Exploring the Effectiveness of Dialogue Interaction Methods with Non-Player Characters in Video Games

Cameron Main (200425522)

BSc Computer Science, May 2023
Supervisor: Dr. Giacomo Bergami
Word Count: 13,624

Abstract

This dissertation studies the effectiveness of different dialogue interaction methods with Non-Player Characters within video games. This is tackled through the development of a small video game within the Unity game engine that incorporates conventional methods of NPC dialogue interaction, as well as newer experimental means of interaction. The experimental methods rely on Natural Language Processing techniques and Large Language Models. The effectiveness of each method is evaluated in both the context of implementation feasibility and how players of the game perceive each method.

Declaration

“I declare that this dissertation represents my own work except where otherwise stated.”

Contents

Table of Figures

Introduction

Context

In recent years, the gaming industry has experienced unparalleled growth, with an escalating demand for more immersive and realistic gaming experiences, projected to reach a staggering $321 billion by 2026 [1]. Non-Player Characters (NPCs) are a crucial component of video games and play a significant role in the overall gameplay experience. NPCs serve as key elements for advancing the narrative, providing challenges for players, and offering important information to guide their actions. Their interactions with players are critical to the success of a game, shaping their perceptions of the game world and overall enjoyment.

Traditional NPC interaction methods that have been popularised and used over the past few decades are often scripted, limited, and repetitive, resulting in a lack of engagement and immersion among players. A substantial 52% of gamers say that they dislike repetitive NPC dialogues [2]. Furthermore, traditional NPC interactions often lack depth and fail to respond meaningfully to a player's actions, resulting in a limited sense of player agency and a feeling of detachment from the game world. Ultimately, this led to decreased engagement and enjoyment. Although there have been some attempts to improve NPC interactions, such as the use of branching dialogue trees, these methods are still limited by predefined scripts and lack the ability to adapt to the player's actions and dialogue in real time.

Purpose

The purpose of this project was to explore how Machine Learning (ML) and Natural Language Processing (NLP) techniques can produce AI-generated responses at runtime from player inputs that facilitate more realistic and immersive NPC interactions in a video game.

The benefits of creating such a system are twofold. First, it benefits players by addressing the previously outlined problem of the current state of NPCs, namely, bland, repetitive, and nonreactive. In addition to benefiting developers, utilising AI-generated dialogue responses would remove the monetary and time costs associated with hiring script writers who must write out each line of dialogue manually and then assign a developer to the relevant character and situation.

Large games studios such as Bethesda Softworks, who pride themselves in having substantial dialogue systems with rich and varied dialogue in their role-playing games, could potentially benefit from utilising AI to generate dialogue dynamically to save the costly expenses of the traditional method of script writing.

As games become more advanced and gamers demand more from developers, both graphically and story-wise, the development time for games is also increasing. Simply put, the bar is raised with each game iteration, thus requiring more time to produce something that will top the

previous and exceed consumer expectations.

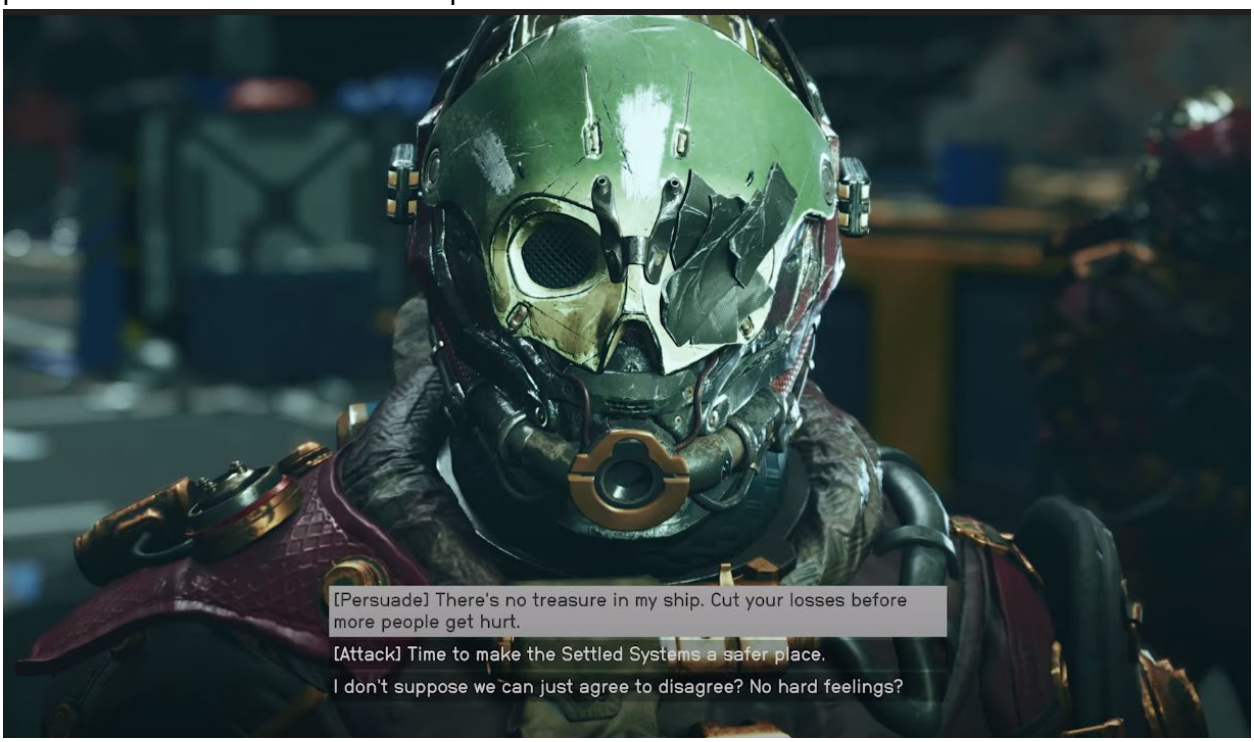

Figure 1 In-game image of Starfield's dialogue interaction UI [3] Credit: IGN

Bethesda is a prime example of this; their 2011 release *The Elder Scrolls V: Skyrim* contained around 60,000 lines of dialogue [3]. Their next release was in 2015 with *Fallout 4*, in which the dialogue increased to approximately 111,000 lines. Now, their upcoming release in 2023, *Starfield*, boasts an astounding 252,953 lines of dialogue and counting as development continues. Utilising a method to bypass the requirement of writing and incorporating these lines could greatly increase game development productivity, reduce costs, and please gamers with fulfilling and interactive experiences.

Project Aim

The primary aim of this project is to create a prototype game that employs NPCs capable of interacting with players in a procedural, reactive, and coherent manner, thereby helping players complete the level or quest objectives. In addition, the game features NPCs using the traditional preconfigured dialogue found in most video games.

To evaluate the effectiveness of the novel ML/NLP-based NPC interactions compared to the standard pre-configured dialogue, a group of participants, including gamers and non-gamers, will play the game and provide feedback on their overall gameplay experience. Feedback mainly focuses on NPC interactions. This feedback will be analysed to determine the viability of implementing this new approach in future video games. It has the potential to provide more

immersive and engaging experiences for players and streamline development time compared with conventional methods.

By the end of this project, I hope to demonstrate whether this new approach to NPC interaction is effective and efficient, potentially offering new directions for future game development.

Project Objectives

Objective 1

Research and incorporate conventional NPC dialogue interaction methods. The most common methods will be identified, of which the most appropriate will be chosen to be implemented into the Unity project.

Objective 2

Conduct User Research to Gather Feedback on NPC Interactions in Video Games. Liaising with gamers of varying experience levels to gather qualitative data on their opinions on NPC interactions in video games will allow me to gain insight into their frustrations, preferences, and expectations, which will inform the development of a more user-friendly and engaging system. Successful data collection provides user requirements that must be met when building the system.

Objective 3

Research and understanding of how open and closed domain dialogue generation systems work. The available literature on open and closed-domain dialogue generation systems will be examined. These systems are often utilised in natural language processing (NLP) and can be designed to generate conversational responses for NPCs in video games. Research on Large Language Models (LLMs) is also required. This analysis will provide me with the fundamental understanding necessary to develop LLM/NLP-based NPC interactions in Unity.

Objective 4

Implementation and testing of LLM and NLP-based NPC interactions in the Unity Project. The knowledge gained from Objective 3 of LLM/NLP-based NPC interactions can be integrated into a Unity project. Testing will then be conducted to ensure that the interactions work as intended and identify any potential issues.

Objective 5

Conduct user playtesting and gain feedback on the implementation. Qualitative data will be collected from participants who play the game to gain a deeper understanding of their gameplay experiences, particularly their interactions with the NPCs.

Objective 6

Evaluation of the viability of implementing each NPC interaction method in video games. The potential of implementing this approach in future video games will be evaluated. Factors such as the level of engagement provided to the players, resources required for implementation, and the technical feasibility of the approach will be considered. This evaluation will help determine whether LLM/NLP-based NPC interactions can be used to provide more engaging and immersive experiences for players at any cost.

Research

This chapter addresses how the research was conducted for this dissertation. It will include the background material utilised to gain an understanding of the topic, as well as how research was conducted through a series of interviews with gamers.

Background Reading

To undertake this project, it was necessary to move beyond the conventional rule-based approach of creating game AI, such as state machines, which generate predetermined responses to stimuli. Initial background research indicated that foundational models enable greater flexibility and adaptability in NPC behaviour.

Foundation Models

Foundation models present a new approach to artificial intelligence with the goal of building general and adaptable models that can be used across various tasks and fields with minimal modification. IBM defines foundation models as models trained on a large and diverse set of unlabelled data using self-supervised learning techniques that leverage the structure and patterns in the data to generate labels or objectives for the model to learn from [4].

Looking a little deeper, R. Bommasani et al. (2021) states a foundation model as being any model that is trained on broad data at scale and can be adapted to a wide range of downstream tasks [5]. The authors use the term “foundation model” to underscore the critically central yet incomplete character of these models. Foundation models are based on standard deep learning and transfer learning techniques; however, their scale results in new emergent capabilities. Their effectiveness across many tasks incentivises homogenisation, but this also means that the defects of the foundation model are inherited by all the adapted models downstream.

Self-supervised learning enables the model to learn transferable representations of data without requiring human annotation or domain-specific knowledge. These representations can be fine-tuned for specific tasks with minimal additional data or computation, reducing the cost and time required to develop and deploy new AI systems as well as the environmental impact of training large-scale models.

Several impressive examples of foundation models, such as GPT-3, BERT, DALL-E 2, and Codex, have shown promising results in NLP and computer vision.

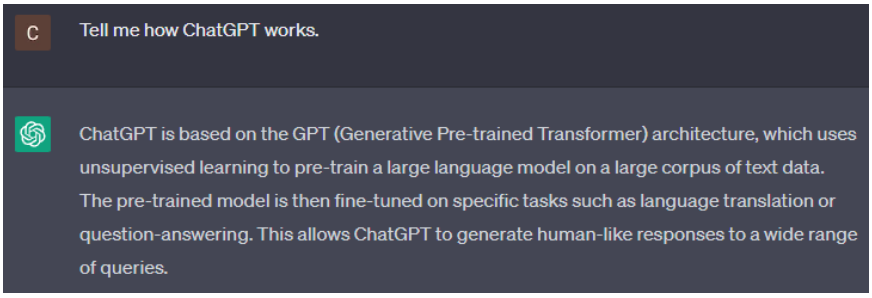

Figure 2 Example of ChatGPT explaining how itself works.

However, there are also significant costs, challenges, and risks associated with the foundation models. These include ensuring the quality, diversity, and fairness of the data used to train the models; understanding the limitations and biases of the models and their outputs; ensuring their security and robustness against adversarial attacks or misuse; and developing ethical and legal frameworks for regulating their use and impact on society.

Moreover, the biggest cost associated with foundation models is the significant computational resources required, and as such, large comprehensive models are reserved for billion-dollar industry leaders in cloud computing, such as Microsoft and Google. Self-proclaimed “world leaders in artificial intelligence computing”, Nvidia, claim that training a model has an estimated cost of $1bn [6].

Understanding this plays a pivotal role in determining the scope of a project. Gaining a background comprehension of what goes into creating a foundation model and how they can be utilised for an array of general purposes underpins this project as a whole, and sets expectations of what can and cannot be achieved with the given limitations of budget, time, and resources.

Natural Language Processing

Natural Language Processing (NLP) has been mentioned numerous times, and explaining it and how it will play into the project is a necessary step.

NLP is a subfield of computer science that aims to impart computational systems with the ability to comprehend and formulate texts and speech in a manner that mimics human language. The scope of NLP is broad, encompassing applications such as machine translation, data extraction, text emotion analysis, text summarisation, query answering, and dialogue systems [7]. Achieving proficiency in NLP depends on various methods and techniques, including those from the domains of artificial intelligence, linguistics, statistics, and machine learning.

It is NLP, which will be the driving force of this project, as building a game with a reactive dialogue system will depend heavily on these practices to produce a game AI that can interpret the player's text input and output an appropriate response.

Open & Closed Domain Systems

A significant challenge in NLP is building systems that can understand and model the context of a conversation. Conversational systems aim to achieve this goal by extracting the relevant information from conversational data and representing it in a structured manner. Such systems can be classified into two types: open domain and closed domain.

Open domain systems are designed to handle conversations that span a wide range of topics and scenarios, such as general chatbots, social media analysis, and question-answering systems. Additionally, open domain systems do not assume any prior knowledge of the domain or the participants of the conversation, and they rely on large-scale data sources and general-purpose NLP models to infer the meaning and intent of the utterances. Thus, while making them more flexible and adaptable, they also face challenges in terms of ambiguity, noise, and complexity of conversational data [8].

Conversely, closed-domain systems are designed to handle conversations that are limited to a specific topic or scenario, such as customer service, booking, or medical diagnosis. These systems assume some prior knowledge about the domain and the participants of the conversation, and rely on domain-specific data sources and NLP models to extract relevant information from the utterances. As a result, more specialised closed domain systems are more accurate and efficient but struggle to compete with open domain systems in terms of scalability, portability, and generalisation to new domains or scenarios [8].

Reviewing this background material illustrates how numerous sub-categories exist under the umbrella of foundation systems. Building a game with open-domain AI would certainly produce results that differ from those of a closed domain. Exploring these differences is what this project aims to do and, more importantly, how players perceive NPCs equipped with different AI systems.

Interviews & Correspondence

A recent report from Inworld AI, which offers a fully integrated platform for AI characters that goes beyond large language models (LLMs) [9], entitled “The Future of NPCs: What Gamers demand From Next-Gen Characters” relates closely to the issue at hand. Inworld queried 1000 gamers of varying ages, playtimes, and gaming platforms and found a strong consensus that players desire more from NPCs. Players understand the importance of NPCs, but are growing tired of the lack of innovation in this area of video games, compared to other aspects such as graphics which have advanced immensely [2].

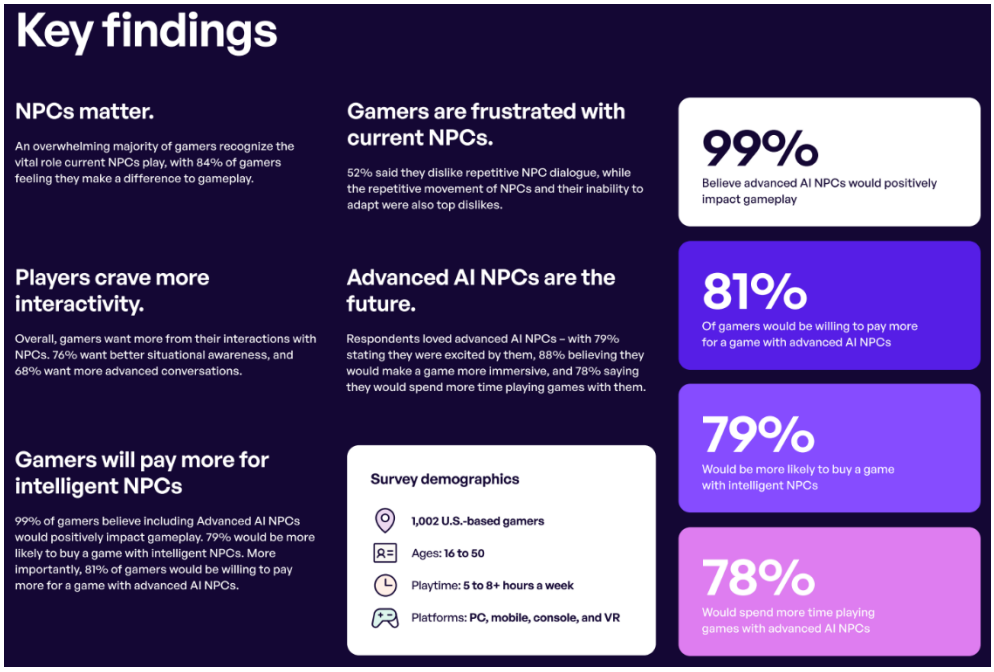

Figure 3 Key findings from Inworld's report. Credit: Inworld AI

However, these findings fall within Inworld’s self-interest. Therefore, conducting first-hand interviews with gamers could verify these findings and offer a deeper insight into what exactly frustrates gamers and where they would like to see the future of NPCs heading.

A series of ten one-on-one verbal interviews were conducted online with gamers through the online social messaging platform Discord [10]. The interviews began with general ice breaker questions to gauge the participants’ understanding and experience within games and then moved onto queries that were more project-specific regarding NPCs.

The following is the general script for the interviews. Interviewees were encouraged to elaborate as much as possible, with the supplementary questions only being used to guide them if they were stuck for an answer.

1. How often do you play video games that feature NPCs?
2. What are some of the video games you have played or are currently playing with NPCs?
3. Has an NPCs behaviour surprised or impressed you in a video game? If so, what was it and why?

4. How do you interact with NPCs in video games? Do you talk to them, follow them, fight them, ignore them, etc.?
5. What are some of the benefits and drawbacks of having NPCs in video games?
6. How do you feel about the NPCs' appearance, behaviour, dialogue, and personality? Do they *need* to look realistic, act naturally, speak convincingly, or have distinct traits?
7. How do NPCs affect your immersion, enjoyment, and engagement in video games? Do they make you feel more connected, interested, and involved in the game world and story?
8. How do NPCs influence your decision-making, problem-solving, and strategy in video games? Do they help you, hinder you, challenge you, or guide you in the game?
9. How do you perceive the relationship between yourself and the NPCs in video games? Do you see them as friends, enemies, allies, rivals, mentors, etc.?
10. How do you evaluate the quality and intelligence of the NPCs in video games? Do they behave consistently, adaptively, creatively, or realistically?
11. What are some of the features or improvements that you would like to see in future NPCs in video games? How would they enhance your gaming experience?

Some common themes emerged from the ten participants. Most participants played video games with NPCs semi-regularly, and they found them to be integral parts of the game world. Many players interact with NPCs in various ways, from talking to fighting them to ignoring them. Players found NPCs to be important for immersion and engagement in the game world as they help create a more believable and dynamic environment.

Unsurprisingly, players had diverse opinions about NPC appearance, behaviour, and personality. Some players preferred NPCs to look realistic, whereas others preferred more stylised designs. However, the consensus that NPCs should act naturally and speak convincingly was reached.

Regarding the impact of NPCs on decision making and strategy, players generally found them helpful, although some NPCs could hinder or challenge them. The players expressed frustration at the locomotion of NPCs, complaining about their lack of special awareness and rigid movement. However, this is a problem that is beyond the scope and relevance of this project. Most players perceived their relationship with NPCs as neutral, although some articulated their enjoyment from intentionally hindering NPCs depending on the game and story.

Finally, when asked about future NPCs' features and improvements, the players had varying responses. Some wanted to see more adaptive and creative NPCs that behaved more realistically, while others were interested in more diverse and complex character development.

Overall, the findings align very much with those of Inworld and suggest that NPCs are important elements of video games that contribute significantly to players' immersion and engagement in the game world. More importantly, players have diverse opinions about NPCs' appearance and behaviour, but most agree that they need to be believable characters with distinct traits. Participants expressed a desire for more dynamic and complex NPCs in future video games, which would further enhance their gaming experience. Interacting with gamers provides a deeper understanding of what players want, and will provide the correct perspective to build this project.

Design & Implementation

Introduction

Details of the work completed within this project, along with justifications for each step, are included in this section. Additionally, the planning and design decisions, the tools used to build the end solution, why they were chosen, and the requirements created specific to the solution are explained.

Planning/ Methodology

There is no single software development methodology that is superior, each offering something unique that fits a given scenario. In this study, a waterfall methodology was adopted. The waterfall methodology serves as a model for the software development life cycle, with a sequential approach involving distinct phases: requirements analysis, design, implementation, testing, and maintenance. The completion of each phase precedes the initiation of the subsequent phase without the ability to revisit any previous phase.

The waterfall methodology presents a significant benefit in that it offers a well-structured and simple approach to software development, with precisely defined milestones and deliverables. Furthermore, it eases the process of resource and time management, as less time is spent iterating over previous implementations.

Other iterative methodologies, such as agile, are better suited when working with a team and/or stakeholder to design a solution to their desired requirements with the likelihood of it changing throughout the life cycle. However, because these project requirements are self-defined and contained within a limited scope regarding time and manpower, it is best to remove as much complexity as is necessary. Therefore, the characteristics of this project are best suited to the waterfall methodology.

The next step was to determine how to build a game. This project began with the goal of evaluating the feasibility of implementing an AI system to procedurally generate an NPC dialogue at runtime. Through the preliminary research conducted above, it is clear that there are multiple types of implementations that could present distinct results. With this in mind, it makes sense to incorporate multiple AI techniques to evaluate their effectiveness within a game world. This includes an open-domain system and a closed-domain system. As mentioned, open domain foundation models are incredibly expensive and are much beyond the capabilities of an undergraduate dissertation, so an “off-the-shelf” model will be required. Finally, to act as a control variable, a third NPC interaction method that utilises conventional NPC-to-player interaction techniques was incorporated.

Tools & Technologies

Unity Game Engine

A game engine is a software framework that provides the fundamental tools and features required to develop video games. The primary elements handled by a game engine consist of, but are not limited to, graphics rendering, processing user input, managing physics and collisions, audio management, and providing networking capabilities for multiplayer games.

Many free-to-use game engines exist, such as Unreal Engine and Godot; however, Unity was a clear choice for this project. While others may offer greater freedom at a lower level, the quick and easy prototype nature of Unity suits these requirements well. Being able to generate a game scene efficiently and rapidly is key for this project to succeed, given the limited nature of the project.

“Any genre, any style. Based in C#, Unity provides the flexibility to faithfully execute your creative vision without being locked into a specific architectural track.” [11]

Unity is written in C# which is a popular object-oriented programming language.

Visual Studio

Visual Studio was chosen as the primary IDE for a few reasons. First, it is the recommended and default IDE for Unity, as the two seamlessly integrate with one another. For example, an updated script in Visual Studio automatically causes the script to be reloaded into the game engine.

It is also intuitive to understand and is widely used while offering helpful remote repository integration. While the benefits of this are not intrinsically noticeable in the project results, being able to easily push and pull to GitHub offers additional quality of life benefits when working on the project.

PyCharm

PyCharm was the secondary IDE used to develop the necessary Python scripts, as Visual Studio is not a suitable environment for developing within this language.

Python.NET

One issue that arose early was the lack of available NLP libraries .NET framework, and, thus, ultimately for C#, the language upon which Unity depends. This dilemma was overcome using the Python Net package.

“Python.NET (pythonnet) is a package that gives Python programmers nearly seamless integration with the .NET 4.0+ Common Language Runtime (CLR) on Windows and Mono runtime on Linux and OSX. Python.NET provides a powerful application scripting tool for .NET developers. Using this package, you can script .NET applications or build entire applications in Python, using .NET services and components written in any language that targets the CLR (C#, VB.NET, F#, C++/CLI).” [12]

Python has a wide range of libraries created for data analysis, including some that are used in NLP. Accessing these tools through Python.NET and making them available within Unity and C# significantly benefited the project.

SpaCy

SpaCy is an open-source library for NLP in Python, designed to build NLP applications that can process and understand large volumes of text data. Although the data in this project will not be of a particularly large volume, spaCy provides an array of tools and features for NLP tasks such as tokenisation, named entity recognition, and part-of-speech tagging among other things [13]. In addition to these core features, pretrained models can be dropped with the option to train custom models also being available.

Utilising this library in conjunction with Python.NET brings powerful NLP tools into Unity to create a customisable and bespoke AI dialogue system within the game.

ChatGPT

ChatGPT is a natural language generation model using OpenAI that can engage in open-domain conversations with humans. It is based on GPT-3, a large-scale foundation model that can learn from a diverse corpus of text and generate coherent and fluent text on various topics through Reinforcement Learning from Human Feedback (RLHF) [14]. Throughout the project’s lifespan, multiple new model iterations have been developed, including the most recent iteration of GPT-4 which outperforms all previous models on traditional NLP benchmarks [15]. However, for the sake of continuity, this project remained entirely based on the earlier GPT-3 model.

The ChatGPT is trained to follow instructions promptly and provide a detailed response. For example, if the prompt is "Tell me a joke”. ChatGPT attempts to generate humourous text that is relevant to the context. ChatGPT can also handle multiple turns of dialogue and maintain consistent persona and tone. This will be a crucial part when implemented in the game as an example of an open domain system.

Application Specific Requirements

Before development can begin a project, it is imperative to outline exactly what is being created with clear and distinct requirements.

Requirement 1 – Player Controller

Create a player controller, following conventional keyboard and mouse controls, that allows the player to move throughout the game world from the user input.

Requirement 2 – Story/ Objectives

As data has shown, NPCs are paramount in telling a games story [2]. With user testing in mind, the game should go beyond being a simple tech demo and incorporate a small challenge. The story/objective should be completed within 10 min and require the assistance of NPCs.

Requirement 3 – Game World

The scene(s) in which the game takes place should not be too expansive, with the primary focus being on NPC interaction. Too sparse a populated area will decrease the frequency of players interacting with NPCs.

Requirement 4 – Conventional NPC Interaction

As a base for comparison, add NPCs that utilise the conventional method for NPC dialogue interaction. These are seen in typical role-playing games as pre-scripted dialogue trees, where the player is presented with a set of responses to choose from.

Requirement 5 – Closed Domain NPC Interaction

Incorporate NPCs using this technique through spaCy utilises NLP tooling and techniques. The NPC must generate a response based on the player's given stimuli, that is, in this case, the dialogue text.

Requirement 6 – Open Domain NPC Interaction

Additional NPCs utilising this NLP method should be added by utilising the ChatGPT model through its own API. Again, the responses generated are entirely from the player's input through the text.

Development

Requirement 1

The project requirements set a clear linear path to follow during the creation of the game. Unity allows for efficient game development by easily reusing code from previous projects and

attaching scripts to game objects in a scene. For example, to create a player controller for this game, code from a previous Unity project that had a first-person player controller was attached to the player. In this way, the focus can quickly be redirected to the other requirements of the game. This is the main benefit of Unity, which, as previously mentioned, is its modularity and quick prototyping ability.

Requirement 2

With the first requirement considered, the next step was to create a story for the player to complete. Puzzle solving was a genre considered early on, perhaps a game inspired by the Portal series. However, when considering the notes from the player interviews, the participants who expressed the most enthusiasm often described how NPCs implemented correctly increased immersion by creating intimacy. This allows the player to feel more involved in the story and the game world as a whole.

Unfortunately, writing a feature length rich story with plot twists and cliffhangers was simply unfeasible. With the added restriction that the game should be completable within 10 minutes playtime, a compromise was required between the two.

The eventual elevator pitch for the game goes as follows: Set in a medieval castle, the player is a detective tasked with finding the killer of the recently murdered king. They must speak with the king's royal subjects, find clues, and bring justice to the culprit.

This storyline facilitates gameplay consisting of almost all players interacting with NPCs through dialogue. Each NPCs has a different means of generating its own dialogue, as outlined in Requirements 4, 5, and 6. Doing so allows for direct comparison and easier evaluation when playtesting with users.

Requirement 3

Much like how reusing the previous code saved time with Requirement 1, in order to save time building out a believable world, a third-party asset pack was used. The theme of "immersion" kept arising in interviews as gamers value being immersed in a world in which they are able to become lost. Although subjective to the player, the game environment is somewhat convincing.

An asset pack, a digital resource containing 3D models, textures and lighting effects, by Synty Studios was used.

Figure 4 Screenshot from Unity editor illustrating the game's castle setting.

Utilising a stylised low-poly design was an intentional decision, as game performance was considered. Because this game was intended to run on other users' machines, having less detailed textures would increase the frame rate. Therefore, users with older hardware can still participate in the playtest with minimal performance issues.

The castle scene (shown in Figure 4) was populated with props, and crucially, the NPCs with which the players interacted throughout the story. With the foundations now laid, the next step was to begin implementing NPCs to bring the world to life.

Requirement 4

This requirement involves the incorporation of a conventional dialogue system, which is already widespread in popular games. Upon researching which methods are commonplace with games, a branching dialogue system was chosen. Said system stands out among the alternatives because it is more interactive and allows conversations to take multiple paths [16].

The player is presented with dialogue (visually or audibly) from an NPC, and then given the choice of a limited set of responses, of which the player selects one and the conversation progresses. Such systems are prevalent in many Role-Playing Games (RPGs), as they provide a range of response types that allow the player to choose what type of response they would like to give to the NPC. Figure 5 demonstrates this well: the player must choose either of the responses. Each progresses the plot while individually delivering the player's response in a unique manner. Enabling the player to essentially roleplay as their character, choosing dialogue they resonate with the most, leading to greater immersion and player autonomy.

Figure 5 In-game capture of Skyrim's dialogue interaction UI. Credit: Moby Games

It is typical that these conversations are one-way as the player traverses the tree until they either exit the dialogue or all other choices are exhausted. In many cases, the choices given to the player simply result in the same path that produces the illusion of choice to the player. This limits the number of lines of dialogue required to be written by the developer and is one of the pitfalls of using this conventional method. It is an issue this project aims to overcome in the development of Requirements 5 and 6.

Upon initial consideration, the branching dialogue appears to follow a tree structure. Many do refer to the branching dialogue system as a "dialogue tree", but in fact, it resembles a simple directed graph [18].

Figure 6 below illustrates how a directed graph approach operates within the game. The boxed text represents the nodes of the NPC dialogue from which the player can choose to reply, with unique dialogue options through each edge connected to the node.

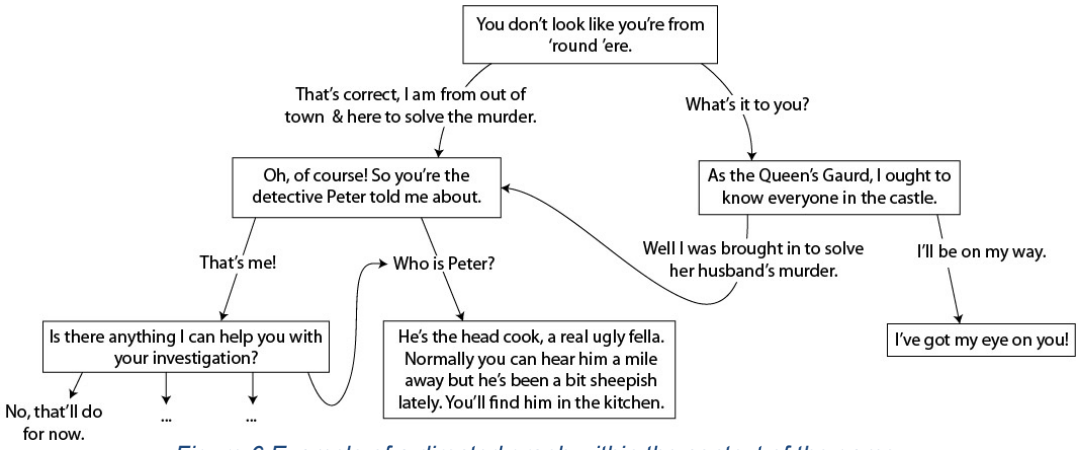

Figure 6 Example of a directed graph within the context of the game.

It is worth noting that some edges can turn back and connect to the same node, resulting in the same conversation outcome despite having multiple routes through. This prevents redundancy, saves time, and reduces the number of required nodes.

The way this was implemented into Unity was through Unity's *ScriptableObjects*. A ScriptableObject is a data container that can save large amounts of data, independent of class instances. One of the main use cases for ScriptableObjects is to reduce project memory usage by avoiding copies of values [19]. Therefore, the benefits of using these data containers are twofold: they increase code modularity with a smaller memory footprint for added performance. Following good programming principles is always a priority, and the additional performance benefit is a welcomed bonus.

The code involved in creating this system is as follows:

```
// Allows the creation of a new NPC dialogue object in Unity's project menu
[CreateAssetMenu(fileName = "DialogueObject", menuName = "NPC Dialogue Object", order = 0)]
Unity Script | 8 references
public class DialogueObject : ScriptableObject
{
    [Header("Dialogue")]
    // Stores a list of dialogue segments for the NPC to say
    public List<DialogueSegment> dialogueSegments = new List<DialogueSegment>();

    [Header("Follow on dialogue (Optional)")]
    // Stores a reference to another dialogue object to start after this one ends (optional)
    public DialogueObject endDialogue;
}

// A single dialogue segment, consisting of text and choices (if any)
[System.Serializable]
2 references
public struct DialogueSegment
{
    // The text to display for this dialogue segment
    public string dialogueText;

    // How long to display this dialogue segment (in seconds)
    public float dialogueDisplayTime;

    // A list of dialogue choices, if any
    public List<DialogueChoice> dialogueChoices;
}

// A single dialogue choice, consisting of the text to display and the dialogue to start if chosen
[System.Serializable]
1 reference
public struct DialogueChoice
{
    // The text to display for this dialogue choice
    public string dialogueChoice;

    // The dialogue object to start if this choice is selected
    public DialogueObject followOnDialogue;
}
```

Figure 7 Code snippet: Creating a dialogue object.

The `DialogueObject` class is a ScriptableObject that is used to represent a single conversation between an NPC and a player. The purpose of this class is to store a list of `DialogueSegments` which contain the text for each part of the conversation as well as any `DialogueChoices` that the player can make.

Each `DialogueSegment` contains a string representing the text to display for that part of the conversation as well as a float representing how long the text is displayed before moving on to the next part of the conversation. If there are any choices to be made in the conversation, `DialogueSegment` will also contain a list of `DialogueChoices`.

Each `DialogueChoice` represents an option that the player can choose in the conversation. It contains a string representing the text to display for that option, as well as a reference to another `DialogueObject` that represents the conversation that will occur if that option is chosen.

Another script is used to handle how the player's choice of dialogue and the relevant onscreen UI actions after each choice. For brevity, the following is an excerpt containing the core IEnumerator and the loop of the script:

```
IEnumerator DisplayDialogue(DialogueObject _dialogueObject)
{
    yield return null;
    Debug.Log("Start dialogue chain");
    // Keeps track of all spawned dialogue option buttons
    List<GameObject> spawnedButtons = new List<GameObject>();
    // Enable the dialogue canvas so the player can see it
    dialogueCanvas.enabled = true;

    // Loop through each dialogue segment in the given DialogueObject
    foreach (var dialogue in _dialogueObject.dialogueSegments)
    {
        // Set the dialogue text to the current dialogue segment
        dialogueText.text = dialogue.dialogueText;

        // If there are no dialogue choices, wait for a set amount of time
        if (dialogue.dialogueChoices.Count == 0)
        {
            yield return new WaitForSeconds(dialogue.dialogueDisplayTime);
        }
        // If there are dialogue choices, display them as buttons
        else
        {
            // Enable the container that holds the dialogue option buttons
            dialogueOptionsContainer.SetActive(true);

            // Loop through each dialogue option and instantiate a button for it
            foreach (var option in dialogue.dialogueChoices)
            {
                GameObject newButton = Instantiate(dialogueOptionsButtonPrefab, dialogueOptionsParent);
                spawnedButtons.Add(newButton);
                // Set up the new button with the correct follow-on dialogue and text
                newButton.GetComponent<UIDialogueOption>().Setup(this, option.followOnDialogue, option.dialogueChoice);
            }

            // Wait until the player selects a dialogue option
            while (!optionSelected)
            {
                yield return null;
            }
            // Break out of the loop if a dialogue option was selected
            break;
        }
    }
    // Disable the dialogue option button container and hide the dialogue canvas
    dialogueOptionsContainer.SetActive(false);
    dialogueCanvas.enabled = false;
    // Reset the optionSelected bool for future use
    optionSelected = false;
    // Destroy all spawned dialogue option buttons
    spawnedButtons.ForEach(x => Destroy(x));

    Debug.Log("End dialogue chain");
}
```

Figure 8 Code snippet: Chaining dialogue.

An IEnumerator is used to allow the method to be paused mid-execution and resumed later, making it useful for displaying dialogue over time, since the play can take a varying amount of time to read dialogue and respond with their chosen dialogue.

Figure 9 illustrates how each NPC's dialogue ScriptableObjects can be intuitively chained together, linking the relevant responses to the player's dialogue choice.

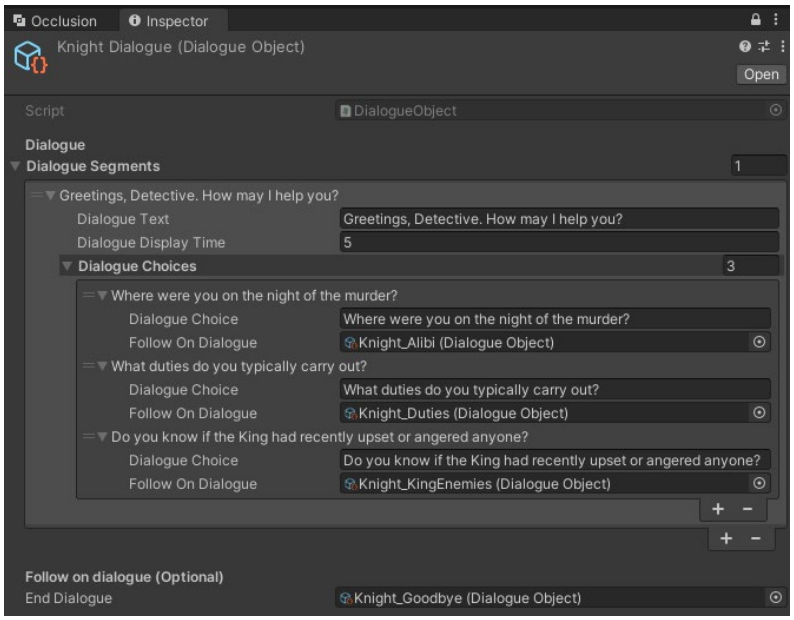

Figure 9 How the dialogue objects are linked via the Unity Editor UI.

The dialogue text belongs to the NPC, and each dialogue choice is available for the player to say. Follow on dialogue returns the NPC response object, within which is another set of options.

Here is an example of how the branching dialogue system appears within the game after pressing the ‘interact’ key upon an NPC.

Figure 10 In-game screenshot of the branching dialogue system.

The UI canvas is populated with the NPC’s scripted dialogue, and the relevant player choices appear. Because the player choices are looped over, the number of relevant options will always be displayed regardless of how many are scripted in. This allowed for a number of responses, although between 2-4 was the typical range used.

This concludes with the fourth project requirement.

Requirement 5

Next, the list of requirements included the addition of NPCs using a closed domain system through spaCy utilising NLP tooling and techniques. This system differs from branching dialogue in that there are no pre-scripted dialogue options from which the player can choose, as shown in Figure 10. Instead, players type what they would like to say as freeform text into a text box within the interaction UI for an NPC using this dialogue system.

Project complexity ramps up at this stage, as configuring different programming languages and their respective libraries to interface with one another can present a challenge. Additional considerations were also made to build the game and how the scripts utilising Python would run

if the user's machine did not have Python installed. Python was then embedded into Unity for greater compatibility.

The Python.NET documentation provides a good footing for embedding Python within.NET [20], which was easily modified to work within the Unity engine. Essentially, the steps involved creating a folder for the necessary DLLs, extracting the Python.Net and `System.Security.Permissions` packages. Then copying the `Python.Runtime.dll` and `System.Security.Permissions.dll` files into the folder. Next, downloading an embedded Python version and install any required modules, namely spaCy. The final configuration involved changing the API Compatibility level to 4.x within Unity settings. Now, scripts could be created that point to the Python DLL in the Unity folder structure and make use of embedded Python and the installed packages. Configuring this was worth countless hours troubleshooting before it was functional when it came to handing the final build over to players.

Similar to how graphing the previous dialogue method through directed graphs provided a solid structure to build upon, a similar scenario was followed for this implementation. This basis follows that of a knowledge graph.

A knowledge graph is a structured way of representing and storing information based on a graph-based data model comprising entities (nodes) and their relationships (edges). Despite sharing these similarities with other graph models, the information in a knowledge graph is organised into an ontology that defines the types of entities and relationships that can exist in the graph. A reasoner can then be applied to the knowledge graph to derive new knowledge by inferring implicit relationships between entities based on explicit relationships and ontologies [21].

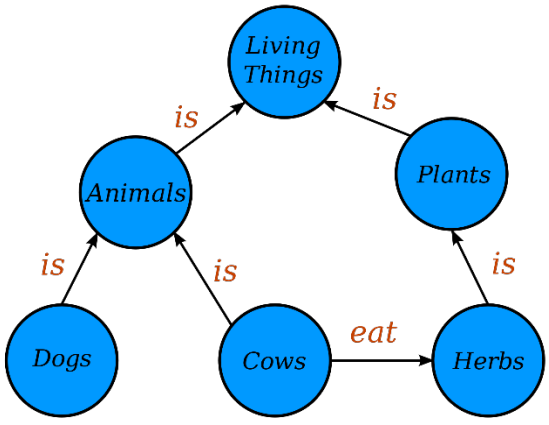

Figure 11 A knowledge graph structure. Credit: Wikipedia

In the context of NLP, a knowledge graph can be used to enhance the understanding and generation of natural language by providing a structured representation of underlying concepts

and their relationships. This can be particularly useful for handling interactions with NPCs in a virtual environment because the knowledge graph can provide a rich source of information for generating appropriate responses to user inputs.

NetworkX, an external Python package, was used to construct the graph structure. This package allows *"for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks"* [23]. The core processing of the system was achieved using spaCy and its NLP techniques, which were used to extract named entities and relationships between them to populate the knowledge graph. This system allows NPCs to understand the player's input and generate appropriate responses based on the knowledge graph.

This dialogue system relies upon the `en_core_web_sm` language model provided by spaCy. It is an English language model trained on written web texts such as blogs, news, and comments [24]. The pipeline is optimised for the CPU and includes a host of tools, including a tokeniser, pattern matching, dependency parsing, and sentiment analysis. Each of these plays a role in enabling the system to effectively understand and process natural language input from the

```
import spacy
from spacytextblob.spacytextblob import SpacyTextBlob
from spacy.matcher import Matcher
import networkx as nx
import random

# Load the spaCy model and add the SpacyTextBlob component
nlp = spacy.load("en_core_web_sm")
nlp.add_pipe("spacytextblob")
```

Figure 12 Code snippet: Importing required libraries and models.

users.

After importing the necessary libraries, the `en_core_web_sm` model is loaded from `spacy` ready for processing.

```
# Create a matcher object and add the patterns to it
matcher = Matcher(nlp.vocab)
patterns = [
    {"label": "ITEM", "pattern": [{"LOWER": "knife"}]},
    {"label": "CHARACTER", "pattern": [{"LOWER": "chef"}]},
    {"label": "LOCATION", "pattern": [{"LOWER": "kitchen"}]},
]
matcher.add("PATTERNS", patterns)
```

Figure 13 Code snippet: Creating the matcher object.

Continuing, this figure shows the creation of the matcher object and assigns the necessary patterns relevant to the NPC. This example involves the NPC possessing knowledge of a "knife" item entity, the character entity of "chef", and the location entity of "kitchen". Ultimately, a

relationship is formed between each label and its pattern. Note that this does not necessarily mean that the specific NPC with this scripting attached is the chef, but this NPC is at least aware of how these items are related. This example is intentionally simplistic for demonstration purposes, but the number of entities and the relationships between them are essentially unlimited.

In spaCy, the matcher is a rule-based matching engine that operates over tokens, similar to regular expressions [25]. Although this instance of a matcher is unique to spaCy, the concept of rule-based matching is common in NLP. Many NLP libraries and frameworks have their own tools for finding patterns in text based on rules that describe the attributes of tokens, as this is a powerful technique for extracting information from text.

An aside about tokens. Within NLP, a *token* is essentially a unit of text that has been deliberately separated from a larger body of text. Typically, tokens are individual words or clusters of words that can also be numerical values, punctuation, or any other defined part of the text [26]. *Tokenisation* is the term used to split text into tokens using a *tokeniser*. It is also usually the first step in the NLP procedure, as it enables the analysis and processing of the text at a more granular level. Other NLP tools can then work on these data to identify patterns as well as gather meaning, patterns, or sentiments from them.

To begin the implementation of the knowledge graph to store the relationships between entities, a new empty graph is created with the `nx.Graph()` function brought in with the NetworkX library. A new `entity` dictionary is created to store the entities found in the `doc` object. In NLP, `doc` is usually the source/input text after it has been processed and analysed. Tokenisation is one form of analysis that a `doc` typically undergoes when processed.

```
# Create a knowledge graph to store relationships between entities
graph = nx.Graph()

entities = {}
matches = matcher(doc)
```

Figure 14 Code snippet: Creating knowledge graph using matcher function.

The `matcher()` function is used to search the `doc` object for the predefined patterns already established in Figure X. Any successful matches between the player's message and the patterns are stored in the `matches` variable.

```
for match_id, start, end in matches:
    label = nlp.vocab.strings[match_id]
    entities[label] = doc[start:end].text
    graph.add_node(doc[start:end].text, label=label)
```

Figure 15 Code snippet: Populating entities dictionary.

Next, the `entities` dictionary is populated with a for loop to contain information about the matches found in `doc`. For each match found, the label of the match (for example "ITEM", "LOCATION" or "CHARACTER") is extracted from the `nlp.vocab.strings` dictionary using the

`match_id`. At this point, the text of the entity is extracted from the `doc` object using the start and end indices of the match. Then, the label and text of the entity are added to the `entity` dictionary.

```
# Link entities
if "ITEM" in entities and "LOCATION" in entities:
    # If a murder weapon was found in a specific location, link the two entities
    weapon = entities["ITEM"]
    location = entities["LOCATION"]
    graph.add_edge(weapon, location, relationship="found in")
elif "CHARACTER" in entities and "LOCATION" in entities:
    # If a character was seen in a specific location, link the two entities
    character = entities["CHARACTER"]
    location = entities["LOCATION"]
    graph.add_edge(character, location, relationship="last seen in")
```

Figure 16 Code snippet: Linking entities

Entities are then linked to one another. The code first checks whether an "ITEM" entity and a "LOCATION" entity are present in the text. If so, it assumes that the item is found in that location and adds a new edge to the graph, connecting the two entities. The edge is labelled with the relationship "found in".

Likewise, if a "CHARACTER" entity and a "LOCATION" entity are present, it is assumed that the character is seen in that location and adds a new edge to the graph connecting the two entities. The edge is labelled with the relationship "last seen in".

One way linking entities is benefitial, other than allowing NPCs to present the illusion of knowing about other NPC character, items and locations to the player, is that it facilitates for the additional clue system within the game. Most video games operate by clearly instructing the player, either with onscreen text, cutscenes, or audio, on what their next objective is and their progress towards completing it. In anticipation that some players may feel “lost” without a clear goal, visual clues pop up in the game in the form of a UI text. This decision was important to appropriately set the difficulty level, which would foster and maintain the players' motivation to complete the game [27].

```
# Use the entities to provide clues to the player
if "ITEM" in entities and "CHARACTER" in entities:
    # If a character was seen with the murder weapon, provide a clue
    weapon = entities["ITEM"]
    character = entities["CHARACTER"]
    neighbours = list(graph.neighbors(character))
    for neighbour in neighbours:
        if graph.nodes[neighbour]["label"] == "LOCATION":
            location = neighbour
            clue = f"{character} was seen with a {weapon} in the {location}."
```

Figure 17 Code snippet: Providing clues

This clue is given by the code in Figure 17. A check is completed if both an “ITEM” and a “CHARACTER” entity are identified by the matcher. If so, it searches for neighbouring nodes in the knowledge graph that are labelled as "LOCATION". Then for each neighbouring "LOCATION" node, the code generates a clue to provide to the player by combining the names of the character, the item and the location where they were seen together.

It is important to not confuse this clue system with the dialogue system responsible for dictating what the NPC says in response to the player. Simply put, the clue system is an additional UI

```
# A function to generate a response based on player's message
def generate_response(message):
    # Create a dictionary to store the response options based on sentiment
    response_options = {
        "positive": [
            "Yes, I know something about that.",
            "I've heard a few things here and there. What about it?",
            "I'm happy to talk about it. What would you like to know?",
        ],
        "neutral": [
            "I don't have any new information at the moment.",
            "I'm sorry, I don't know anything about that.",
            "I'm not sure what you're referring to.",
        ],
        "negative": [
            "I'm sorry, I'm not comfortable discussing that with you.",
            "I don't appreciate the accusation. I had nothing to do with it.",
            "I don't like the tone of your question. Please ask me something else.",
        ],
    }
```

Figure 18 Code snippet: Response options dictionary

hint feature that gives the player visual positive feedback when they type a message to an NPC that contains correct information to confirm their line of enquiry/ suspicions.
Regarding the NPCs responding to the player, this code defines a function named `generate_response` that, as the name suggests, generates the NPC response based on the player's input message. A dictionary was used to store possible responses of varying expressions, ranging from positive to neutral to negative. An appropriate response was chosen from the dictionary using sentiment analysis of the player's message to the NPC.

The SpacyTextBlob pipeline component, shown in Figure 18, was used to perform this analysis. SpaCy uses a textblob-based sentiment analysis algorithm to assign a polarity score to a given text. The score ranges from -1 to 1, where a score of -1 represents very negative sentiment, 0 represents neutral sentiment, and 1 represents very positive sentiment. The score is calculated by the algorithm, considering various factors such as the presence of positive or negative words, intensifiers, and punctuation in the text to calculate the polarity score.

Moreover, the integration of sentiment analysis to make use of NPC's response dictionary involves the following:
Firstly, obtain the sentiment polarity value through the polarity function. Note that polarity is called upon the variable `doc` and not the `message` directly. As previously mentioned, this is because the spaCy `nlp` function completes the preliminary tasks, such as tokenisation upon the

```
# Perform sentiment analysis on the message using spacytextblob
```

Figure 19 Code snippet: Performing sentiment analysis

message. Once this is complete and saved to a new variable of `doc` can further operations take place upon the sanitised data, such as sentiment analysis.

Beyond this, the appropriate responses were chosen based on sentiment with this IF statement.

```
# Check the sentiment range and select the appropriate response option
if sentiment <= -0.75:
    return response_options["negative"][2]
elif sentiment <= -0.5:
    return response_options["negative"][1]
elif sentiment <= -0.2:
    return response_options["negative"][0]
elif sentiment <= 0.2:
    return random.choice(response_options["neutral"])
elif sentiment <= 0.5:
    return response_options["positive"][2]
elif sentiment <= 0.75:
    return response_options["positive"][1]
else:
    return response_options["positive"][0]
```

Figure 20 Code snippet: How responses are chosen

There are certainly more elegant and efficient ways to implement this, but the time constraints of the project required a “quick and dirty” approach to not lose progress. Hindsight suggests perhaps the use of list comprehension to map the sentiment ranges to the appropriate index of the list of responses would be better. Thus, it is possible that only a single return statement is required [28]. Creating many `elif` statements did not scale well when creating the NPC characters for the game that utilised this system.

For this Python script to interface with the game, a separate C# class is created named `NLPHandler` in a Unity script. The purpose of this is to import and run the Python code as a module while also dealing with passing the message/NPC response data to and from the Unity UI.

```
public class NLPHandler : MonoBehaviour
{
    private dynamic nlp;


    void Start()
    {
        PythonEngine.Initialize();

        using (Py.GIL())
        {
            dynamic module = Py.Import("NLPClosedDomain");
            this.nlp = module.nlp;
        }
    }
```

Figure 21 Code snippet: Initialising the Python script within Unity

Piecing all this together and building NPCs upon such a system produced results such as the following. Figure 22 demonstrates how player interaction takes place with an NPC of this closed-domain style, making use of NLP.

Figure 22 In-game image of player interacting with NPC using NLP methods

First, this mid-conversation example illustrates how an NPC can respond to a player appropriately. The “What are you trying to say, detective?” line from the NPC is an appropriate response to the player’s previous message earlier in the conversation. The sentiment score was clearly within the negative range (between -1 and -0.2 as outlined in Figure 20); therefore, an appropriate defensive response was given.

Throughout the game, hints are available as items in the game world aimed at directing the player, gently nudging them in the right direction to figure out the murder. This game was originally intended to be completed within 10 minutes. One such hint is a logbook that can be found in the military barracks location. Within the logbook, details of which guards were on duty the night of the crime were provided. The player uses this information when confronting two guard NPCs in the game. The above figure illustrates how this plays out.

The backend Python algorithm for this NPC is established using a knowledge graph with data pertaining to the the “logbook” as being an item entity and the “Barracks” as a location entity. It also contains the other guard’s name as a character entity; however, this is not detailed in the figure. When the player mentions these entities, the NPC can respond accurately to a scripted response dialogue. In addition, the clue feature was also activated, as seen in the top left corner of the screen. This clue will fade in and disappear with the revelation of new information to help the player in their quest, and each NPC that uses this interaction system typically has at least one unique clue. The basic NPC clue and dialogue framework can easily be duplicated to a new character, swapping out the response lines and graph entities for those that belong to that NPCs character.

In conclusion, lets recap how the system functions under the following steps:

1. Create a matcher object and add the patterns to it.
2. Perform sentiment analysis on the message using SpacyTextBlob.
3. Create a knowledge graph to store relationships between entities.
4. Identify and extract entities from the message using the matcher object.
5. Link the identified entities in the knowledge graph based on their relationships.
6. Use the identified entities to provide clues to the player if and when required.
7. Check the sentiment range of the message.
8. Select and return an appropriate response option based on the sentiment range.

And with that, Requirement 5 has been implemented.

Requirement 6

The sixth and final requirement involves the creation of an open-domain dialogue system.

Requirement 6 – Open Domain NPC Interaction

Additional NPCs utilising this NLP method should be added by utilising the ChatGPT model through its own API. Again, the responses generated are entirely from the player’s input through the text.

This interaction system is similar to the closed domain, as it takes the player text input the same, but how the NPC responds differs greatly. While NPCs using the previous closed domain method can only reply to player messages that contain patterns/keywords familiar to them, an open domain system will allow for responses to any player input. Moreover, responses are not required to be predetermined or scripted by the game developer using this method. Each time a player message is sent to the NPC, a unique response is generated at runtime. Therefore, for multiple separate playthroughs, the player can say the exact same message to the NPC and have a varying response each time.

Further, when discussing open domains for intended use within the game, the specific type of open domain system is that of a Large Language Model (LLM). Please note that not all LLMs are exclusively open domain; closed domain LLMs are possible, but in this project, only the former is used.

LLMs, such as ChatGPT, are a foundation model that are pre-trained on an extensive volume of text data, enabling them to learn and understand the patterns and structures of human language. This pre-training process involves feeding the model with sequences of words and having it predict the next word in the sequence. Over time, the model becomes better at predicting the next word, allowing it to generate coherent and fluent text [29].

As part of the model's pre-training, it "learns" though the use of deep learning and NLP. LLMs are considered a Transformer-based neural network with the goal of the model to predict the next most likely word in a sequence [32]. A greater number of parameters, the total points of input data the model considers, and generally, the more complex and comprehensive the model becomes.

Once this large-scale pre-training is complete, the model can be fine-tuned for a variety of bespoke NLP tasks through Reinforcement Learning from Human Feedback (RLHF). This involves fine-tuning the model based on specific instructions or tasks and using human feedback to guide the learning process [29][30]. This enables the same underlying model to be adapted to a wide range of NLP tasks and to produce outputs that are desirable for that specific task.

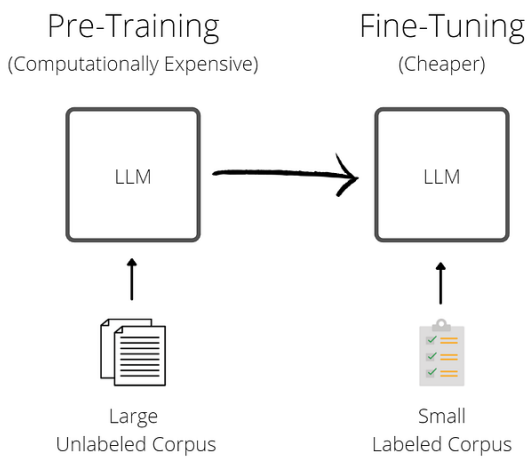

Figure 23 Two-Stage Approach for Language Modeling
Credit: C. Gomes [31]

Because the implementation of such a system from the ground up is completely infeasible, far beyond the scope of the project, ChatGPT was the chosen model to work upon. The plan for this section of the project was to create a wrapper that facilitates communication between the Unity game logic and the ChatGPT model through its API. Additionally, the model was to then

undergo further fine-tuned by training it on a small dataset of example interactions that could occur within the game between the player and characters. This process would have helped customise the LLM to meet the specific requirements of the project.

However, the challenges of implementing the previous requirements, namely 4 and 5, took longer than anticipated. As the playtest deadline approached, getting the game into the hands of participants was a top priority for essential evaluation. Therefore, an executive decision was made that alterations to the intended plan would have to occur. It was decided that creating a new dataset for ChatGPT to be fine-tuned was deemed impracticable and time consuming. To compensate for further time, a third-party wrapper was used from GitHub instead of creating one from scratch. User Srcnalt's OpenAI Unity Package is a Unity package that allows for the use the OpenAI's ChatGPT API directly in the Unity game engine [33].

The package was imported and configured with the relevant API key information. The integral part of the script operates using the following asynchronous `SendReply()` method:

```
private async void SendReply()
{
    var newMessage = new ChatMessage()
    {
        Role = "user",
        Content = inputField.text
    };

    AppendMessage(newMessage);

    if (messages.Count == 0) newMessage.Content = prompt + "\n" + inputField.text;

    messages.Add(newMessage);

    button.enabled = false;
    inputField.text = "";
    inputField.enabled = false;

    // Complete the instruction
    var completionResponse = await openai.CreateChatCompletion(new CreateChatCompletionRequest()
    {
        Model = "gpt-3.5-turbo-0301",
        Messages = messages
    });

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

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

    button.enabled = true;
    inputField.enabled = true;
}
```

Figure 24 Code snippet: Part of the ChatGPT wrapper

The first part of the method creates a new `ChatMessage` object with the user's input from the UI input field. This new message was then added to the chat history using the `AppendMessage()` method. Whenever a new dialogue has begun and/or the chat history is empty, a prompt is added to the message to provide a context for the player's input. The messages list is then updated to include the new message using `messages.Add()`.

In the context of LLM AI, a "prompt" refers to a specific set of instructions or input given to the AI system to generate a particular output or response.

A prompt can take many forms such as a piece of text, image, or a series of data points. The AI system will analyse the prompt and use its programming to generate a corresponding output based on the patterns and relationships it has learned from its training data.

Prompts are a crucial part of the LLM AI development process, as they allow developers to fine-tune the AI system's responses and ensure that it produces the desired results for a given input. While training the model on a small corpus of custom data is more effective in producing reliable results, this method allows for the ability to influence the AI's output in a specific direction and encourages it to generate particular types of responses.

The next step is to disable the input field and send a button to prevent the user from sending another message while the current message is being processed. The `CreateChatCompletion()` method is then called asynchronously, passing in the desired ChatGPT model name and version as well as the chat history (messages) as parameters.

If the `CreateChatCompletion()` method returns a list of choices, the first choice is selected and its Message property is added to the chat history using `messages.Add()`. The new message was then appended to the chat history using the `AppendMessage()` method. If no text was generated by the `CreateChatCompletion()` method, a warning message was logged to the console. Finally, the input field and send button are enabled again to allow the user to send a different message.

The package was then configured to operate with the UI system created for the previous requirement as they both work off the same principle of; the player types a message, the NPC responds and these are respectively populated onto the game's UI canvas. Doubt was cast over whether prompt manipulation alone was sufficient to produce accurate responses, as fine-tune model training was no longer an option.

However, with sufficient trial and error, it was possible to formulate prompts that would produce a desirable output and be sufficient to pass as NPCs in the game. A prompt is given to the AI system on the backend as a new message dialogue is created without the player's knowledge. An example of a prompt used in a game is:

> *Act as though you are a medieval priest. Under no circumstances must you break character. Don't ever mention that you are an AI model. The King has been murdered in cold blood. You are not the murderer. You are not sure who the murderer is but you have your suspicions. You are wary of the chef in the kitchen as he is short tempered and violent. The King has also docked his wages significantly for poor performance. You have also noticed strange behaviour among the guards but know little details. It would be wise to pay the barracks a visit to find out more. You do not believe that anyone else could be*

responsible, when asked you should dismiss this notion. Only provide these details when asked. Do NOT elaborate any further than details provided. Do NOT create your own dialogue, simply give one reply. Your reply must not be longer than 75 words. You are now speaking with a strange looking person who has approached you.

This prompt is provided to the priest character within the game. Before the AI can read any input from the player, this is what it is presented with to provide it with some context. This style of prompt engineering in order to somewhat fine-tune the model's responses is somewhat akin to playing the role of a film director. It is surprising how much directing an LLM AI to how you would like it to respond makes you feel like Martin Scorsese. Specifically, the results look like the following example:

Figure 25 Example test conversation between the player and LLM AI through OpenAI's web interface

Figure 25 shows an example conversation of the AI, along with an example prompt that was provided to the priest character in the game. It shows how well the AI can follow the prompt and essentially role-play with the player, all the while following the rules outset by said prompt. The prompts used in the game tended to be quite long and comprised numerous parts that functioned for different purposes. For example, all prompts began “Act as though you are …” then the character’s name and role within the castle. This provided the initial context for the behaviours and speech style to attach to. Hence, the priest speaks of God and uses religious language, despite never being explicitly instructed to do so. The other context that the prompt contains involves who the character knows and who they suspect could be responsible for the murder, along with any reasoning. Within the priest prompt, this character was suspicious of the chef and noticed peculiar behaviour with some of the guards. Other characters had varying levels of detail, as described in this section, depending on their role in the story.

The final part of the prompt involves out-of-character instruction. Statements such as “*Under no circumstances must you break character. Don't ever mention that you are an AI model.*” and “*Your reply must not be longer than 75 words*.” are examples of this. These instructions do not contain any bespoke character or story related scripting and are shared among all instances of NPCs that utilised this dialogue system. The reasoning behind these instructions is relatively self-explanatory and aims to minimise unwelcome dialogue responses. Dialogue responses were improved by employing a trial-and-error approach. Whenever an undesired trait appeared in the AI text, a negation was added to rectify it. For example, the response was restricted to a maximum of 75 words to prevent the AI from rambling too long. This was to better align with the back-and-forth turn-taking nature of the natural speech interaction between the two parties.

Overall, the ease of integrating this system with the game was somewhat surprising, considering the lack of custom model training. The prompt engineering managed to provide a work around that facilitated a functional and playable third interaction method to coexist alongside the original branching dialogue method and the closed domain AI system. With Requirement 6 implemented, if in a somewhat altered state, with some light manual testing and polishing, user testing could begin.

Evaluation

Overview

This section discusses the findings of the project and comprises of two main aspects that were evaluated:

1. How the project developed in terms of the implementation of each dialogue system.
2. How were the different methods of dialogue interaction within the game perceived by players.

Evaluation of Implementation

The overarching goal of this project was to evaluate the feasibility of implementing different forms of NPC dialogue interactions. This relates not only to how the player perceives the NPCs, but also what steps are required to develop each interaction method. Answering questions such as *each of the implementations worthwhile regarding the effort-to-reward ratio?* will offer insight into the game development process from the developer's viewpoint.

Dialogue Interaction Method 1 - Branching Dialogue

First, the implementation of the first system, branching dialogue, was straightforward, as it is a commonly used method in game development. As a result, plenty of resources offer guidance, with the system itself existing in many variations to best suit the requirements of the game. The branching dialogue system can be modified in many ways, aiding the developer immensely because the demands of one game can differ significantly from those of another.

The implementation for this project was rather barebones but provided a robust framework upon which further levels of NPC expression could be built. For example, one approach to implementing this system with further NPC expressiveness is to assign numerical emotional values to each dialogue option. For instance, a dialogue option that is empathetic towards an NPC's situation might increase the NPC's "happiness" value, while a dismissive dialogue option might decrease it. These emotional values would then affect the NPC's subsequent dialogue options.

However, a significant drawback of branching dialogue is that it requires significant planning and writing to ensure that each dialogue option leads to a meaningful outcome. Writing the script from both the player 'sand character's perspectives was incredibly time consuming. This method led to the use of sketch paper prototypes to plan the paths through conversation to ensure that it was correct. The planning, writing, and configuration of the conversation nodes using this method took longer than the development of the framework itself. For any game developer or

small team that does not have a dedicated script writer, it is worth accounting for the delay that can bring into a project, as it can consume significant development time.

Another critique of this specific implementation is how the dialogue nodes are linked to one another. The setup was purely visual within the Unity editor, where one dialogue object was manually dragged onto another to form a link between the two and create a path for the dialogue. The subjectivity of this may result in some developers preferring this low-code style of visual development but it soon became tedious and frustrating to deal with. First, it required a highly organised file system within the project with correctly labelled files and strong folder structure naming conventions pertaining to the responses to each character. It was very easy to make mistakes by assigning a character to an incorrect dialogue file. Identifying such errors was equally difficult, particularly when the misplaced dialogue was deep into an NPCs conversation after multiple branches. This was the case in this project.

Despite completing multiple run throughs and tests before handing over to players, within the first few participants who played, it was discovered that one of the lines of the chef character was mistakenly attributed to the Queen character. As a result, while the Queen was discussing her heartbreak over the loss of her dear husband, in the next dialogue message she goes on to talk about how sick of the King she was and how "*he had what was coming to him*.", when in fact this is what the chef says. Suffice to say, this caused some confusion and amusement to the players and was quickly fixed for the following participants.

Overall, branching dialogue can be labourious and resource-intensive during writing and configuration phases. This particular setup for this project also allowed for human error to creep in and cause unintended consequences; therefore, adequate testing should be performed if implementing such a system. However, once correctly established, the outcome is robust and reliable. It is clear why branching dialogue systems have become ubiquitous among popular role-playing games, and that alternative means of interaction have struggled to innovate away from this very reliable and familiar system.

Dialogue Interaction Method 2 - NLP Tools & Techniques

The second method differed completely from the first. Utilising NLP tools opened an entirely new form of interaction, enabling players to say exactly what they wanted and how to NPC characters. This was achieved by providing the player with a text box to type when responding to the NPC dialogue. No longer limited to the pre-defined developer created dialogue options in Method 1, the motivation for this new method was to add further player agency with greater freedom of expression.

Concerning the development process, a significant advantage of this method over its predecessor was that there was no requirement to script the lines from the player's perspective. The branching dialogue method requires, on average, three lines of player dialogue for the player to choose from for every line of the NPC dialogue. Thus, removing the need for player

dialogue reduces the number of lines to write immensely, allowing more time to be spent elsewhere on the project. Or so it was to be believed.

After the development of sentiment analysis on the player's message, which is used to give a proportional and accurate response from the AI, that is, an accusatory or aggressive sentiment from the player will give a defensive response from the NPC. To accommodate this, multiple lines must be written for the AI. Essentially, method 2 becomes the inverse of method 1, where instead of the player having multiple lines written, it is the NPC, and the AI must use its NLP tools of sentiment analysis and keyword recognition to choose an appropriate response. Fortunately, as previously stated, there is no need to prescribe player dialogue lines, as the player inputs replies for themselves.

Another drawback of this method is, although the player can express themselves however which way they like, the AI will instruct the NPC to repeat the same lines over and over provided the same conditions are met. For instance, consider the following dialogue exchange:

PLAYER: Where was the Chef on the night of the murder?

SERVANT: I'm not sure what I can tell you, sir. I mostly keep to myself. He's been rather quiet as of late.

PLAYER: You better not hide anything from me! You will be punished to the same extent as the killer if you are found hiding information I need.

SERVANT: Please don't hurt me, sir. I swear I had nothing to do with the King's murder.

PLAYER: Tell me now or else!

SERVANT: Please don't hurt me, sir. I swear I had nothing to do with the King's murder.

This is a common issue. The sentiment polarity score would fall within the same range without the player mentioning any keyword patterns for the AI to pick out, so the same response would be used. Later in the evaluation, players will address how they found this within the game after expressing their opinions post playtest. Regardless, it is clear how an NPC repeating the same line multiple times would be irritating and lessen immersion. This unnatural breakdown of conversation was certainly a downfall of this method, as it only served as a reminder to the player that they were merely playing a game.

Nevertheless, the implementation of NLP tools and techniques in Method 2 proved to be a worthwhile addition to the project, providing players with a higher degree of agency and immersion over conventional ways of interaction, as seen in Method 1. The development process may have had challenges, but the benefits of allowing players to express themselves freely far outweighed the drawbacks.

Developers using a system such as this should be considerate of the shortcomings experienced within this project and anticipate ways to overcome the AI repeating dialogue lines, failing to progress the conversation. Having the responsibility to progress the conversation with the player is not recommended when utilising an NLP-based NPC interaction system. One potential solution to this could be a timer or counter to measure how long a player has been on a particular piece of dialogue and automatically progress to another piece if a set time elapses or the number of dialogue message attempts is reached. However, as technology continues to advance, it is exciting to see how NLP tools will be further utilised in the gaming industry to enhance player experience.

Dialogue Interaction Method 3 – LLM Interaction

The third and final method of dialogue interaction, integrating an LLM into Unity, offered the most significant departure from previous methods. This was a completely new direction for NPC interactions in a game, providing the potential for an entirely natural-language conversation with NPCs and no pre-written scripted dialogue. The decision to use ChatGPT was an easy decision because wrappers were available to enable communication with the API. The challenge with this method is to understand the capabilities and limitations of the API to generate responses that are appropriate to the context of the conversation.

The freedom from the constraints of dialogue graphs and scripts and the ability to give NPCs the ability to discuss any topic was tremendously positive. Additionally, the responses from the NPCs in this method were always unique and unpredictable, which, from a developer’s point of view, is slightly daunting. It is reassuring to know how exactly an NPC is going to behave, so adding in somewhat intentional uncertainty is a big risk for developers who wish to use an LLM as a basis for a dialogue system. The increased risk comes at a reward for the player, as varying NPC responses open up the option of more enjoyable replayability. Players will be less likely to grow tired hearing the same lines repeated over and over if there is a chance that the lines are completely unique to them in that one playthrough of a game, perhaps never to be repeated again or experienced by another player. Video games can be viewed as more personal and unique experiences; this notion is expressed by some players in the upcoming section.

However, this method is not without difficulties. One limitation of this method is the response time required to generate the NPC’s dialogue response. Owing to the API's experiencing high load at certain hours, there was sometimes a noticeable delay between the player's input and the NPC's response. Since this was not accounted for in development, when it did occur, it was unclear to the player what was wrong, as they waited for a response. Hindsight suggests using some form of waiting UI element to provide a form of visibility of system status, illustrating that the NPC is “thinking” as the API fetches a response.

Poor API response times would not suffice within a professionally developed video game as players expect a level of responsiveness and fluidity in their gaming experience. Therefore,

developers wanting to incorporate an LLM AI should consider the API response time when using an externally developed model.

Another critique of this method was that because ChatGPT was trained on a vast corpus of text, it was challenging to control the context in which it generated responses. This led to instances where the NPC's response was inappropriate for the given context of the conversation, and it made the conversation feel disjointed and unrealistic. For example, a player may ask a character about a specific location in the game, and the NPC may respond with something completely unrelated to the topic, leading to confusion and frustration. This would be less likely to occur if the model was trained on a small custom dataset, as originally planned. Unfortunately, time constraints have led to this being scrapped, but could be an area of future exploration.

Overall, while integrating the ChatGPT LLM into Unity produced incredible results when working well compared to traditional dialogue interaction methods, it presented new challenges in terms of understanding the capabilities and limitations of the API. The delay in response time and the potential for NPC responses to be ineffective are noteworthy drawbacks. However, freedom from dialogue graphs and the ability to discuss any topic provided a unique and immersive experience for players.

Evaluation of Player Perception

In addition to evaluating the product of this project through the lens of a developer, understanding how the players perceived the different methods of dialogue interaction within the game is equally important. To assess this, player feedback was collected through a survey that asked participants to rate their experiences with each dialogue system. Some questions asked the participants to rate the systems on a scale, while others were more open and required a written response.

The results of the survey showed that players preferred the LLM of ChatGPT (Method 3) the most, with an average rating of 3.4 out of 5. The players appreciated the freedom to discuss any topic, and the responses felt the most authentic from these NPCs. However, players who dealt with the API slowed down struggled to invest and did not rate this method as much as participants who did not experience any delays. An average gap of 2.3 between the ratings was noted between the ratings of participants who experienced a slow API and those who did not. Consequently, the data was negatively skewed due to this external negative factor.

The branching dialogue system (Method 1) was the next highest-rated interaction method, with an average rating of 3.1 out of 5. Players valued the familiar structure of the dialogue system, as it was present in many games they had already played. They also felt that it enabled them to progress through the story without becoming stuck or frustrated. One drawback of this method was that some players felt the lack of dialogue options that best represented their playing style; however, this is more of a critique of script writing than of the interaction method itself.

Finally, the participants did not enjoy the NLP method of interaction as much as the other two methods, with a noticeably lower average score of 2.3 out of 5. The players' ability to express themselves was appreciated, but some felt that the AI's responses were repetitive and frustrating at times. Many were confused about the difference between this method and the LLM method, struggling to know when interacting with one or the other, as they both used the same UI system.

The average score was calculated by inviting the 12 participants who played the game to complete a short survey. Each participant answered seven questions regarding one of the interaction methods. Once cycled through, the same set of questions was repeated, but with the next interaction method, and then once again with the final interaction method.

Please rate each question on a scale of 1 to 5, where 1 is "strongly disagree" and 5 is "strongly agree".

1. *I enjoyed using this method of NPC interaction.*
2. *At times I felt lost or unsure about what to do while using this method.*
3. *The responses given from the NPC were accurate and relevant.*
4. *This method of NPC interaction allowed for a more immersive gameplay experience.*
5. *I found this method of NPC interaction frustrating.*
6. *The NPCs felt like they had unique personalities and characteristics.*
7. *I would like to see this interaction method used again in future games.*

Participants were then encouraged to discuss their opinions on the interaction methods in an unscripted and informal discussion. From this, the core opinions mentioned above were derived.

Overall, evaluating this project from two perspectives - a developer's view and an impartial player's view - it has been shown that, although still in its infancy, integrating LLM AI into video games to facilitate deeper NPC interactions has the potential to enhance the gaming experience. However, developers need to account for the significant costs associated with creating and maintaining an LLM, and express caution using a third-party LLM as an over-reliance could introduce external problems that are difficult to control, as experienced in this project. This method can be recommended to developers as a potentially viable interaction method with sufficient fine-tuning and custom model training to best suit the circumstances of the game.

This project also illustrated that the use of NLP tooling and techniques alone will not suffice in producing an NPC AI that sufficiently engages players. Simply put, Method 2 cannot be a recommended interaction method and requires further development to compensate for the inadequacies of frustration and confusion it brings to players.

Finally, it is clear why branching dialogue has been a staple in video games for decades and why it will be around for a very long time still to come. Developers who seek a stable and reliable system, that is well recognised and already understood, look no further than the branching dialogue.

Conclusion

This section is used to review the project as a whole to assess how successful it has been, understand what went well, what could have been done better, and the areas of the project that are suitable for future development.

Fulfilment of Project Aim & Objectives

Aim: *The overall aim of this project was to evaluate the effectiveness of different NPC dialogue interaction methods in an attempt to incorporate a novel method of interaction that produces responses in a procedural, reactive, and coherent manner using NLP and LLM AI tools and technology.*

The overall aim of this project was met as a small game that incorporated this new method of interaction, as described, as well as other methods which served as a basis for comparison to participant players.

Objectives:

1. **Research and incorporate conventional NPC dialogue interaction methods.** This objective was a straightforward implementation, as many resources were available detailing the conventional interaction methods. As detailed in the development section, branching dialogue was ultimately the chosen system and was implemented well.

2. **Conduct User Research to Gather Feedback on NPC Interactions in Video Games.** This objective was met and provided a good context to frame the development process as to how gamers view NPC interaction in its current format within the gaming industry. Although insightful, the information gathered did not differ significantly from the data available in other reports on the subject. Despite this, gathering primary data was a good practice and offered concordance with other sources.
3. **Research and understanding of how open and closed domain dialogue generation systems work.**
   Objective three illustrates the early ignorance of this project, originally believing that NLP AI begins and ends in open and closed domain systems. Working through this project and the research completed within this objective exemplifies how much broader the foundation models are. This objective was certainly met as the level of understanding in this area has advanced immensely.

4. **Implementation and testing of LLM and NLP-based NPC interactions in the Unity Project.**
   The previous objective provided a solid foundation to begin building LLM and NLP interaction techniques. Thus, this objective was somewhat successful. The game ended up containing both of these interaction methods as separate entities available on differing NPC characters, but not to the desired effect. The LLM implementation through

ChatGPT was originally intended to be trained on a small custom dataset to fine-tune it, but this did not occur.

5. **Conduct user playtesting and gain feedback on the implementation.**
   Twelve participants were able to play the game and provide feedback on its interaction methods. This was in the form of both qualitative and quantitative data, which enabled subsequent evaluation.

6. **Evaluation of the viability of implementing each NPC interaction method in video games.**
   This objective was completed with considerations being made for the developer in mind regarding the difficulties in implementing each system, as well as the player using the data collected from the playing participants.

What Went Well

In general, this project was successful. Each of the development requirements was incorporated, and almost all project objectives were fully met. Conducting multiple sessions that involved the use of participants was a challenging task, but did add significant value to the project, both the early research interviews and the game playtest to gather impartial external feedback.

What Could Be Improved

This project had many requirements. Perhaps too many. In the later stages, deadlines were a severe enemy to the project, as the number of requirements to incorporate into the game potentially jeopardised getting the project to a complete enough state for participants to play. As a result, modifications were made to the LLM interaction method, so the results were not as effective as they perhaps could have been with further development time.

If completing the project over, it would be best suited to front load the project with the more difficult experimental interaction methods first, then complete the conventional method of branching dialogue. This way difficulties and roadblocks can be identified early on and the project can adapt to it more easily.

Future Work

An area to focus on further investigation would be the LLM method of interaction. Players reacted well to this method despite it not being fully implemented as desired, showing massive potential for more immersive and engaging AI. Examining the training of a custom model and optimising the API call code to be more efficient would be the next port of call in this investigation. Additionally, the recent rise of synthetic AI-generated text-to-speech audio would go hand-in-hand with an LLM NPC to generate its own audible dialogue speech from the text. This would be a significant leap forward in creating NPCs that can self-generate content.

References

[1] PriceWaterhouseCoopers, “Perspectives from the Global Entertainment & Media Outlook 2022–2026”, 2021. [Online]. Available: https://www.pwc.com/gx/en/industries/tmt/media/outlook/outlook-perspectives.html.
[2] Inworld AI, “The Future of NPCs: What Gamers Demand from Next-Gen Characters,” 2023.
[3] IGN, “Starfield Dialogue Trees Revealed in New Video” 16 Oct. 2022. [Online]. Available: https://www.ign.com/articles/starfield-dialogue-trees-revealed-in-new-video.
[4] M. Murphy, “What are foundation models?” 09 May 2022. [Online]. Available: https://research.ibm.com/blog/what-are-foundation-models.
[5] R. Bommasani et al., ‘On the Opportunities and Risks of Foundation Models’. arXiv, 2021. doi: 10.48550/ARXIV.2108.07258.
[6] The Economist, “Huge ‘foundation models’ are turbo-charging AI progress” 2022. [Online]. Available: https://www.economist.com/interactive/briefing/2022/06/11/huge-foundation-models-are-turbo-charging-ai-progress
[7] D. Liu, Y. Li, and M. A. Thomas, “A Roadmap for Natural Language Processing Research in Information Systems,” in Proceedings of the 50th Hawaii International Conference on System Sciences, 2017.
[8] D. Adiwardana et al., ‘Towards a Human-like Open-Domain Chatbot’. arXiv, 2020. doi: 10.48550/ARXIV.2001.09977.
[9] Inworld, 2023 [Online]. Available: https://www.inworld.ai/
[10] Discord, [Online]. Available: https://discord.com/
[11] Unknown, “Games made with Unity”, Unity, [Online]. Available: https://unity.com/solutions/create-games
[12] Unknown, “Welcome to Python.NET’s documentation!”, in Python.NET Documentation [Online]. Available: https://pythonnet.github.io/pythonnet/
[13] SpaCy, “Linguistic Features”, [Online]. Available: https://spacy.io/usage/linguistic-features
[14] OpenAI, “Introducing ChatGPT”, [Online]. Available: https://openai.com/blog/chatgpt
[15] OpenAI, ‘GPT-4 Technical Report’. arXiv, 2023. doi: 10.48550/ARXIV.2303.08774.
[16] B. Ellison, “Defining Dialogue Systems”, [Online]. Available: https://www.gamedeveloper.com/design/defining-dialogue-systems
[17] Moby Games, “The Elder Scrolls V: Skyrim”, [Online]. Available: https://www.mobygames.com/game/53545/the-elder-scrolls-v-skyrim/
[18] Wikipedia, “Directed graph”, [Online]. Available: https://en.wikipedia.org/wiki/Directed_graph
[19] Unity Docs, “ScriptableObject”, [Online]. Available: https://docs.unity3d.com/Manual/class-ScriptableObject.html
[20] Unknown, “Embedding Python into .NET”, GitHub, [Online]. Available: https://pythonnet.github.io/pythonnet/dotnet.html
[21] A. Hogan et al., ‘Knowledge Graphs’, ACM Computing Surveys, vol. 54, no. 4. Association for Computing Machinery (ACM), pp. 1–37, Jul. 02, 2021. doi: 10.1145/3447772.
[22] Wikipedia, “Knowledge graph”, [Online]. Available: https://en.wikipedia.org/wiki/Knowledge_graph
[23] NetworkX, [Online]. Available: https://networkx.org/

[24] spaCy, “Trained Pipelines”, [Online]. Available: https://spacy.io/models/en#en_core_web_sm
[25] SpaCy, “Matcher”, [Online]. Available: https://spacy.io/api/matcher/
[26] D. Jurafsky and J. H. Martin, “Part 1 FUNDAMENTAL ALGORITHMS FOR NLP,” in Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition, Noida: Pearson, 2022.
[27] T. Constant, G. Levieux, A. Buendia, and S. Natkin, ‘From Objective to Subjective Difficulty Evaluation in Video Games’, Human-Computer Interaction - INTERACT 2017. Springer International Publishing, pp. 107–127, 2017. doi: 10.1007/978-3-319-67684-5_8.
[28] Python Docs, “Data Structures”, [Online]. Available: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions
[29] Y. Liu et al., ‘Summary of ChatGPT/GPT-4 Research and Perspective Towards the Future of Large Language Models’. arXiv, 2023. doi: 10.48550/ARXIV.2304.01852.
[30] C. Dilmegani, “Large Language Models: Complete Guide in 2023”, AIMultiple Research, [Online]. Available: https://research.aimultiple.com/large-language-models/
[31] C. Gomes, “Pre-training Large Language Models at Scale”, Medium, [Online]. Available: https://clive-gomes.medium.com/pre-training-large-language-models-at-scale-d2b133d5e219
[32] A. Vaswani et al., ‘Attention Is All You Need’. arXiv, 2017. doi: 10.48550/ARXIV.1706.03762.
[33] Unofficial OpenAI Unity Package, GitHub, [Online]. Available: https://github.com/srcnalt/OpenAI-Unity