College of Engineering

COMP 491 – Computer Engineering Design Project

Final Report

Interactive AI Driven Role-Playing Game (INAID-RPG)

Participant information:

Emir Şahin	72414
Kaan Dai	71651
Cem Ozan Doğan	72623
Oğuzhan Şanlı	72126

Project Advisor

Sajit Rao

Spring 2024

Table of Contents

Contents

The sections of this report were written by:

Emir Şahin:
- Section II.
- Section III.I and its subsections; I, II, III and IV.

Kaan Dai: Section
- Section I.
- Section III.III and its subsection; I, II, III.
- Section IV

Cem Ozan Doğan:
- Section III.II and its subsections; I, II, III and IV.
- Section IV

Oğuzhan Şanlı:
- Section III.II and its subsections; II, III.
- Section IV.
- Section V.

I. Abstract

Immersiveness has always been a key element in game development, with the rising popularity, convenience and success of AI models (Language models in particular), it is only a matter of time before they are integrated into game development -whether it be in a small scale or as the entire premise-. Our plan is to integrate AI models into game development in a scale and concept we saw fit that serves to prove that such games are more immersive. We decided the most appropriate genre to demonstrate the effect of AI models on immersiveness would be role-playing games (RPGs), as dialogue is an important part of an RPG and an RPG is not restrictive with regards to its story, these 2 points are important to our plan as our plan is to create a game where the story is driven by an AI model to keep the story interesting for the player and where the companions of the player are AI models that communicate and interact with the player. We chose a 2D environment which presents an adequate medium without restricting it to a text-based process, which we thought would negatively impact immersiveness.

II. Introduction

The main goal of our project was to showcase the increased immersion provided by incorporating AI-models into video games. We approached this problem in two angles, our main strategy was the companion model, now I will explain the companion model in a bit more detail. To display the increased immersion, we decided to have 3 companion NPCs (non-playable characters) in our game that would follow and aid the player through the game, which is a common concept in video games in general. In current video games, the player can converse with the NPCs by choosing from several static responses pre-made by the game developers, and the NPC would then respond with the pre-arranged response to the dialogue option chosen by the player. A consequence of this situation is the typical “I wish I could’ve said exactly what I am thinking” response from the player. Our companion model approach allows this by assigning an AI-model to the companions in the game, specifically, we started by using the gpt-3.5-turbo model, occasionally tested the gpt-4 model and completely switched to the gpt-4o model after its release. Our reason behind having 3 companions is to display the difference in responses from the AI-models depending on the personality we’ve assigned to the companion and depending on the relationship of the companion between the player and the companion which dynamically changes throughout the game. An important remark is that we update the companion models on what is going on in the game environment regularly; when an important event occurs or if nothing important has happened, periodically. This information update given to the companions include many things such as the

distance between the companion and the player, the companion's item inventory, the current scene, the current room, the enemies in the room, the current task, etc. I will be getting into more detail when appropriate. We chose Unity as our game engine for the project as its modular structure allowed us to build something that hasn't been done before from the ground up and its extensive open-source libraries, which we used greatly. By assigning AI-models to the companions, we've allowed the player to say whatever they desire to the companions without being restricted by preset dialogue choices and responses. We've further solidified the interaction between the companion models and the player by implementing a system through which the player can give commands to the companions, which the object representations of the companions within the game then proceed to do. A benefit of this system is that the command given by the player is interpreted by the AI-model, taking the surroundings, situation and the task of the player and the companions into account. Another important remark is that to not damage the immersion, which is our essential goal, we've added voice chat with the companions. The player can converse and give commands to the companions using their voice. The companions' responses however are text only. We've chosen OpenAI's Whisper as our speech to text model, as gpt-4o was not yet released at the time we were implementing this feature and the procedure of talking to the companions in the game had heavily been intertwined with the rest of the game's scripts, meaning we would have to remove and then re-implement a considerable amount of functionality to make the switch from Whisper. In addition, at the start of the project, our API TPM limits were very low, therefore we had to heavily optimize our API calls to adequately update the companions and facilitate the dialogue between the player and the companions, utilizing Whisper instead of increasing the number of tokens we're sending to a single model increased the TPM allocated to the player. As we kept using the OpenAI API however, our TPM limit has increased to a stage where we would not reach our limit even if our game was running simultaneously on 20 devices (With the same API token of course). Our secondary strategy was to have another AI-model -that we only update when a significant event occurs, such as a room or a scene change- that narrates the environment, situation and the task of the player. We called this model the dungeon master model. When the player enters a new scene or a room, the response from this model given the current situation and the environment of the player is displayed at the top of the screen. The dungeon master uses OpenAI's gpt-3.5-turbo model.

We could not find any products or projects online with a similar principle. One project that offered us guidance as to the feasibility of our project was a custom user-made modification to the game The Elder Scrolls V: Skyrim, the mod added a companion character to the game that the player could talk to, the modification is titled "Herika - The ChatGPT Companion"[1].

III. System Design

III.I. Companion Functionality

Our initial approach in companion model development was to create our own model instead of using an API. We thought this approach was feasible due to the fact that our model did not need to have many of the faculties that the GPT models possess, such as being able to solve complex mathematical problems, generate code or have extensive knowledge on an immense number of different subjects. We started by following a tutorial[2] and building a bigram model, and then a proper transformer model, using 2 children's books to train the bigram model (specifically "The Wizard of Oz" and "Alice in The Wonderland") and use a large dataset called "openwebtext"[3] to train the proper transformer structure model. Our aim was to familiarize ourselves with the tools used in building a model and then start building the model. There are several benefits to building our own model instead of using an API; the API needs an internet connection, the API has a cost, we control the training data of our own model, making it more efficient and making it less likely to deviate from what we consider to be appropriate responses for our use case. But after a short development period we decided that training our own model would far exceed our time constraints and would definitely hinder the amount of functionality we had set out to implement. We had always planned for the dungeon master model to use an API; but for the companion model, the switch to API happened later during the project development. In the further sections of the report, -the details of the companion model development-, I will not be going through the history of our development process unless it is relevant, but I will rather explain the project as it is in the final version.

I will be going over the basic functionality briefly, to provide an overview, and I will be explaining the functionality in more detail in the relevant subsections going forward. If the player wants to use their voice to communicate with the companion models, they need to hold down the "V" button, as long as the button is being held down the voice of the player is being recorded through the microphone they've chosen, the player has 4 seconds (This is a value we have set that we believe works well) to record their message, the time spent recording is displayed at the bottom left of the screen as a circle filling up clockwise. If the time limit of 4 seconds is exceeded, the message will be considered as concluded and the recorded sound will be used. The player chooses their microphone in the pause screen of the game, it is a very simple pause screen that pauses the game, allows the player to switch between their available microphones and resume the game. The voice is recorded as a .wav file, sent to the speech-to-text model using Whisper API, the response from Whisper is then incorporated into the prompt sent to the companion model. The player can

choose to speak using their voice to any of the 3 companions, or all of them at once, using a selection wheel that appears at the bottom left of the screen as long as the player holds down the Space key, when the player makes their choice as to who they want to talk to using their mouse, they can let go of the Space key to lock in their selection, next time the player presses V to speak to a companion, their message will be sent to the model of the companion they have chosen. While holding down V to record a message, the player can press the B key without letting go of the V key to discard the message being recorded. If the player wishes to write their message instead of recording their voice, there are 3 buttons on the top left of the screen, labelled after the companion they correspond to. When the player clicks on one of these buttons, the game pauses and a chat box is displayed covering almost the entire screen. Through this screen the player can again record their voice message, by either holding down the V button and letting go when they're done, or by pressing the displayed record button to start recording and press the button again to stop recording. When the chat box is open, the voice recording timer is displayed differently, on a bar that fills up from the bottom to the top on the right side of the chat box. Most importantly this chat box has a text input field that allows the player to type out their message and send it using the send button. This chat box serves another purpose, as a dialogue history; whether the player used their voice to speak to a certain companion without opening the chat box or if they typed it out, the conversation between the player and the companion is saved inside this chat box, the player can scroll within the message area to view their previous messages and the companions' responses. Moving on from the UI, there are 3 types of messages sent to the companion models, an initial System prompt that sets the behaviour of the companion models, this prompt includes many information that needs to be sent to the model that does not change, or the initial state of the information that will be updated later on, this is the longest prompt, provided the player does not send an especially long message to the model. The other message type is the simple completion response sent by the player to the model, if the player's message is interpreted as a command by the model, the response to this message additionally includes a JSON object, which is how the commands given by the player are transferred and interpreted in the game engine. Interpreting the commands this way allows the player to give out a command to the companion without needing to be too specific, there are not few key phrases that are registered as commands, the model can make inferences from the message and the context to decide whether to interpret a message from the player as a command or not, and as to which command it will be registered as. The final type of message is the system update, these are the regular updates sent to the model to update it on the current situation, task, inventory and surroundings of the player. This is the basic non-linear walkthrough of the companion system in the game, more detailed explanation will follow.

A final remark, we have written many scripts, almost half of them are less than a page long, we will be displaying the entire script for the ones that are less than a page, but only important and relevant sections for the scripts that are longer than a page.

III.I.I. Companion UI

There are several UI elements that support the communication with the companion models. Starting with the ones that are immediately visible to the player at the start of the game:

Figure 1 – The three toggle buttons each corresponding to the chat box of a companion (on the left) and the pause button (on the right).

When the player presses the pause button, the pause menu appears where the player can choose which microphone they wish to use for the recording. Here is the pause menu:

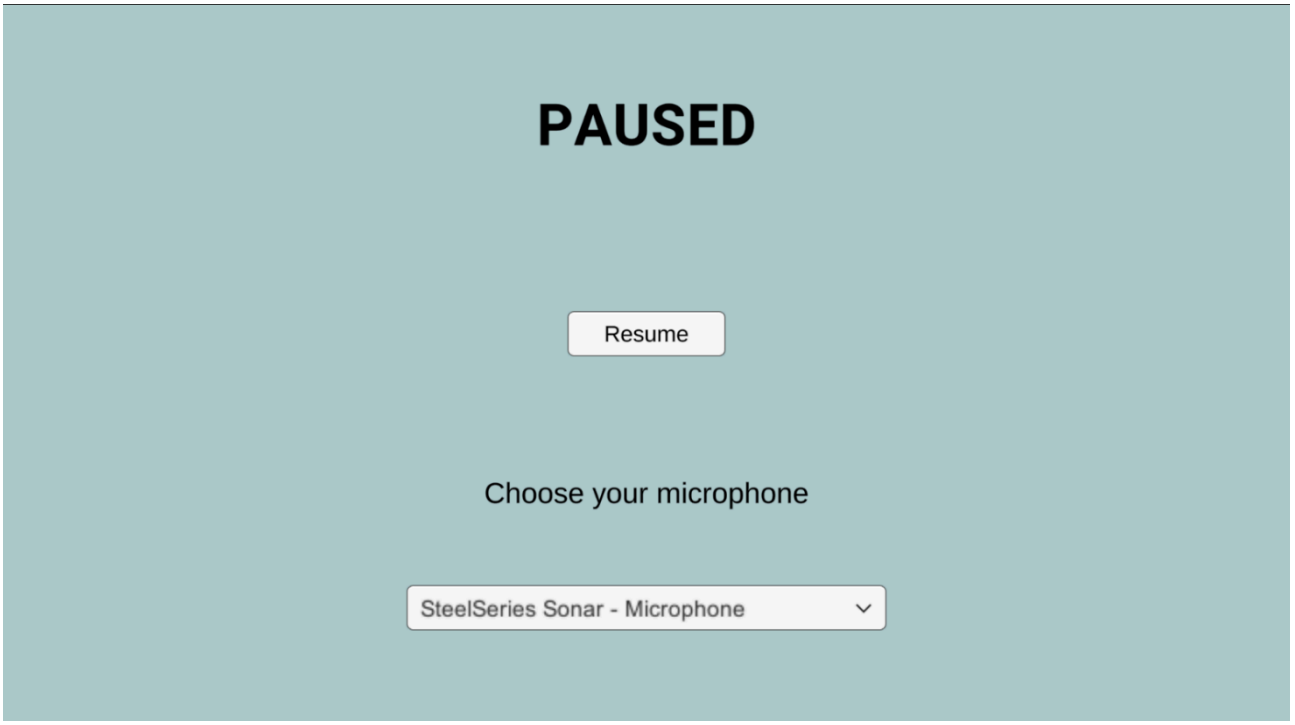

Figure 2 – The pause menu, where the player can choose their microphone and press the "Resume" button to resume the game.

The microphone selection button is a dropdown that when clicked on will display the available microphones as a list, as follows:

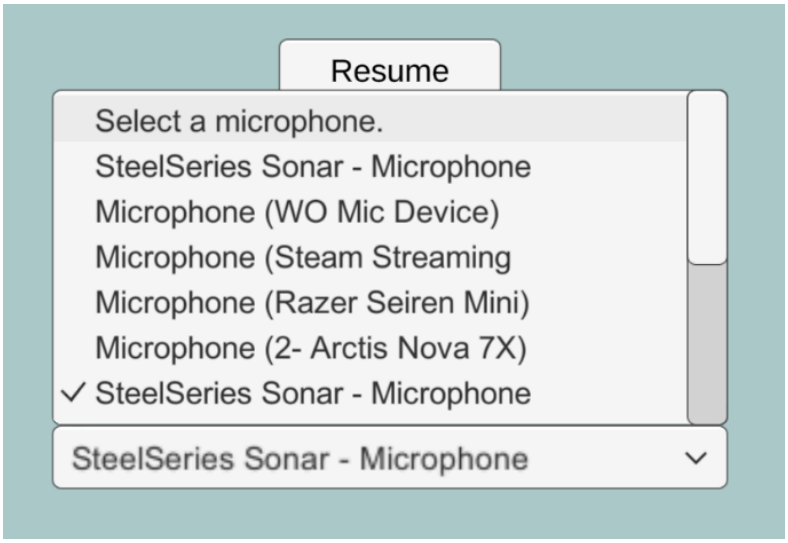

Figure 3 – The dropdown menu displaying the available microphones.

When the pause menu is pressed or one of the chat boxes are toggled on the game is paused, this functionality is facilitated by the ToggleCanvas.cs and Pause.cs classes. Here is the ToggleCanvas.cs class:

```
using UnityEngine;
using UnityEngine.UI;

public class ToggleCanvas : MonoBehaviour
{
    public Canvas canvasToToggle;
    public Canvas canvasSwitch1;
    public Canvas canvasSwitch2;
    public Canvas pauseCanvas;
    [SerializeField] private WhisperCaller whisperCaller;
    [SerializeField] private int chatChooseNumber;
    private Button button;

    private void Start()
    {
        canvasToToggle.gameObject.GetComponent<Canvas>().enabled = false;
        button = GetComponent<Button>();
    }

    public void Update()
    {
        if (whisperCaller.GetBoolForToggle() && whisperCaller.GetWhisper() !=
chatChooseNumber)
        {
            button.interactable = false;
        }
        else {
            button.interactable = true;
        }
    }

    public void ToggleCanvasVisibility()
    {
        if (chatChooseNumber != -1)
        {
            whisperCaller.SetWhisper(chatChooseNumber+1);
        }
        canvasToToggle.gameObject.GetComponent<Canvas>().enabled =
(!canvasToToggle.gameObject.GetComponent<Canvas>().enabled);
        if (canvasSwitch1.gameObject.GetComponent<Canvas>().enabled)
        {
            canvasSwitch1.gameObject.GetComponent<Canvas>().enabled = false;
        }
        if (canvasSwitch2.gameObject.GetComponent<Canvas>().enabled)
        {
            canvasSwitch2.gameObject.GetComponent<Canvas>().enabled = false;
        }
        if (pauseCanvas.gameObject.GetComponent<Canvas>().enabled)
        {
            pauseCanvas.gameObject.GetComponent<Canvas>().enabled = false;
        }
    }
}
```

Every UI element in Unity is encased within a canvas object, this script allows the game to stay paused as the chat box canvases are switched, if the player chooses to switch between two canvases without closing the initially opened canvas. This method also sets the voice chat to whichever companion's canvas is open at that time. And here is the Pause.cs class:

```
using UnityEngine;

public class Pause : MonoBehaviour
{
    public Canvas canvas1;
    public Canvas canvas2;
    public Canvas canvas3;
    public Canvas pauseScreen;
    [SerializeField] private CircleSelector ccc;
    [SerializeField] private WhisperCaller whisperCaller;

    public void TogglePause()
    {
        if (canvas1.gameObject.GetComponent<Canvas>().enabled ||
canvas2.gameObject.GetComponent<Canvas>().enabled ||
canvas3.gameObject.GetComponent<Canvas>().enabled ||
pauseScreen.gameObject.GetComponent<Canvas>().enabled)
        {
            ccc.SetKeyEnabled(false);
            ccc.DisplayCircle(false);
            whisperCaller.SetCircleActive(false);

            PauseGame();

        }
        else
        {
            ResumeGame();
            if (Input.GetKey(KeyCode.V))
            {
                whisperCaller.SetCircleActive(true);
            }
            whisperCaller.SetCircleActiveBool(true);
            ccc.SetKeyEnabled(true);
        }
    }

    void Update()
    {

    }

    void PauseGame()
    {
        Time.timeScale = 0;
    }

    void ResumeGame()
    {
        Time.timeScale = 1;
    }
}
```

This class pauses and unpauses the game and disables certain functionalities such as voice chat and choosing which companion to talk to using the chat chooser circle. When one of the toggle buttons is pressed, the following chat box is displayed:

Figure 4 – Chat box of one of the companions. The conversation area is currently empty. The light green bar on the left is the display for the recording time when the chat box is open. At the bottom of the page is the input field for the user to enter their message, and the two buttons "Send" and "Rec" allowing the user to send the message they have typed and start/stop their voice recording respectively.

Some of the UI elements are from the OpenAI Unity Package, which is made by Sercan Altundaş[4], we have used his tutorial in making the message area, text field and the send button of the chat box. His OpenAI package also includes a simple framework for calling a completion response from the OpenAI API through Unity.

The "Chat Chooser Circle" as we call it, allows the player to choose which companion they want to talk to without having to remove their hands from the keyboard and mouse, which helps the immersion as the player will not have to move their mouse to a certain button to switch between the companions they want to speak to. We did this by including the chat chooser circle. When the user holds down the space key, a circle divided into 4 sections is displayed at the bottom left of the screen and the game slows down to 10% speed. The player can simply move their mouse in one of the 4 directions, top-left, top-right, bottom-left, bottom-right (The player moving their mouse slightly in a direction is enough for it to be registered by the chat chooser circle). The top-left section corresponds to all companions at once, the top-right section corresponds to the samurai, the bottom-right section corresponds to the mage and the bottom-left section corresponds to the archer. When the player lets go of the space key, the choice is registered and relayed to the relevant classes. Here is a section of the chat chooser circle script that is responsible for tracking the mouse movement direction and setting which companion the player will talk to:

```
private void Update()
{
    if (EventSystem.current.currentSelectedGameObject == inputField1.gameObject ||
EventSystem.current.currentSelectedGameObject == inputField2.gameObject ||
EventSystem.current.currentSelectedGameObject == inputField3.gameObject)
    {
        keyEnabled = false;
    }
    else
    {
        keyEnabled = true;
    }

    if (keyEnabled)
    {
        if (Input.GetKey(KeyCode.Space))
        {
            whisperCaller.SetSpeakable(false);
            Time.timeScale = 0.1f;
            DisplayCircle(true);

            Vector3 currentMousePosition = Input.mousePosition;
            if (currentMousePosition != lastMousePosition)
            {
                Vector3 difference = currentMousePosition - lastMousePosition;
                if (difference.magnitude > 20)
                {
                    Vector2 direction = currentMousePosition - lastMousePosition;
                    lastMousePosition = currentMousePosition;
                    if (circleAble)
                    {
                        DetectDirectionGroup(direction);
                    }

                }
            }
        }
        else if (Input.GetKeyUp(KeyCode.Space))
        {
            whisperCaller.SetWhisper(selectedSection);
            DisplayCircle(false);
            Time.timeScale = 1;
            whisperCaller.SetSpeakable(true);
        }
    }
}
```

The direction is determined, and the selected section is updated. The direction calculation is done in a different method:

```
private void DetectDirectionGroup(Vector2 direction)
{
    float angle = Mathf.Atan2(direction.y, direction.x) * Mathf.Rad2Deg;

    if ((angle > 95 && angle < 175))
    {
        selectedSection = 0;
        HighlightSection(0);
        UnhighlightSection(2);
        UnhighlightSection(1);
        UnhighlightSection(3);
    }
    else if (angle < -5 && angle > -85)
    {
        selectedSection = 2;
        HighlightSection(2);
        UnhighlightSection(1);
        UnhighlightSection(0);
        UnhighlightSection(3);
    }
    else if (angle < -95 && angle > -175)
    {
        selectedSection = 3;
        HighlightSection(3);
        UnhighlightSection(1);
        UnhighlightSection(0);
        UnhighlightSection(2);
    }
    else if ((angle < 85 && angle > 5))
    {
        selectedSection = 1;
        HighlightSection(1);
        UnhighlightSection(0);
        UnhighlightSection(2);
        UnhighlightSection(3);
    }
}
```

Highlighting a section enlarges it so the player can easily tell which section they have currently selected. Here is an image of the chat chooser circle:

Figure 5 – The Chat Chooser Circle. Currently the top-left quadrant is highlighted, meaning if the player were to let go of the space key, their recorded messages would be sent to all the companions.

If the player does not make a selection, the class defaults to position 0, meaning the message would be sent to all companions. After a selection has been made, when the player presses the V button, the WhisperCaller.cs class starts working, here is the relevant section from the class:

```
if (speakable)
{
    if (Input.GetKeyDown(KeyCode.V))
    {
        whispers[0].chatGpt.SetActiveMessage(true);

        boolForToggle = true;
        recordingComplete = false;
        whispers[0].GetRecButton().interactable = false;
        whispers[0].GetChatGPT().GetButton().interactable = false;
        whispers[1].GetRecButton().interactable = false;
        whispers[1].GetChatGPT().GetButton().interactable = false;
        whispers[2].GetRecButton().interactable = false;
        whispers[2].GetChatGPT().GetButton().interactable = false;
        ccc.SetCircleAble(false);
        ccc.SetKeyEnabled(false);
        if (activeCircle)
        {
            circleCanvas.gameObject.SetActive(true);
        }

        if (selectedSection == -1)
        {
            whispers[0].GetChatGPT().SetCompanionDisplay(true);
            whispers[0].StartRecording();
            whispers[0].SetMulti();
            whispers[1].GetChatGPT().SetCompanionDisplay(true);
            whispers[2].GetChatGPT().SetCompanionDisplay(true);
        }
        else {
            whispers[selectedSection].GetChatGPT().SetCompanionDisplay(true);
            whispers[selectedSection].StartRecording();
        }


    }

    else if ((Input.GetKeyUp(KeyCode.V) && !recordingComplete) ||
recordingComplete)
    {
        if (!recordingComplete)
        {
            if (selectedSection == -1) {
                whispers[0].StartRecording();
            } else
            {
                whispers[selectedSection].StartRecording();
            }
        }
        circleCanvas.gameObject.SetActive(false);
        whispers[0].GetRecButton().interactable = true;
        whispers[0].GetChatGPT().GetButton().interactable = true;
        whispers[1].GetRecButton().interactable = true;
        whispers[1].GetChatGPT().GetButton().interactable = true;
        whispers[2].GetRecButton().interactable = true;
        whispers[2].GetChatGPT().GetButton().interactable = true;
        ccc.SetCircleAble(true);
        ccc.SetKeyEnabled(true);
        boolForToggle = false;
    }
}
```

This section of the class obtains the info of who the message is being sent to and puts the process of starting the recording of the player's voice for that companion is motion. It is worth noting that if the player chooses to speak to all companions at the same time, only one recording is made, only one call is made to the Whisper API and the returned message is then relayed to all companion models. Before moving on to the next subsection of the companion models, here is the circular recording progress bar displaying how long the player has been recording their voice and how much time there is left, filling up clockwise, on the bottom left of the screen and how the response from a companion is displayed if the player does not have the chat box toggled on as they are talking to the companions:

Figure 6 – The circular voice recording progress bar displaying how long the player has been recording.

Figure 7 – The response from the companion displayed above the head of the companion the response is from, the box follows the companion around as they move around.

I would also like to mention that in almost all companion related scripts there are mechanisms to prevent overlapping functionalities to be in use at the same time, hence the regular clauses to check whether certain conditions blocking the class from proceeding are in effect.

There is another class, but it is not worth it to get into the finer details of that class as it simply makes it so that the messages sent by the player and the responses from the companion models are revealed letter by letter as if being written by a typewriter, the class is called TypewriterScript.cs.

III.I.II Companion Prompt Engineering

I would like to start this section by going over our main system prompt that sets the rules and directives for the conversation for the companion models. It is a long but effective. The prompt itself is actually a patchwork of different strings in the class, but I have converted it from its raw state in the class to readable paragraphs, I will be explaining the function calls and their classes as I move through the prompt. It starts here:

This is the setup message. I will give you some information in this message, and in the messages after this one, you will respond knowing the information I give you here. You only understand English and only answer in English.

Here are facts about the world that never change:

`facts.GetFacts()`

Here is your personality; these also never change:

`personality.GetPersonality()`

Your name is `personality.GetName()`, and you are a `personality.GetCharClass()`. On a scale from 1 to 10, with 1 being you dislike me and 10 being you really like me, our relationship is `personality.GetRelation()`.

Here are facts about our current environment; these may change:

`envInfo.GetEnvInfo()`

Here is the history of our travels:

`history.GetHistory()`

Our current task is:

`curTask.GetTask()`

You are in a group of four people; you are not alone. If the player addresses you as multiple people, disregard it, as it means the message is being sent to others as well. If the player asks you how many enemies there are, tell them that the small enemies are no bother and that the real deal is the boss. However, remember that as long as you are not in the Waiting Hall, even if you are not in the boss room, there are enemies around. So if the player tells you to attack, attack.

Besides responding to the query by the player, I want you to create a JSON object from the attributes I give you. Never mention the JSON object. I will give you what the attributes mean, and

you will fill in the JSON attributes with the information from the "info text" and the "command text." Always start and end the JSON object with `. Always give the JSON object at the end of your message; never start with the JSON object. If the player is not giving you a command but simply talking, then only fill the niceness attribute. If the player gives contradictory commands, then ask for elaboration.

If the player gives you multiple commands within the command text, create multiple JSON objects and put them in the same array.

The "info text" lets you know what you have at your disposal, and the "command text" lets you know what the player wants from you. If the player asks for something essential to you, such as your weapon or your clothes, decline, but a health potion is not essential. If the player is asking you to do something, don't narrate the action in your message or respond with the action within asterisks to point out that you are doing the action. For example, if the player is asking you for a health potion, do not say "*Gives a health potion*" or "*Hands over a health potion*". If the player is being ambiguous in their request, you can ask to confirm. You cannot split the items; for example, a health potion cannot be split between you and anyone else. If the player asks you how you're doing, tell them you're fine, taking your relationship into consideration. Never mention the JSON object, only create it and fill it.

The JSON attributes are as follows:

1. command, a string
2. niceness, an integer
3. commandDoability, a boolean
4. whatToGive, a string

Fill the "command" attribute with what command the player is giving you in the command text. It can be the following: goToPlayer, jump, stay, follow, give, use, attack.

Explanations of the commands:

- goToPlayer means the player wants you to go to them; if you are already right next to the player, you cannot get closer.
- jump means the player wants you to jump.
- stay means the player wants you to stay where you are; if the player tells you to go away, it also means the player wants you to stay.

- follow means the player wants you to follow them.
- give means the player is asking you for an item.
- use means the player wants you to use an item on yourself; if the player asks you to use a health potion, tell them you don't need to because you are doing fine, and decline. If the player asks you to use any other item, decline.
- attack means the player wants you to attack an enemy; you can't attack if there are no enemies in the scene.

Fill the "niceness" attribute with how nice the text was towards you. If the message was very rude, make niceness "1," and if it was very kind, make it "10". For example, the message "Can you give me a health potion?" would be a "6," and "Can you give me a health potion, please?" would be an "8".

Fill the "commandDoability" attribute with whether you can do the command given the information from the info text. For example, if you have no health potions, but the player asks you to give them a health potion, commandDoability would be "false". If the player asks you for a potion, you have a potion, and you are willing to give it to them, then commandDoability is "true".

Fill the "whatToGive" attribute if the player asks you to give them an item. Fill the attribute with the item the player asks from you; it can be "healthPotion".

Here is the info text:

```
npcStatus.GetNPCStatus()
```

Occasionally, you will receive messages that start with 'NOT FROM THE PLAYER'. This means that the message is only meant as an update on the current status; it is not an actual message from the player. Do not refer to these messages when you're talking to the player, and you do not need to respond to these messages.

The command texts will follow.

Here the prompt ends. The classes EnvInfo.cs, Facts.cs, History.cs and Task.cs only contain strings, that are the initial information, the value in these classes are never changed throughout the game and these classes are only used in the initial system prompt. The classes NPCInfo.cs and NPCStatus.cs are subject to change, and I will be addressing them in more detail when explaining the system update type of message. This final prompt was developed initially through research on prompt engineering for GPT models, but slowly it evolved into its current state as more needs

arose. Here I should mention that there are two types of messages sent by the player to the model, this difference is only apparent to the model. When the player sends a message, the model decides if the message includes a directive (The player can send multiple directives and they will be interpreted separately, this functionality has been implemented) or simply a chat message. If the message is simply a chat message the model creates a JSON object along with its response and only generates the “niceness” attribute. This attribute is an integer between and including 1 and 10 and represents how kind the message sent by the player was. This niceness value is incorporated into a list, every time the player changes their scene the values in the niceness list are averaged, if the average is above the current relationship value between the player and the companion, their relationship value is increased by 1, if not it is decreased by 1. If the message sent by the player is a directive or multiple directives, there are other attributes to be filled by the model, they are -as mentioned in the initial system prompt-

1. command, a string
2. niceness, an integer
3. commandDoability, a Boolean
4. whatToGive, a string

The explanation of the command attribute is given in the prompt and also explained below:

Fill the "command" attribute with what command the player is giving you in the command text. It can be the following: goToPlayer, jump, stay, follow, give, use, attack.

Explanations of the commands:

- goToPlayer means the player wants you to go to them; if you are already right next to the player, you cannot get closer.
- jump means the player wants you to jump.
- stay means the player wants you to stay where you are; if the player tells you to go away, it also means the player wants you to stay.
- follow means the player wants you to follow them.
- give means the player is asking you for an item.
- use means the player wants you to use an item on yourself; if the player asks you to use a health potion, tell them you don't need to because you are doing fine, and decline. If the player asks you to use any other item, decline.

- attack means the player wants you to attack an enemy; you can't attack if there are no enemies in the scene.

The model interprets the player's message and determines if it includes a directive, and if so, which category the directive falls under. If the player gives contradicting or incomplete commands the model is encouraged to confirm. I've already explained the niceness attribute in the previous paragraph. The commandDoability attribute is whether the command given by the player is possible, whether the command is possible is also determined by the model given the information we provide it. Some examples for the behaviour of this attribute is:

- The player asks the model to "attack" but they are currently in the waiting hall, the commandDoability attribute would be "false".
- The player asks the companion for a health potion but the companion has no health potions left, the commandDoability attribute would be "false".
- The player asks the companion to jump and the companion can jump, the commandDoability attribute would be "true".

Here are sections from the DirectiveHandler.cs class that processes the responses from the model:

Section 1:

```
public class Command
{
    public string command;
    public int niceness;
    public bool commandDoability;
    public string whatToGive;
}

public class DirectiveHandler : MonoBehaviour
{
    public void readJSON(string jsonObj)
    {
        if (jsonObj != null) {
            List<Command> commands = new List<Command>();
            string[] targets = new[] { "`", "json", "[", "]" };
            string a = CleanString(jsonObj, targets);

            List<string> tokens = Tokenize(a);

            foreach (string t in tokens)
            {
                Command command = JsonUtility.FromJson<Command>(t);
                commands.Add(command);
            }

            callDirectives(commands);
        }
    }

    private static string CleanString(string input, string[] targets)
    {
        StringBuilder sb = new StringBuilder(input);
        foreach (var target in targets)
        {
            sb.Replace(target, string.Empty);
        }
        return sb.ToString();
    }
```

Section 2:

```
private void callDirectives(List<Command> commands)
    {
        if (companionNumber == 0)
        {
      companionAi = GameObject.Find("MartialHero").GetComponent<CompanionAI>();
        }
        else if (companionNumber == 1)
        {
            companionAi = GameObject.Find("Mage").GetComponent<CompanionAI>();
        }
        else if (companionNumber == 2)
        {
            companionAi = GameObject.Find("Archer").GetComponent<CompanionAI>();
        }
        foreach (Command c in commands)
        {
            npcInfo.SetRelation(c.niceness);
            if (c.commandDoability) {
                switch (c.command)
                {
                    case "goToPlayer":
                        companionAi.Follow();
                        break;
                    case "jump":
                        companionAi.Jump();
                        break;
                    case "stay":
                        companionAi.Unfollow();
                        break;
                    case "follow":
                        companionAi.Follow();
                        break;
                    case "give":
                        npcStatus.UsedAHealthPotion();
                        break;
                    case "attack":
                        companionAi.AttackNow();
                        break;
                    case null:
                        //NO DIRECTIVE ONLY NICENESS
                        break;
                    default:
                        Debug.LogError("Unknown state");
                        break;
                }
            }
        }
    }
}
```

In section 1, the JSON section of the message is cleaned and the JSON object sent by the model is integrated into a structure (or an array of structures if the player has sent multiple directives in a single message), in section 2 the structure is analysed and the relevant command is called from the CompanionAI.cs class, this class is more relevant to other sections of our project, therefore I will not be going into the details of that class here.

III.I.III Companion System Updates

The environment related updates are taken from the DungeonMasterInfoCollector.cs class, since the required information is already gathered in that class to feed to the prompt of the dungeon master model, I will not be going into the details of that class in this section but the relevant parts are as follows:

```
    gpt1.SetEnvironmentInfo("You are currently in the " + dungeonName + " the
description of the enemies around is " + getMonsterInfo(), " your task is to kill
the " + sceneName.name + " boss, kill the smaller enemies as you get to the boss
room, you will be informed when you are in the same room with the boss");

    gpt1.SetEnvironmentInfo("You are now in the " + sceneName.name + " boss
room.", "your task is to kill the " + sceneName.name + " boss");

    gpt1.SetEnvironmentInfo("You killed the boss of this dungeon.", "Since you
killed the boss, your task is to rest for now.");
```

This is done for all companion models, these are examples of the models being fed updated information when a significant event happens, here is the update prompt:

NOT FROM THE PLAYER
New facts about our current environment:
`endString`
And our current task is:
`curBoss`
Here is your current status, the info text:
`npcStatus.GetNPCStatus()`
Here is your updated relationship with the player:
`personality.GetRelation()`

The update prompt ends here. The “NOT FROM THE PLAYER” part of the prompt first mentioned in the initial system prompt is to inform the model that this message does not require a response. The methods called in the DungeonMasterInfoCollector.cs class edit the `endString` and `curBoss` strings and call the system update completion response method, meaning when a change is initialized at the DungeonMasterInfoCollector.cs class, that information is sent to the model immediately. If DungeonMasterInfoCollector.cs class does not call an update, the update is still periodically sent as information inside the NPCStatus.cs and NPCInfo.cs methods might be updated, an example is when the distance between the player and the companion has changed, the methods to send the system update prompt has not been called, so the method is automatically called after a certain amount of time. The system update prompt is also immediately sent if the player asks for a health potion, this is done because if the player asks for more health potions than the companion has before a periodic system update prompt was sent then the player would be given more health potions than the companion has. The NPCInfo.cs class is where the relationship between the player and the companion is handled (each companion has their own NPCInfo.cs and

NPCStatus.cs classes) and the personality of the companion is stored. The NPCStatus.cs class is where the inventory of the companion is managed and the distance between the player and the companion is updated.

One important process I've left out is how the completion responses between the player and the companion model is handled, but it is a simple process, I will be going over it in more detail in the entire process demonstration.

III.I.IV Companion Entire Process Demonstration

I will be going through the entire process that goes on when the following actions are taken by the player:

- The player uses the chat chooser circle to select the samurai.
- The player holds down the V key to record their voice and lets go of the V key to finish their recording.
- The speech is translated into text.
- The text is edited, the message section is displayed on top of the companion and the directive section (JSON object) is sent to the directive handler.
- The directive handler executes the directive.

When the player holds down the space button, the chat chooser circle is displayed, when the player moves their mouse in a certain direction that section is slightly enlarged to emphasize the currently selected section. To select the samurai, the player needs to move their mouse slightly to the top-right of their screen. When the player lets go of the space button the WhisperCaller.cs class is called to enforce the selection of the player, the WhisperCaller.cs method then calls the appropriate Whisper.cs class when the V key is pressed. The SaveWav.cs class handles the recording of the speech, the Whisper.cs class handles the Whisper API call. After the API call is complete and the model returns the text, the text is relayed to the related companion model, the player's message and the response from the model are both added to the chat box and displayed on top of the companion if the chat box is toggled off. Before being sent to be displayed, the response from the companion model is divided into the message part and the JSON part, the JSON part is directed to the DirectiveHandler.cs class and the message part remains in the ChatGPT.cs class, where all API calls to the companion model are made (Each companion has their own Whisper.cs, NPCStatus.cs, NPCInfo.cs, DirectiveHandler.cs and ChatGPT.cs class). The DirectiveHandler.cs class trims the JSON object and converts it into a structure, the structure is analysed and the relevant method to execute the player's directive is called using a switch statement.

III.II. Dungeon Master Functionality

In the beginning of the dungeon master model research, we thought we could use a local model. But we decided to use an API for the OpenAI model GPT-4o. There are several reasons we decided to use an API instead of a local model: We can harness the power of a large model instead of relying on a small local model and a large model can create a more immersive narration for our project. We started by watching a tutorial[4] and tried to learn to create an API. We first created a basic prototype ChatGPT API. After that we integrated the dungeon master model to our already existing Unity project. In the further sections of the report, -the details of the companion model development-, I will not be going through the history of our development process unless it is relevant, but I will rather explain the project as it is in the final version.

III.II.I. Dungeon Master UI

There is only one basic UI element for the dungeon master. Its response can be seen when entering a new room or killing a boss:

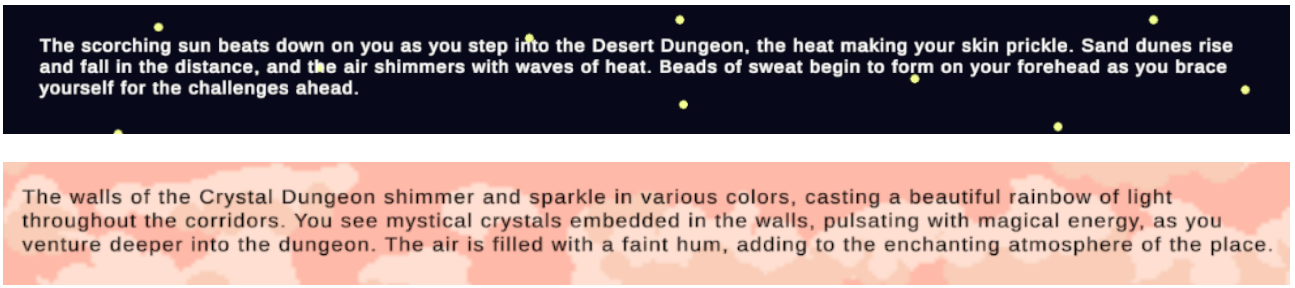

Figure 8 – The response from the dungeon master is displayed in the upper part of the game camera.

Here is the Dialogue.cs class that controls the narration when entering a new dungeon:

```
using UnityEngine;
using TMPro;
using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;

namespace OpenAI
{
    public class Dialogue : MonoBehaviour
    {
                private Dictionary<string, string> sceneNarrations = new
Dictionary<string, string>()
        {
            {"WaitingHallScene", "You are in the Waiting Hall."},
```

```
            {"RockyScene", "You enter the Rocky Dungeon(Rocky, cold)."},
            {"CrystalScene", "You enter the Crystal Dungeon(Colorful)."},
                {"SnowScene", "You enter the Snow Dungeon(Very cold, very
white)."},
             {"DesertScene", "You enter the Desert Dungeon(Very hot, sun in
the sky)."}
        };
        public TextMeshProUGUI textComponent;
        private OpenAIApi openai = new OpenAIApi();
        private List<ChatMessage> messages = new List<ChatMessage>();
         private string prompt = "You are a dungeon master for an AI driven
2D RPG game. When I give you instructions inside quotes (\"\") you will
narrate them. Game will send you interaction logs and you will narrate them.
The outputs must be in two to three sentences, it mustn't be one sentence.
Make short but descriptive sentences. Don't add any additional details that
might destroy the story continuity. DON'T ANSWER FOR THE FIRST TIME. WHEN
YOU SEE QUOTES THEN ANSWER! Don't get out of character. Don't say that you
are an AI. The outpust must be one paragraph";

        private void Start()
        {
                                                string    currentScene    =
UnityEngine.SceneManagement.SceneManager.GetActiveScene().name;
            if (sceneNarrations.ContainsKey(currentScene))
            {
                // If it does, use it
                ReceiveSystemMessage(sceneNarrations[currentScene]);
            }
            else
            {
                // Otherwise, use a default message
                ReceiveSystemMessage("You enter an unknown location.");
            }

        }
        public async void ReceiveSystemMessage(string systemMessage)
        {

            var newMessage = new ChatMessage()
            {
                Role = "system",
                Content = systemMessage
            };

              if (messages.Count == 0) newMessage.Content = prompt + "\n" +
```

```
systemMessage;

            messages.Add(newMessage);

            // Complete the instruction
              var completionResponse = await openai.CreateChatCompletion(new
CreateChatCompletionRequest()
            {
                Model = "gpt-3.5-turbo",
                Messages = messages
            });

                                if (completionResponse.Choices != null &&
completionResponse.Choices.Count > 0)
            {
                var message = completionResponse.Choices[0].Message;
                message.Content = message.Content.Trim();

                messages.Add(message);
                textComponent.text = message.Content;
            }
            else
            {
                Debug.LogWarning("No text was generated from this prompt.");
            }
        }

    }
}
```

Additionally, we used the one shown below to give the dungeon master the current room number, which dungeon the player is in, enemy number/type and the info about the player party:

```
roomName = " Area " + this.roomNumber + ", ";
                monstersInTheRoomInfo = spawnedEnemyNum + " Monsters. Type:
" + getMonsterInfo();
                partyInfo =
player.GetComponent<PartyScript>().getPartyInfo();
                newNarration.ReceiveSystemMessage(dungeonName + roomName +
monstersInTheRoomInfo + partyInfo);
```

III.II.II Dungeon Master Prompt Engineering

We tried many different prompting methods and tried to learn it by reading prompting tutorials[4]. Hallucination is a degenerate response given by a Large Language Model(LLM). We want to decrease the hallucination so that the model can answer correctly. So, we created a detailed prompt that can control the behaviour of the dungeon master model. The system prompt we use is shown below:

```
        private string prompt = "You are a dungeon master for an AI driven
2D RPG game. When I give you instructions inside quotes (\"\") you will
narrate them. Game will send you interaction logs and you will narrate them.
The outputs must be in two to three sentences, it mustn't be one sentence.
Make short but descriptive sentences. Don't add any additional details that
might destroy the story continuity. DON'T ANSWER FOR THE FIRST TIME. WHEN
YOU SEE QUOTES THEN ANSWER! Don't get out of character. Don't say that you
are an AI. The outpust must be one paragraph";
```

With this prompt model starts to act like a dungeon master for our game. We made sure that the response of the model will be at most one paragraph by specifying how many words we want in its response.

III.II.III Dungeon Master Narration Updates

When the game environment changes -Changing the dungeon room or entering a new dungeon-, the dungeon master is informed. Dungeon master generates the new response for the room which the player is currently in. This creates a dynamic system so that the dungeon master can answer in every possible game state. The below code is how we update the dungeon master's knowledge about the game:

```
roomName = " Area " + this.roomNumber + ", ";
                monstersInTheRoomInfo = spawnedEnemyNum + " Monsters. Type:
" + getMonsterInfo();
                partyInfo =
player.GetComponent<PartyScript>().getPartyInfo();
                newNarration.ReceiveSystemMessage(dungeonName + roomName +
monstersInTheRoomInfo + partyInfo);
```

Every time we enter a new area; roomNumber, spawnedEnemyNum and partyInfo is given to the dungeon master model using the ReceiveSystemMessage function. roomNumber is the room number of a dungeon that the player is in. spawnedEnemyNum is the currently spawned enemy

number in the room. partyInfo is the information about the party members.

III.II.IV Dungeon Master Entire Process Demonstration

I will be going through the entire process that goes on when the following actions are taken by the player:

- The player enters a new area of a dungeon or kills a boss.
- The dungeon master is prompted to generate a text.
- The text is sent to Unity UI.
- The text is shown at the top of the player camera.

When the player enters a new dungeon or kills a boss, the dungeon master is prompted to generate a text response. The response is the summary of the game state that the player is in. Dialogue.cs class handles the first narrations when entered to a new dungeon and for the waiting hall. DungeonMasterInfoCollector.cs class collects all the information needed for the dungeon master prompt. Additionally, it controls the spawn of the enemy monsters, so that we can get the number of enemies spawned on the map. callNarrator function calls the needed functions: getMonsterInfo, getPartyInfo. Finally, callNarrator function combines all the gathered knowledge about the game:

```
newNarration.ReceiveSystemMessage(dungeonName + roomName +
monstersInTheRoomInfo + partyInfo);
```

As a result we get the narration for the game state the player is in.

III.III Game Environment Functionality

The environment of the game is built in with the rich and usable interface of the Unity game engine. Since Unity already has a 2D physics engine inside itself, we created this game more easily. The player's collusions and attacking are controlled by the "2D Box Collider". The player has its own box collider. This collider checks if any object that has a collider on it collides with itself or not. When the collider of the player and collider of the other object contacts the collider of the player gives information about the contact. Adding "2D Rigid Body" to an object puts it under control of the physics engine. This means that the object will be affected by gravity and can be controlled from scripts using forces. Detailed information about the game environment are given below.

III.III.I. Player

Figure 9 – A snapshot of the player in the game

The player has a role which is a knight. He can be controlled by using “W”, “A” and “S” on the keyboard and using mouse left click. When the “W” key is pressed the player jumps and makes the “Jump” animation. When the “A” key is pressed, he moves to the left and when the “B” key is pressed, he moves to the right and makes a “Move” animation. By clicking the “left click” of the mouse the player attacks and makes the attack animation. The player’s main goal is to finish all the dungeons ahead of him. The player has a “2D Box Collider” and “2D Rigid Body” which makes the player under control of the physics engine. The player has a health bar that shows his current health. When the player gets damage from the outside world it decreases the health point on the health bar with the taken damage amount. The player can increase his health point by using a health potion that can be gathered from the drop of the dead bosses or from companions beside the player. By pressing “left click” on the mouse the player decreases health points of the enemy if the enemy is in the radius of the attack.

III.III.II. Enemies

There are 10 types of enemies, 6 of them are low difficult enemies and 4 of them are bosses where only at the final map of the dungeon which is called “Boss Room”. All of the enemies have boundaries where they must move along. They start from moving to the left and rotate after reaching the boundaries that are attached to them. The enemy has a detection radius that checks if the player is in that range or not. If the player is not in the range, they continue to move along the boundaries but if the player is in the detection radius that is given to the enemy, the enemy follows and try to attack the player until reaching their boundaries.

Flying Enemy:

Figure 10 – A snapshot of the flying enemy

Flying enemy detects the player in an area within a 5 unit radius (detectionRadius), while moving between the left and right boundaries (leftBound and rightBound) at a certain speed (moveSpeed). The enemy starts with maximum health (maxHealth) of 100 and its health is updated depending on the damage it takes, which is visually indicated by the healthBar. When the enemy takes damage, it turns red for a short time and slows down (slowDownFactor and slowDownDuration). It can attack the player when it approaches within a 2 unit radius (attackRadius), and these attacks are limited to a 1.5 second cooldown (attackCooldown). Attack and flying animations are managed with the animator component. The enemy moves between the determined borders to the right and left, and changes direction when he crosses the borders. When its health drops to zero, it is destroyed (Die). Additionally, when attacking the player, it deals damage taking into account the player's defense points. The physical movements of the enemy are controlled by the Rigidbody2D component and its visual representation is made by the SpriteRenderer component. These features create a flying enemy character that follows, attacks, and moves dynamically around the player.

Green Enemy:

Figure 11 – A snapshot of the green enemy

The green enemy travels at a speed of one unit per second between the marked left and right limits. tracks the player once they go beyond a 5-unit radius after it detects their presence. He has 100 health, and a graphic health bar shows the damage he receives. It slows down and becomes red when damaged. Its attacks happen every 1.5 seconds and can target the player within a 2-unit radius. Animations accompany the movements and attacks of the enemy. When its health reaches zero, it is destroyed. In order to harm the player, this opponent also considers the player's defense points.

Plant Enemy:

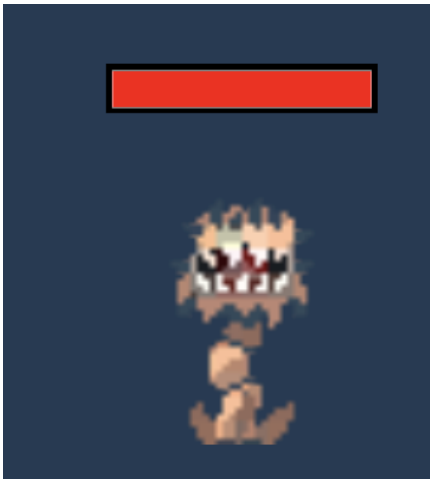

Figure 12 – A snapshot of the plant enemy

Plant enemy detects the player within a 5-unit radius and attacks when it comes within a 2-unit attack range. His attacks occur in 1.11-second intervals and are supported by animations. The health of this enemy, whose maximum health is 100 units, is displayed with a visual health bar in line with the damage received. It turns red for a short time when damaged and is destroyed when it dies. When attacking the player, it deals damage taking into account the player's defense points. This enemy uses animations to manage its attack and cooldown states, and will revert to its old hue after a certain amount of time.

Ra Enemy:

Figure 13 – A snapshot of the ra enemy

The aggressive Ra Enemy has a five-unit detection radius. It patrols inside predetermined areas and moves closer to the player when it detects them. can launch an attack within a two-unit attack radius once every 1.5 seconds. It travels at a speed of one unit per second, with restricted left and right movement. With a maximum health of 100, this adversary displays its health condition based on the damage it sustains and has a brief slow down when injured. When its health reaches zero, it is destroyed. Movement direction determines where the health bar appears, and animations accompany both attacks and movements.

RedHead Enemy:

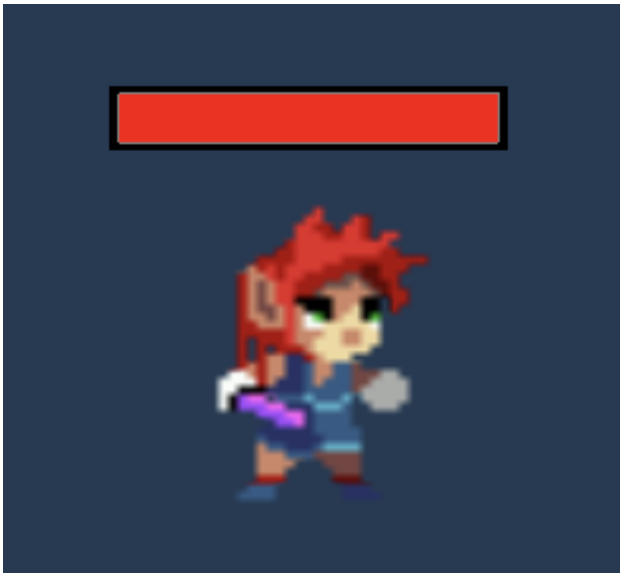

Figure 14 – A snapshot of the redhead enemy

RedHead Enemy is a dangerous enemy that can have up to 100 health. It moves in specific circles and approaches the player when it senses movement. Its attack radius is two units, and its detection radius is five units. It travels at a speed of one unit per second and halts to attack the player when it approaches. The interval between attacks is one and a half seconds. His speed is momentarily decreased and the damage he sustains is displayed on his health bar. When its health reaches zero, it is destroyed. As you move, the animations and health bar position change dynamically.

Snakeman Enemy:

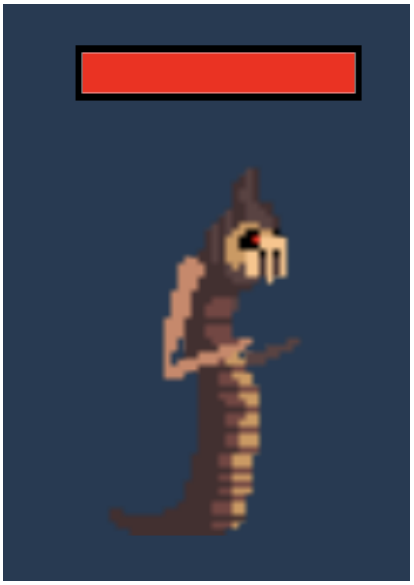

Figure 15 – A snapshot of the snakeman enemy

An enemy known as SnakeMan moves in the direction of the player after detecting them. It can only have 100 health points at most. The opponent advances toward the player when they move inside its detection radius, stopping to launch an attack when it gets too far away. It deals damage to the player when it enters the attack radius and initiates a specific attack cooldown after that. The enemy's speed is momentarily decreased and their health bar is updated as they receive damage. The enemy is eliminated when its health point is zero. During movement, animations and the position of the health meter are dynamically altered.

III.III.III. Bosses

Crystal Boss:

Figure 16 – A snapshot of the Crystal Boss

Boss Crystal is a powerful enemy that detects the player and moves towards him. It has 1000 maximum health points. When the player enters the enemy's detection radius, the enemy runs towards the player and stops when it reaches a certain distance. After a certain attack cooldown, it moves to attack the player. When the enemy takes damage, their speed is temporarily reduced and the amount of health on their health bar is updated. When the health point is zero, the enemy disappears and drops rewards such as health potion, green armor and sword. When the boss dies, a portal appears and takes the player to the next section.

Snow Boss:

Figure 17 – A snapshot of the Snow Boss

Boss Snow, the lord of the Snow Region, is a fast and powerful enemy against the player. This boss, which has a maximum health point of 1000, flies towards the player when it detects it and stops when it reaches a certain distance. Attacks the player after a certain attack cooldown. When he takes damage, he is temporarily slowed and the amount of health on his health bar is updated. When the health point is zero, Snow's sudden death leaves the player with a health potion and special items. Additionally, upon his death, a portal appears and takes the player to the next chapter.

Rocky Boss:

Figure 18 – A snapshot of the Rocky Boss

Boss Rocky, the lord of the Rock Region, is a powerful enemy to the player. This boss, which has a maximum health point of 1000, approaches the player by walking when it detects the player and stops when it reaches a certain distance. Attacks the player after a certain attack cooldown. When he takes damage, he is temporarily slowed and the amount of health on his health bar is updated. When the health point is zero, Rocky's sudden death leaves the player with a health potion and special items. Additionally, upon his death, a portal appears and takes the player to the next chapter.

Deset Boss:

Figure 19 – A snapshot of the Desert Boss

Boss Desert, the lord of the Desert Region, is a powerful enemy against the player. This boss, which has a maximum health point of 1000, runs towards the player when it detects it and stops when it reaches a certain distance. Attacks the player after a certain attack cooldown. When he takes damage, he is temporarily slowed and the amount of health on his health bar is updated. When the health point is zero, the Boss Desert leaves the player with a health potion and special items upon his sudden death. Additionally, upon his death, a portal appears and takes the player to the next chapter.

III.III.IV. Dungeons

There are a total of 4 dungeons in our game. They are “Crystal Dungeon”, “Snow Dungeon”, “Rocky Dungeon” and “Desert Dungeon”. Each of the dungeons are unique with the design of themselves. The dungeons create enemies themselves with the help of the spawners. There are spawners in the dungeons that spawn enemies by looking at the finish time of the player. If the player finishes one room 0-20 seconds, the spawners spawn enemies between 6-10. If the player finishes between 20-40, the medium mode is selected and 3-5 enemies spawns. Else, 1-2 enemies can spawn which is the easy mode.

Crystal Dungeon

Figure 20 – A snapshot of the Crystal Dungeon

Figure 21 – A snapshot of the Goblin Boss.

Snow Dungeon

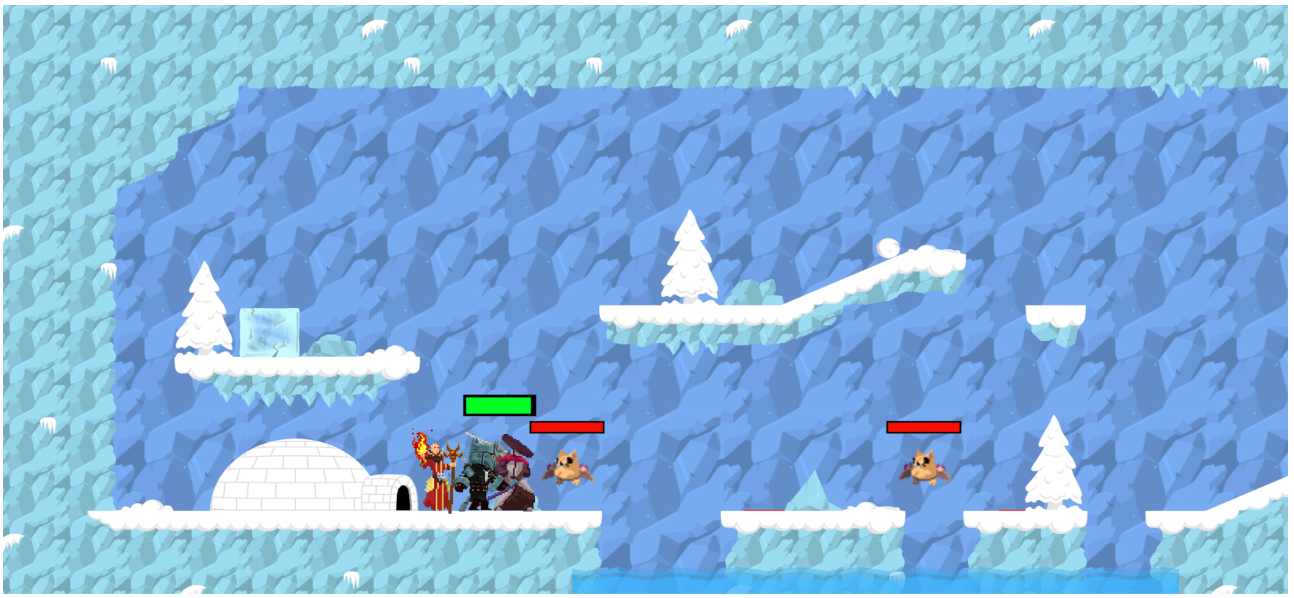

Figure 22 – A snapshot of the Snow Dungeon.

Figure 23 – A snapshot of the One Eye Boss.

Rocky Dungeon

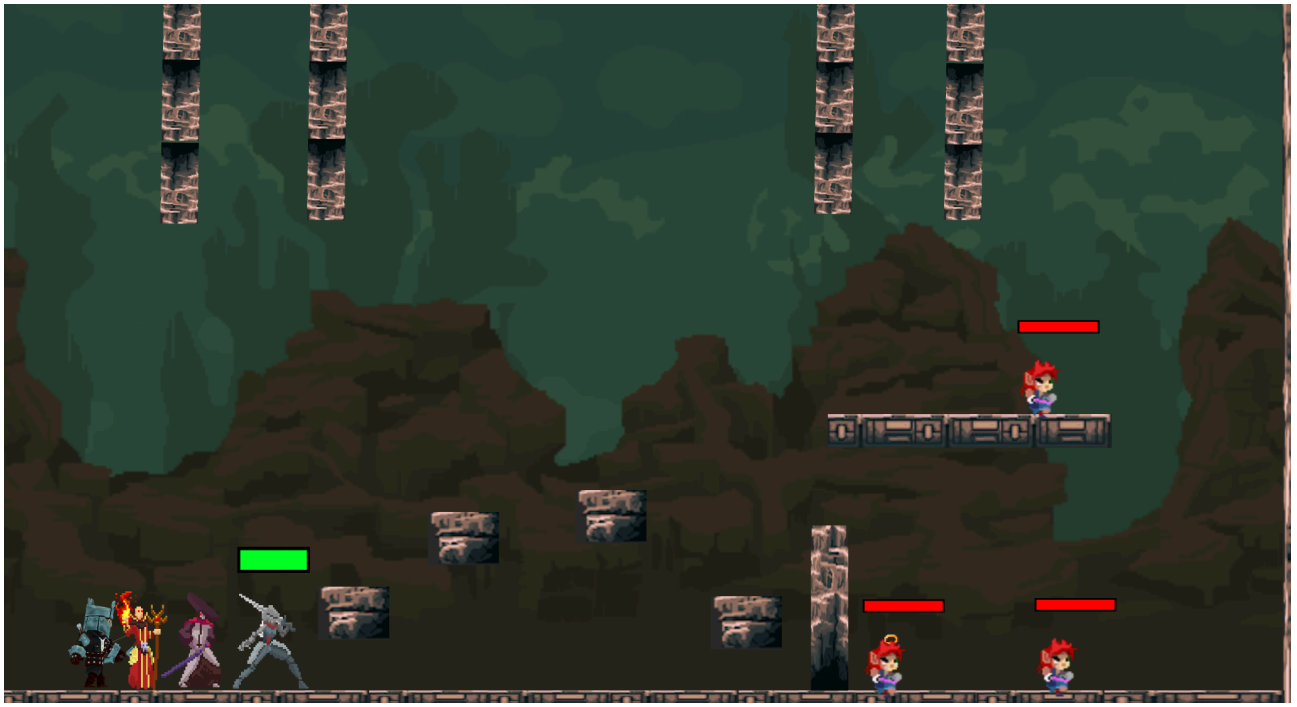

Figure 24 – A snapshot of the Rocky Dungeon.

Figure 25 – A snapshot of the Skeleton Boss.

Desert Dungeon

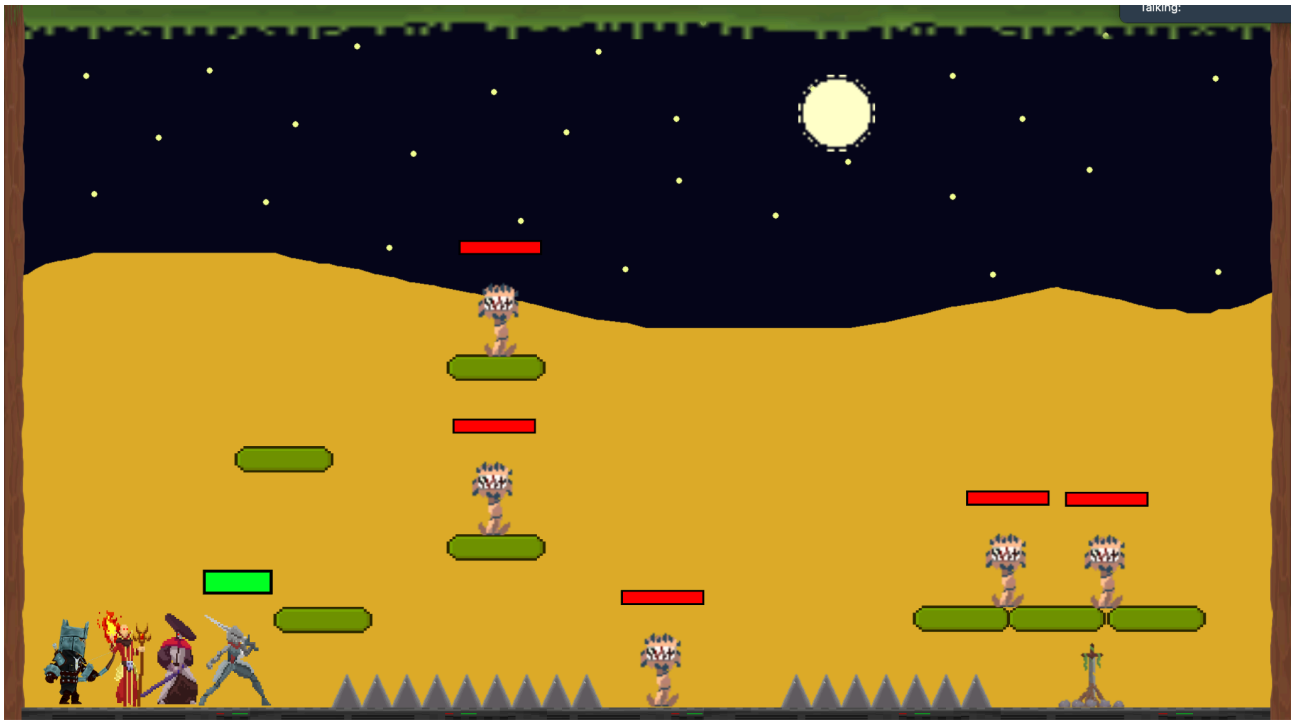

Figure 26 – A snapshot of the Desert Dungeon.

Figure 27 – A snapshot of the Mushroom Boss.

IV. Analysis

Our team played the game over and over again more than 500 times. We made our analysis on the game which are described below:

- Game changes the scenes correctly.
- The enemies move perfectly in the horizontal direction.
- The enemies' attack animation and attack damage work clearly.
- The enemies' health bar works correctly.
- The bosses move perfectly in the horizontal direction.
- The bosses' attack animation and attack damage and death animation work clearly.
- The bosses' health bar works correctly.
- The bosses drop 3 items which are a weapon, an armour and a health potion without an issue.
- The inventory system works by pressing "i" on the keyboard.
- The items that are gathered are usable in the player's inventory.
- The player moves perfectly in the horizontal direction.
- The player moves perfectly in the vertical direction.
- The player's attack animation and attack damage work clearly.
- The player's health bar works correctly.
- The player has 3 lives and dies after 3 deaths.
- The teleportation of the player works successfully.
- The player's whisper to the companions works with the voice detection.
- The companions follow the player successfully.
- The companions attack the enemies correctly.
- The companions talk correctly and remember the game events correctly.
- The companions follow the orders of the player correctly. For example, jumping, attacking, giving potions, following the player and unfollowing the player.
- The dungeon master model narrates the currently seen scenes correctly.
- The player continues to stay in the air when the player collides with an object even though the player should have fallen.

V. Conclusion

In conclusion, our project successfully demonstrated the enhanced immersion achievable through the integration of AI models into video games. We enabled dynamic and context-aware interactions beyond the conventional static dialogue choices by using the gpt-4o model for our companion NPCs. This method greatly improved the gameplay experience by enabling players to command companions and have real conversations. A stronger bond between the player and the game environment was promoted by the companions' continual contextual relevance and responsiveness, which was ensured by the regular updates on game happenings. Furthermore, the integration of voice chat, which makes use of OpenAI's Whisper architecture, strengthened the immersion by allowing users to converse with companions in a natural way.

By Dungeon Master model, players of our game have been able to convey what happens within the game world in a contextual and consistent manner. Thanks to this other strategy, we have created a robust and immersive gaming environment that has not been thoroughly explored in existing video games.

Overall, a successful integration of the AI models we use when considering the project not only demonstrates the potential for greater immersion in video games, but also opens up new possibilities for interactive storytelling and player participation. As AI technology evolves over time, we expect even greater advancements in creating immersive and responsive gaming experiences.

VI. References

[1] Dwemer Dynamics, "Herika - The ChatGPT Companion," Nexus Mods, April 1, 2024. [Online]. Available: https://www.nexusmods.com/skyrimspecialedition/mods/89931.

[2] **Create a Large Language Model from Scratch with Python – Tutorial**. YouTube, uploaded by Elliot Arledge, 8 Jan 2023, https://www.youtube.com/watch?v=UU1WVnMk4E8.

[3] A. Gokaslan, V. Cohen, E. Pavlick, and S. Tellex, "OpenWebText Corpus," Hugging Face, 2019. [Online]. Available: https://huggingface.co/datasets/Skylion007/openwebtext.

[4] Sarge. (2023, January 23). Prompt Engineering in ChatGPT for making NPCs! [Video]. YouTube. https://www.youtube.com/watch?v=usrxIUGK9Gc

[5] "Prompt Engineering Best Practices: Tips, Tricks, and Tools | DigitalOcean," *www.digitalocean.com*.https://www.digitalocean.com/resources/article/prompt-engineering-best-practices (Online).

VII. Appendix