiency and performance.
Lastly, we plan to add two new buttons and the virtual reality keyboard to the system. One button will allow users to export the environments they have created. For example, if a user built an environment related to the Nature or Industry macro category, he can now use them in his own project.
The second one is similar to a reset button that will set the system back to what it was like at the starting point of the application, with no empty Models inside of it. This feature could be very useful in two different scenarios. The first one is when the LLM takes too long to generate an acceptable C# script which causes the system to freeze and no longer accepts any new requests. The other problematic situation, which rarely happens, is when the LLM exceeds the maximum allowed time to generate a script, resulting in the script not being accepted even if it is correct. In such cases, the framework would need to be restarted.

These planned improvements aim to make the system more robust, user-friendly, and versatile, ultimately enhancing the overall user experience and expanding the framework's capabilities.

Acknowledgments

Bibliography

[1] Morton Heilig. The cinema of the future. 1955.

[2] I. E. Sutherland, "Sketchpad-A Man-Machine Graphical Communication System", Proceedings of the Spring Joint Computer Conference, Detroit, Michigan, May 1963 (Washington, D.C.: Spartan, 1964).

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin “Attention Is All You Need” , June 2017.

[4] OpenAI. https://openai.com, 2024.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei , “Language Models are Few-Shot Learners” , July 2020.

[6] ChatGPT. https://chat.openai.com , 2024.

[7] Jasmine Roberts, Andrzej Banburski-Fahey, Jaron Lanier, “Steps towards prompt-based creation of virtual words”, November 2022.

[8] Daniele Giunchi,Nels Numan, Elia Gatti, Anthony Steed, University College London UK, “DreamCodeVR: Towards Democratizing Behavior Design in Virtual Reality with Speech-Driven Programming”, 2024.

[9] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell (200 additional authors not shown), “GPT-4 Technical Report”, March 2023.

[10] Meta Platforms Inc. https://www.meta.com, 2024.

[11] Meta Quest Link Complete Prerequisites. https://www.meta.com/it-it/help/quest/articles/headsets-and-accessories/oculus-link/requirements-quest-link, 2024.

[12] Roslyn C# - Runtime Compiler. https://assetstore.unity.com/packages/tools/integration/roslyn-c-runtime-compiler-142753, 2024.

[13] Ubiq. https://vr.cs.ucl.ac.uk/ubiq, 2024.

[14] Ubiq-genie. https://github.com/UCL-VR/ubiq-genie?tab=readme-ov-file

[15] Nels Numan, Daniele Giunchi, Benjamin Congdon, Anthony Steed, “Ubiq-genie: Leveraging External Frameworks for Enhanced Social VR Experiences”, University College London, 2023.

[16] API OpenAI-DotNet. https://github.com/RageAgainstThePixel/OpenAI-DotNet , , 2024.

[17] OpenAI Tokenizer: https://platform.openai.com/tokenizer , 2024.

[18] Meta XR All-in-One SDK. https://assetstore.unity.com/packages/tools/integration/meta-xr-all-in-one-sdk-269657, 2024.

[19] Hugging Face API Unity. https://github.com/huggingface/unity-api

[20] Low-Poly Simple Nature Pack, https://assetstore.unity.com/packages/3d/environments/landscapes/low-poly-simple-nature-pack-162153, 2024.

[21] Furniture Asset Pack https://assetstore.unity.com/packages/3d/props/furniture/glassofcoins-furniture-asset-pack-200983, 2024.

[22] Tiny Low Poly Cars, https://assetstore.unity.com/packages/3d/vehicles/land/tiny-low-poly-cars-180034, 2024.

[23] City Props Pack! https://assetstore.unity.com/packages/3d/vehicles/land/tiny-low-poly-cars-180034, 2024.

[24] Industrial Models, https://assetstore.unity.com/packages/3d/props/industrial/industrial-models-171071, 2024.

[25] 15 Original Wood Texture, https://assetstore.unity.com/packages/2d/textures-materials/wood/15-original-wood-texture-71286, 2024

[26] Asphalt Materials, https://assetstore.unity.com/packages/2d/textures-materials/roads/asphalt-materials-141036, 2024.

[27] Stylized Grass Texture, https://assetstore.unity.com/packages/2d/textures- materials/glass/stylized-grass-texture-153153, 2024.

[28] John Brooke, "SUS: A quick and dirty usability scale", November 1995

[29] Bettina Laugwitz, Theo Held, Martin Schrepp Construction and Evaluation of a User Experience Questionnaire, November 2008

[30] Sketchfab.com , https://sketchfab.com

[31] Google Gemini, https://gemini.google.com/?hl=it, 2024.

[32] Meta Llama 3, 2024. https://llama.meta.com/llama3/, 2024.

[33] NVIDIA NeMo, https://www.nvidia.com/it-it/ai-data-science/products/nemo/, 2024.

[34] Martin Schrepp, "User Experience Questionnaire Handbook", September 2015.

[35] Unity Technologies. Unity Engine. https://unity.com/ , 2024.

[36] User Experience Questionnaire. https://www.ueq-online.org , 2024.