***User Manual - New Framework Based on Large Language Models for Creating Customized Virtual Environments*** Riccardo Caprile Università degli Studi di Genova DIBRIS

In this manual I will explain what have been changed in the framework since 1st October 2024 and it can modified or improved, every other information can be seen in the Thesis pdf downloadable [here](https://unire.unige.it/handle/123456789/9166?show=full).

**UI Interface – User and Developer Mode**

**Immagine che contiene testo, schermata, Carattere, logo

Descrizione generata automaticamente**

The user has now the possibility to choose more Large Language Models when is wearing the headset and not by changing the LLM directly in Chat.cs.

In the panel Server Connections Buttons there all the buttons that make possible the connection to the Python Server for the respective LLM. So, if the user wants to use the Google Gemini LLMs, firstly he has to click the button with the Google image in the panel and he will be connected to the server; for the other LLMs server button is the same process.

On the right, there is a dropdown menu called “Large Language Model Selection”, where the user can actually choose he wants to use for a particular virtual environment. He can choose : GPT (available for the thesis release), GEMINI (gemini1.0 – gemini1.5) , Meta (Llama3.1), Codex(gpt4o-mini), Qwen (Qwen2.5-coder), Codegeex (codegeex4), Codellama (codellama).

It has also been added the button “Reset”, in this way if there problems with the code generation and execution the user can reset the scene to the starting point.

These modifications are available for both modes : User and Developer.

**SCRIPTS EXPLANATION**

**Chat.cs**

**Start() ->** it has been added all the code necessary to manage all the different LLMs through the Python servers, the code is the same for all the LLMs. Basically, the code generated by the LLMs is awaited and then stored in the variable *result\_aux*. This variable is then cleaned from all the “non-code” words through two different methods : *RemoveTextBeforeUsing()* and *TrimAfterLastBrace*; with these methods we get rid of all the useless words before the fist “using” (which is the real start of the script that should be executed) and all the words after the last ‘}’ (which should be the end of the script). Then is called the method *AIList()*  and the number of tries is incremented by one. This process is the same for all the newly added LLMs.

**AIList(result,firstNonwithSpaceChat,Number\_of\_Objects,start) ->** It checks that the script can be actually executed by doing some sanity checks.

1. The AI generated script must contain all the words in the *Mandatory\_Words* list
2. The AI generated script must contain at least one words from the *Material\_Words* list.
3. The first character of the script must be a ‘u’
4. It must contain at least two object’s name stored in *All*  list.
5. The substrings Nature or Furniture or Industry or Cars or City must be contained at least 2 times in the script.

If all the checks are good, the script is accepted and the *input* variable is set “STOP”. In this way no more requests are sent to the LLM, because we achieved the desired goal.

**ReadStringInput(TMP\_InputField) ->** It has been changed the way that all the objects are destroyed when the button “Generate Script” is clicked. We destroy all the objects that contain in the name “Clone” and “Model”. In this way, if after the execution of the AI generated script there are some “clones” they are correctly deleted. The input and coordinates management has not been changed and it is the same as before.

**Domain.cs ->** We have now a different handling of the scripts that contains a syntactic errors or Unity Exceptions. Basically we save , as before , the total number of attempts required to AI for generating an acceptable but , now, we do not accept scripts that contain Unity Exception, and the number of the so called “Faulty Scripts” is saved in the variable *FaultyScriptCount* and the Faulty script is saved inside the text file FaultyScript.

**OnLogMessageReceived ()** -> Basically, if an error or exception is raised in the Unity Console, we have to execute again the LLM request by calling the method *CodeErrorExecution().* The variable *errorcount* is necessary in order to execute the code only one time, otherwise the code would be executed the number of times equals to the number of errors raised in the console.

**CodeErrorExecution()** -> It sets the flag *IsExecutable* to false, because that script cannot be saved inside the LogFile, so It is inserted in the FaultyScript file. Then, the number of FaultyScript generated is increased by one and a pop up appears in the UI to notify the user that the requests is sent again. Then, we called the method *ReadStringInput()* and *DoScript()* for a new LLM request.

**IEnumerator WaitIA()** -> Here, we set Roslyn as before, and if the script can be executed correctly we create the log file and we reset all the counters. Thanks to a while cycle there is no need to have a timer that put the script in pause. In this way the output\_text is constantly checked and if the acceptable code generated is printed in the window, the script can be actually executed.

**CreateLogFile()** -> We track the number of faulty scripts generated for that particular virtual environment, thanks to the counter *FaultyScriptCount.* At the of the method all the counters are reset.

**CreateFaultyScriptFile** ->

**\*NetworkManager.cs**

**PYTHON SERVERS**

**UNITY HIERARCHY**