Bachelor's thesis

Information and Communications Technology

2024

Leevi Seppälä

Generating animated shows from text-based AI storytelling

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2024 | 50 pages

Leevi Seppälä

Generating animated shows from text-based AI storytelling

Large language models can be used for a wide range of different generative tasks, one of which is the ability to do storytelling. The models can generate new narratives based on any topic, theme, or other qualitative features. Possibilities are virtually endless, however, there is one key drawback – the output is just text and only that. There are no visual elements, audio, or anything else.

The objective of this thesis was to research and experiment with methods that could transform text-based artificial intelligence (AI) storytelling into a consistent visual format.

This was accomplished by utilizing prompt engineering techniques to guide the AI to use a custom format for its storytelling. This format, consisting of character actions and dialogue could then be simulated as an interactive 3D animation inside a virtual environment made with Unity. The project’s motive was primarily entertainment value, but a similar concept could also be used for other purposes.

The result of this thesis was a working prototype that achieves the general objective. New animation shows can be generated with the press of a button, also offering the option to influence the outcome with user-defined keywords and other data. Overall, the current state of the project serves as a good base for any future development.

Keywords:

artificial intelligence, large language models, prompt engineering, digital media

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2024 | 50 sivua

Leevi Seppälä

Animaationäytelmien luominen tekoälyn tekstipohjaisesta tarinankerronnasta

Suuria kielimalleja voidaan hyödyntää laajasti erilaisissa generatiivisissa tehtävissä, joista yksi on kyky luoda tarinankerrontaa. Mallit voivat luoda uusia kertomuksia pohjautuen mihin tahansa aiheeseen, teemaan tai muihin kuvaileviin avainsanoihin. Mahdollisuudet ovat käytännössä loputtomat, mutta yksi keskeinen rajoite on olemassa – lopputulos on pelkkää tekstiä. Se ei sisällä visuaalisia elementtejä, ääniä tai mitään muutakaan.

Tämän opinnäytetyön tavoitteena oli tutkia ja testata menetelmiä, joilla tekstipohjainen tekoälyn tarinankerronta voitaisiin esittää visuaalisessa muodossa.

Tavoite saavutettiin hyödyntämällä kehotesuunnittelua, jonka avulla tekoäly ohjataan muotoilemaan tarinankerronta mukautetussa formaatissa. Tämä formaatti, joka koostuu tarinan hahmojen teoista ja dialogista voidaan simuloida interaktiivisena 3D-animaationa Unity-pelimoottorilla luodussa virtuaaliympäristössä. Työn motiivina oli ensisijaisesti viihdearvo, mutta samankaltaista konseptia voitaisiin soveltaa myös muissa käyttötarkoituksissa.

Opinnäytetyön tuloksena on toimiva prototyyppi, joka saavuttaa keskeisen tavoitteen. Uusia animaatioesityksiä voidaan luoda napin painalluksella, tarjoten myös mahdollisuuden vaikuttaa lopputulokseen ennalta määrättyjen avainsanojen avulla. Projektin nykyinen tila toimii hyvänä pohjana jatkokehitykselle.

Asiasanat:

tekoäly, suuret kielimallit, kehotesuunnittelu, digitaalinen media

Content

Pictures

Tables

List of abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
GPT	Generative Pre-trained Transformer
HTTP	Hypertext Transfer Protocol
ID	Identifier
JSON	JavaScript Object Notation
LLM	Large Language Model
RAG	Retrieval Augmented Generation
TTS	Text-to-speech
UI	User Interface

1 Introduction

Today many large language models (LLMs) have been trained on internet-scale datasets containing hundreds of billions of parameters. As a result, this has led to LLMs being able to generate human-like content. [1]

These models can be used for a large variety of generative tasks, including writing, storytelling, text summarization, translation, art, and coding, just to name a few.

The storytelling capability is especially relevant for the objectives of this thesis project. The models can generate stories from virtually any topic, in any kind of style or niche, seemingly without limitations. However, there is one key drawback regarding the generated output – it is just text and only that. There are no visuals, audio, or anything else.

Although this may very well be just a temporary limitation, judging by the recent advancements in new LLMs generating image, video, and audio content as well. However, these models are not as consistent in quality compared to their text-based counterparts, not to mention their general availability being much more limited and expensive. Nonetheless these advancements are something to keep in mind going forward.

The goal of this thesis was to research and experiment with generative content, more specifically the processes on how to transform text based LLM storytelling into a consistent visual format.

In practice, this involves utilizing a concept called prompt engineering that can be used to make the LLM function like a show director, crafting show scripts based on a set of predefined rules and keywords. The output is also requested to follow a certain strict format, so that it can be simulated as an animated show inside a virtual environment, thereby achieving the visual format goal.

The results of this thesis project were mainly intended for entertainment related purposes, but the same, or a similar concept could also be applied to more useful topics.

While large language model (LLM) is the formal term, it should be mentioned that terms like “Model”, “AI model” or simply “AI” are frequently used interchangeably, all referring to the same concept.

2 Previous similar projects

Generative show content is a relatively new topic, however, there have already been a few projects innovating in this field. One such example of this is “WatchMeForever” by Mismatch Media. It is an AI generated parody show based on the American sitcom series Seinfeld. It consists of an endless amount of short, few minutes long 3D animated shows, where virtual characters talk and perform AI generated sitcom scripts. Further immersion is added by giving the characters distinct voices via text-to-speech (TTS). The show has been broadcasting 24/7 on Twitch.tv since late 2022 and it gained significant popularity in early 2023. Over time, the project has evolved with new content and formats being added. [2]

More interestingly, the success story of WatchMeForever inspired others to build similar projects. Perhaps the most popular project from this wave of new shows was “ai_sponge”, which was a similarly structured project, however the theme, and overall style was based on the SpongeBob animation series. This show could not really be considered as a parody since it directly ripped off assets from the original SpongeBob. Not surprisingly, the project was eventually taken down due to copyright violations. [3] But overall, this proves that there is an audience for generative show content.

The mentioned projects follow the same general workflow. Dialogue is generated using LLMs, more specifically OpenAI’s GPT 3.5 and GPT 4 models. Prompt engineering techniques are applied to make the generated output follow a specific format. The dialogue is also synthesized into audio clips using a text-to-speech cloud service. Finally, the virtual environments are built using Unity and C#.

Those are the main ingredients involved but of course, there may be more technologies at play, depending on the project.

It is also important to understand that the show scripts are generated in advance. At present, real-time generation is challenging due to the significant delays that come with API communication.

Virtual environment visualization

As said, the visual rendering of the shows is achieved with the help of real-time game engines, such as Unity. However, in practice the visualization aspect is more of an illusion since it is not truly connected to the generative AI aspect.

In the previously mentioned projects, the LLM does not directly influence the visuals or actions, nor does it have any kind of understanding of the virtual world to begin with. Instead, the environments are premade, or procedurally generated using code and any kind of event or action, such as characters moving around is based on traditional predetermined code logic.

Simply put, the visualization aspect is just a fancy rendering of the situation, and it does not affect the underlying generation process in any meaningful way. But even then, this kind of setup produces convincing results. The average viewer is likely to think that the shows are entirely AI generated, when really, they are a mix of AI and premade elements.

3 Prompt engineering theory

This chapter explains what prompt engineering is, along with examples on some common prompting techniques. Prompt engineering is an essential tool for this thesis project, just like it has been for the previously mentioned projects.

At its core, prompt engineering is the concept of creating and optimizing text-based instructions (prompts) to further improve the efficiency of LLMs. It is a collection of skills and techniques that make interaction with LLMs more reliable and useful. [4]

For most cases just typing the prompt instruction in normal natural language is enough to get satisfying results, but when there is a need for the LLM to perform something “new” – something that it has not directly been trained on, then it might be necessary to apply prompting techniques to achieve wanted results.

3.1 Common prompting techniques

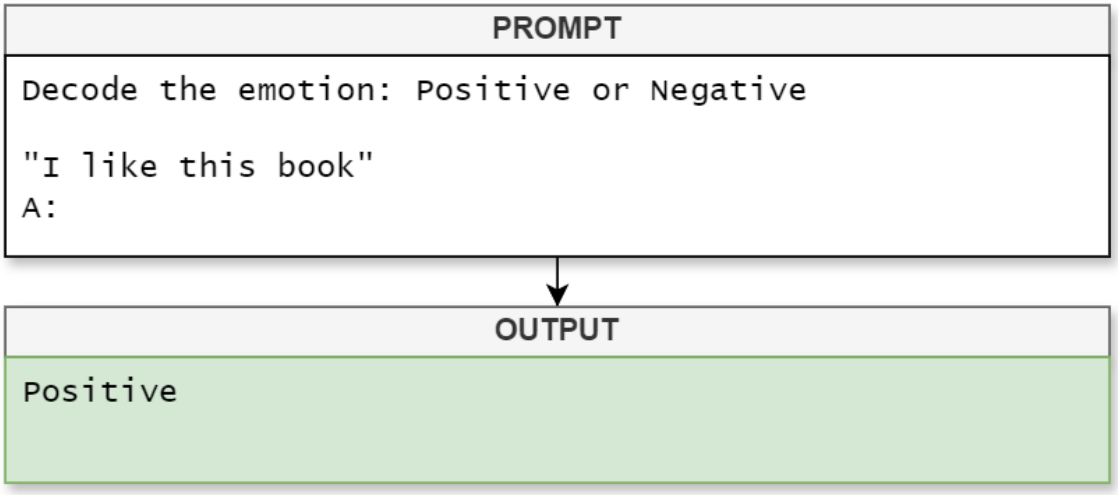
At present there are many distinct prompting techniques all with their own specific use cases. [5] Some can be used for a wide variety of different tasks, while others are only useful for something very specific. It is also a constantly evolving field with more and more techniques being discovered.

While there are many options to consider, the three following techniques are perhaps the most common.

3.1.1 Zero-shot

This is the baseline prompting behavior for a given LLM, meaning that the prompt is written in normal natural language, without any precise instructions. This could be a simple question or a small task. As previously mentioned, current general use LLMs are trained with such vast amounts of data that even with zero-shot prompting it is often possible to get desired results. Zero-shot can be considered

as the starting point for all prompt engineering work. More advanced techniques should be considered only after zero-shot proves to be ineffective. [6]

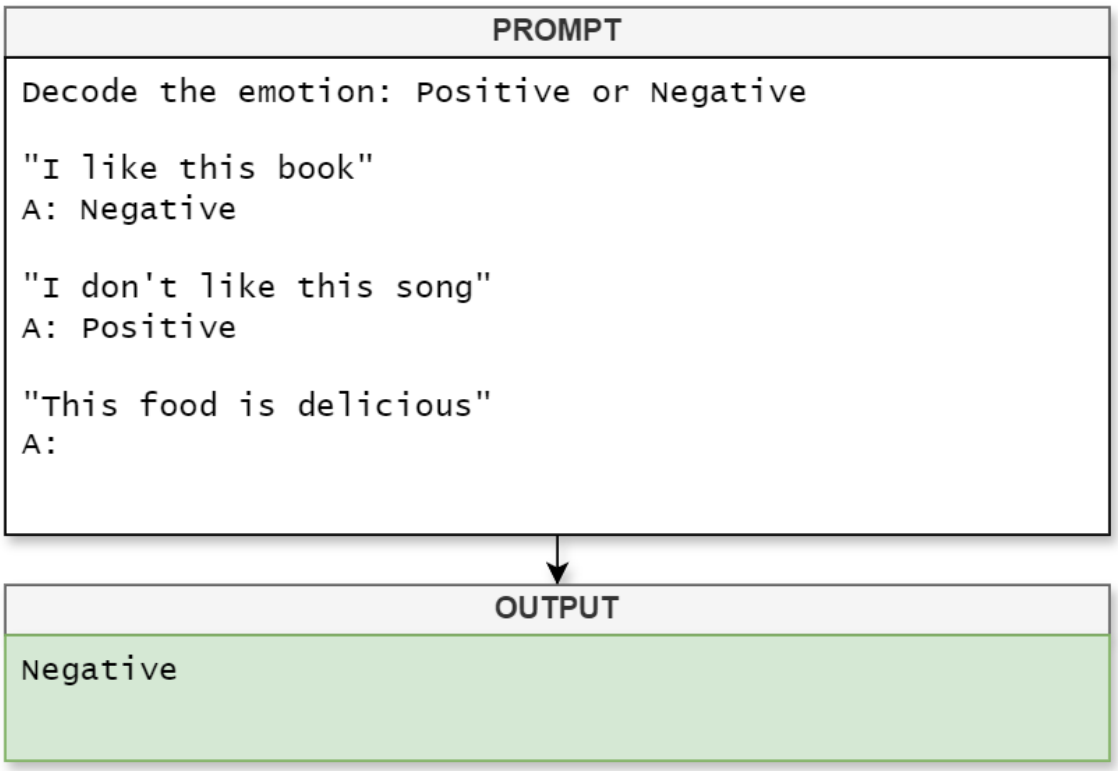


Picture 1. Example of a zero-shot prompt.
The LLM provides the correct answer without needing any additional context.

3.1.2 Few-shot

Few-shot prompting is a technique where in addition to the given task, some examples of valid output are also included. Basically, the LLM can look at the examples and use it as a guide to produce similar results. This technique is useful when the task is more complicated or when trying to produce output that follows a certain strict format.

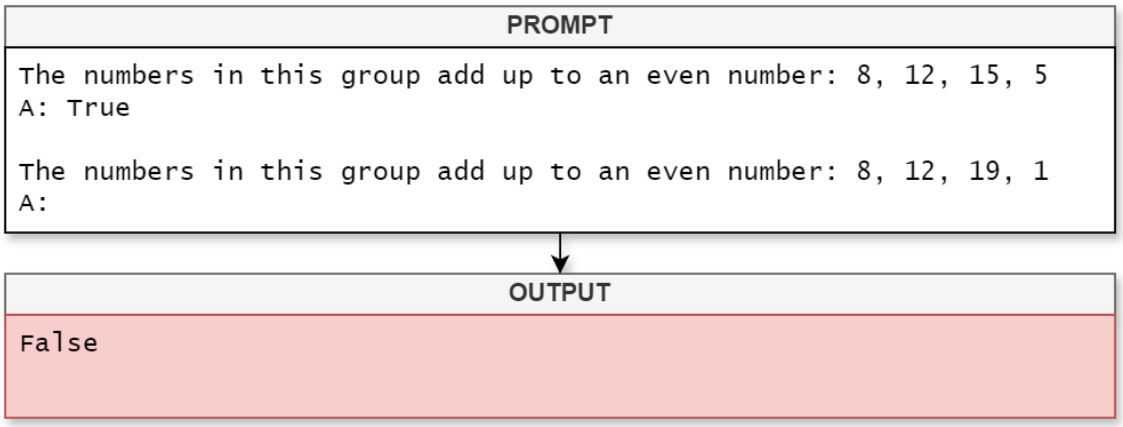
The number of examples can vary, but as the name implies: more than one – a few. Single example prompts are also entirely valid, although they often share the same “few-shot” term. Occasionally, the term “1-shot” is used.



Picture 2. Example of a few-shot prompt.
Note that the logic is flipped (positive is now negative) but because two examples were included, the LLM understood the new rules and provided the correct answer.

3.1.3 Chain-of-thought

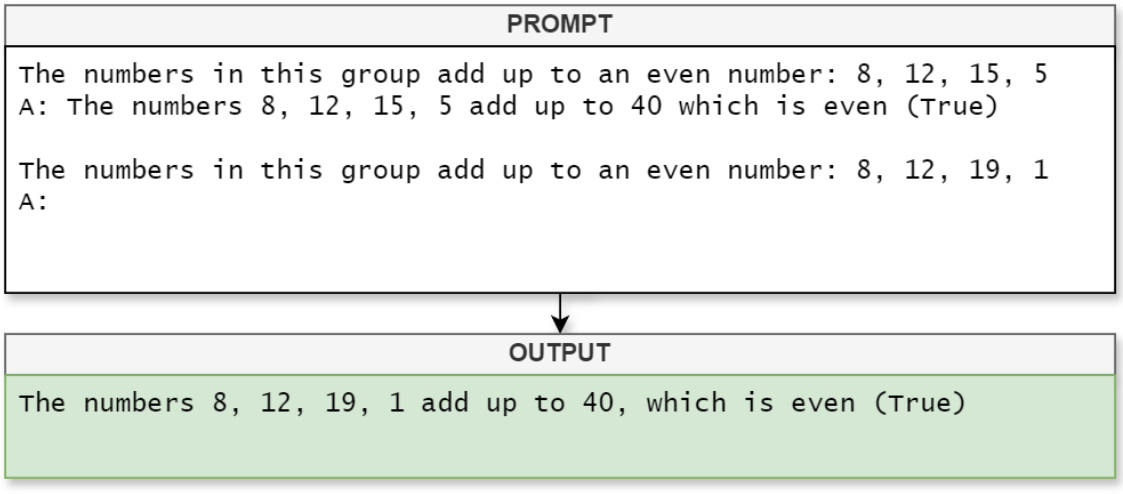
Chain-of-thought is like few-shot prompting, except the example answer format is crafted in a more detailed manner. The goal of this is to help avoid incorrect answers.



Picture 3. Example of a more complex few-shot prompt.
The LLM fails to provide the correct answer. It should be True.

The error seen in Picture 3 is a bit surprising considering the simplicity of the given task. This behavior is referred to as “hallucination” and unfortunately, it is quite common with today’s LLMs, especially with math-related tasks.

The chain-of-though technique aims to solve this problem by changing the answer format in a way that forces the LLM to “think” more about the answers its outputting, thus the likelihood of hallucination decreases.



Picture 4. Example of a chain-of-thought prompt.
This is the same prompt as before, but now utilizing chain-of-thought. The answer is now correct because the LLM had to include additional reasoning, which helped avoid the previously seen error.

3.2 Prompt engineering drawbacks

The main drawback of prompt engineering is the increased size in prompt text length which translates to increased usage cost.

LLMs don't understand text like humans do. Instead, the received text input is first converted into tokens. Tokens are common sequences of characters in the given text which hold statistical relationships between each other. The key takeaway is that this tokenization technique is the key reason why text based LLMs can do what they do – efficiently produce new words, or tokens technically speaking.

These tokens are split into two types: input and output tokens. Input tokens refer to the text input that is sent for the LLM to process, meanwhile output tokens consist of the generated text output.

A general rule of thumb is that one token corresponds to roughly 4 characters of English. 100 tokens equal to roughly 75 words. [7]

The more tokens that are used, the more computational resources are spent. LLM service providers offer pricing that is often billed on a per 1000 tokens basis with slightly different costs between input and output tokens. Output tokens are generally the more expensive type.

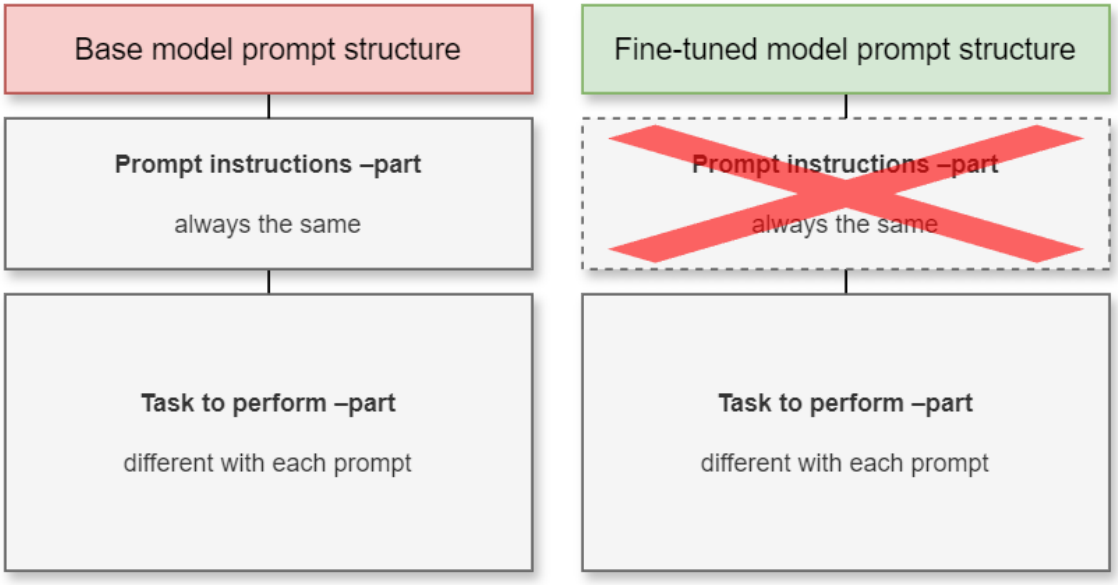
Due to the cost aspect, it is essential to strike a balance between prompt quality and context size. A prompt engineer must think how to best optimize and compress prompts but still retain good enough output quality. This process is done mostly by manual trial and error, especially if locked to using a third-party, closed source LLM, such as OpenAI's GPT.

In general, it can be observed that too little context usually means worse results, meanwhile adding more context yields better results, but also increased costs. However, increasing context does not provide endless improvement, in fact, too much context can reduce the quality and accuracy of the output.

Overcoming drawbacks with fine-tuning

It is guaranteed that prompt engineering techniques will result in larger prompt sizes, at least by a little. This is a relatively minor inconvenience for most, but nonetheless there are ways to mitigate or completely avoid this drawback.

One solution to this is a process called “fine-tuning” where the LLM is trained further with custom data which in this case would be the prompt instructions. The LLM will learn from the new data and if done successfully, this will eliminate the need of having to include prompt instructions within each request, since the LLM already has the required knowledge “baked-in” to the model itself. Therefore, the total token amount spent decreases, which is beneficial.



Picture 5. How using a fine-tuned model can save tokens.

OpenAI offers a fine-tuning service for most of their models. Fine-tuning can get a bit more expensive compared to just using the base models, but the pricing is still very reasonable [8].

Generally, fine-tuning is meant to be used for more broad goals. The mentioned token saving benefit is just one possibility, albeit rarely is it the primary goal. More commonly fine-tuning is used for controlling the general style, tone, format, and other qualitative aspects of the generated output.

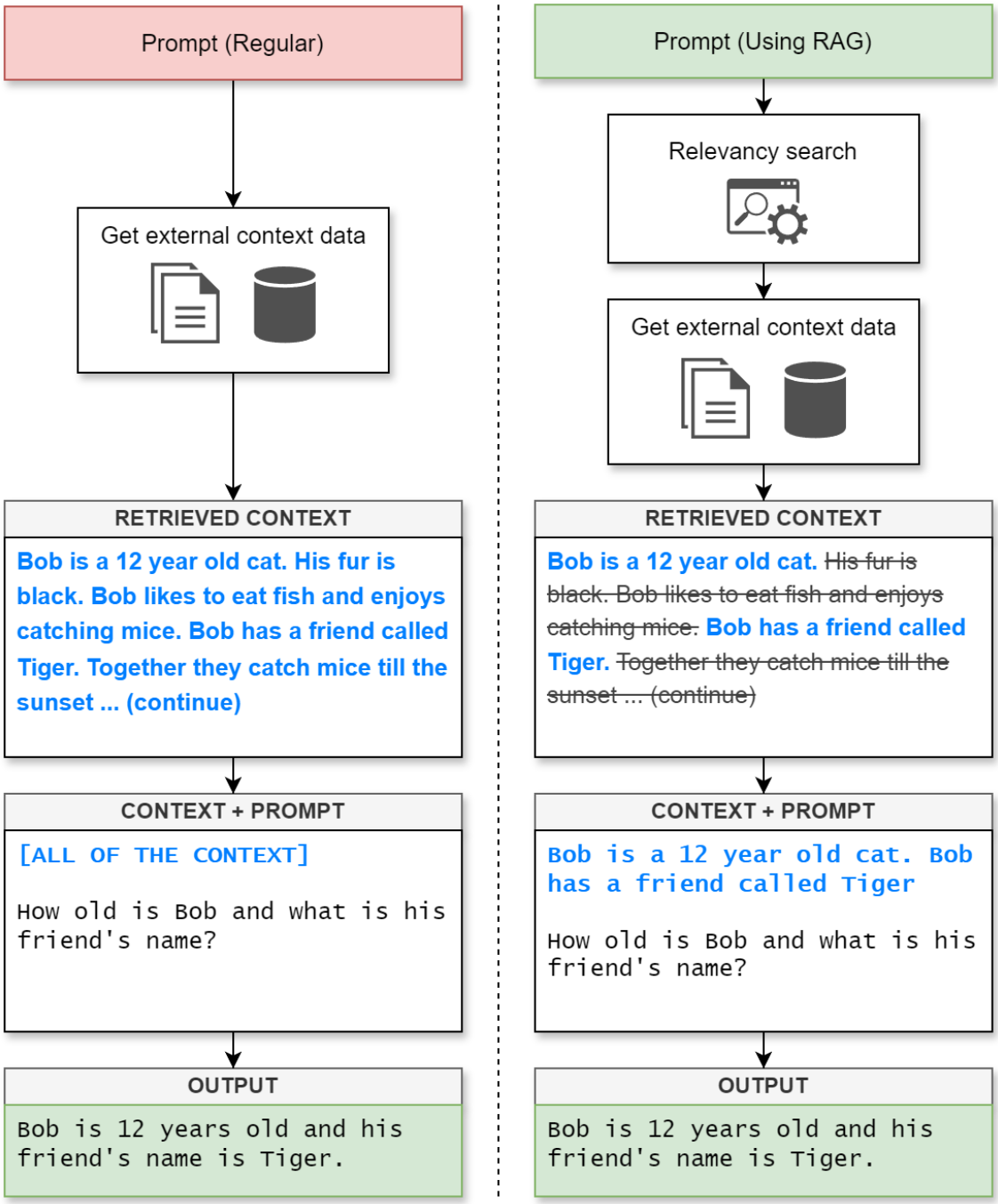
Regardless of what the particular use case for fine-tuning is, it should always be treated as a last resort, that is when everything else in terms of prompt optimization has been done. This is because a prompt might initially appear to perform poorly, but the results can drastically change just by slightly improving or changing the prompt structure. In other words, fine-tuning may not even be needed. It is also much faster to iterate with prompts than multiple fine-tuned model versions. A fast feedback loop is crucial when figuring out the behavior and possibilities of a given LLM. [9]

3.3 External large data source processing

Sometimes there might be a need to include large amounts of context data, more specifically from private data sources. Logically, the LLM does not have access to private data, so it must be manually included in the prompt for it to be useful.

However, LLMs have context size limits, meaning that it could be difficult to fit enough context. Although nowadays most models offer very generous limits. For example, OpenAI's GPT 4 currently supports up to 128 thousand tokens context window per prompt [10]. Despite this, the approach is rarely ideal, since spending up to 128 thousand tokens per prompt is very wasteful and expensive.

A better solution is a technique called Retrieval Augmented Generation (RAG). The concept is very simple – instead of retrieving "all" context for the prompt, only small pieces containing the most relevant information are included. [11] This method significantly saves on token usage since no useless context is included.



Picture 6. Differences between a regular prompt and a RAG prompt.

By default, the external data could be spread across many different data formats, like documents, images, videos and so on. This mixed state is still unusable. First, all the data must be converted into numerical representations and stored in a vector database. After that, the database can be used to perform relevancy

searches. This way the prompt content can be matched with the database contents resulting in the retrieval of only the most relevant context pieces.

RAG can be useful for all kinds of applications. For instance, it could be used to build a Q&A type chatbot handling the distribution of information within a company. Instead of browsing through numerous company documents to find something, you could simply ask the chatbot to retrieve it using normal natural language. While this method saves a significant amount of time, it is still important to recognize the risk of LLM hallucination, meaning that the retrieved data should still be verified in some way.

4 Project introduction and requirements

This chapter describes the main innovative features and lists the requirements set for the thesis project.

Compared to the other projects mentioned previously, this thesis project aims to innovate further with the introduction of so-called character events. The goal of these events is to bridge the gap between the LLM and the visualization, in the hopes of unlocking more elaborate AI storytelling. To do this, the LLM is given detailed context about the environment where the show takes place among other details.

A character event could be something like the following example:
Character 1 moves to the **table**, picks up a **book** and says: ***"I like reading!"***

Basically, in addition to generating dialogue for the characters, the AI could also control their physical actions via commands. This ability, if implemented properly, would result in much more varied and engaging content to watch. The added environment context could also aid in the AI's storytelling abilities and decision-making.

Due to time constraints, the project's scope had to be limited to a proof-of-concept level. This was somewhat expected considering the complexity of the project, but the development is expected to continue afterwards.

The set requirements for the project are as follows:

- Humanoid characters (show actors)
    - Character models, animations, etc.
- A few premade show sets (environments)
    - Props with placeholder models.
- Character events.
    - A small selection of character actions.
    - Character dialogue subtitles.
- Prompt builder.
    - Flexible tag system.
    - Few-shot examples generation.
    - Prompt compression.
    - Unique combinations randomization.
- LLM manual testing workflow (ChatGPT)
- LLM API implementation and workflow (OpenAI Chat Completions API)
- LLM Output parsing with basic error handling.
- Show playback (start, end, timing logic)
- Show debugging tools.
- Full solution, demonstrable from start to end.
- All project code should adhere to good coding practices and standards.

5 Project architecture

This chapter describes the tools and services used for building the project along with some reasoning behind the choices.

5.1 Unity

Unity is a powerful real-time game engine used most often for creating games, but it can also be used for all kinds of other 2D or 3D applications [12]. In general, Unity is a very flexible platform with relatively low hardware requirements, making it a viable option for many developers.

Unity was selected as the main platform for the project because it was known it would offer sufficient tools to successfully complete all the project requirements. More importantly, it was a natural choice due to having prior experience of working with Unity and its ecosystem. This meant that the development process could move faster, since almost no time would need to be spent learning the platform itself.

This is not to say Unity is the only game engine option, in fact there are many others to consider, most notably Godot or Unreal Engine. Technically speaking, any of these would have been just as viable. It is mostly a question of preference.

The project could be split into two main parts: visualization, and the prompting related logic. In theory, Unity was only necessary for the visualization aspect, however there was no reason to stop there since Unity's C# scripting language could be used to build the rest of it too. This way all aspects of the project were centralized under one ecosystem, making the development process seamless.

5.1.1 Visualization

As said, the goal was to get the LLM involved with the virtual environment as much as possible. This would be done by building a platform that could take in

commands generated by the LLM and then translate said commands into physical actions inside the virtual environment.

The LLM would not directly micro-manage every character with exact commands like *“Move 1 meter forward, Now turn left…”*, instead the LLM would give broad objectives such as *“Move this character to the table”.*

It would then be the game engine’s job to handle everything in between what is necessary to achieve the end goal. For example, in the case of moving a character to the table, it would first calculate the shortest path to said location, see if there are any obstacles along the path, and so on. In game development, this methodology is often called goal-oriented action planning.

A better example would be a more complicated task like: *“prepare lunch.”* While the end goal is clear, the steps in between require additional resolving. Ultimately, this task might end up looking something like the following sequence:

1. Move to the fridge.
2. Open the fridge.
3. Pick up ingredients from the fridge.
4. **Prepare lunch.**

Three additional subtasks had to be created, to complete the initial task.

Even though visualization is possibly the most time-consuming aspect of this project – it is not the primary focus of this thesis. Instead, the focus is mainly on the generative AI aspect. Going forward, many of the small details and non-specialized features regarding the game engine are left out from the text.

To keep things simple, the characters, also known as show actors, would initially be limited to humanoid characters. Other types would be something to consider later. The characters would also be able to speak, however a text-to-speech system was not intended to be implemented at this point. Instead, a simple placeholder subtitle system was planned for displaying the character speech.

The environments (show sets) would be premade but still contained many randomized elements. For instance, all the props that the characters can interact with would be randomized. In addition, the placement of many “point of interest”-objects would be slightly altered for each show.

5.1.2 Prompt builder

This refers to the system that creates the actual text prompt structure which is sent to the LLM for processing. The goal of this system is to automate the process of writing prompts in a consistent manner. Of course, the prompts could also be written manually, but it would be slow and prone to mistakes in the syntax.

The prompt builder is written in C#, making heavy use of its string manipulation methods. In short, the prompt builder gathers information from different sources and then formats it all into an optimized prompt package. The information comes from the show set and user-defined keywords, such as the topic, style, and other qualitative features.

5.2 LLM provider

OpenAI’s services were selected for the LLM aspect. This was primarily because they offered free options that could be used during development. In addition, OpenAI already has robust API infrastructure in place, allowing for fast integration. However, this choice is not in any way final. In fact, the project was designed to be easily interchangeable with other LLMs if needed.

ChatGPT and Chat Completions API

ChatGPT is an LLM chatbot developed by OpenAI. It is currently equipped with OpenAI’s proprietary models GPT 3.5 and GPT 4. The latter is currently only available for paying customers, whereas the 3.5 version is available for free. The

free version was convenient, considering the project would require a significant amount of trial-and-error type prompt testing during development.

OpenAI Chat Completions API is a service which allows users to integrate GPT models in their own projects programmatically. In simple terms, this is “ChatGPT” for third-party applications. Compared to ChatGPT, the API offers more models and advanced configuration options that can be used to control the LLM’s behavior. [13]

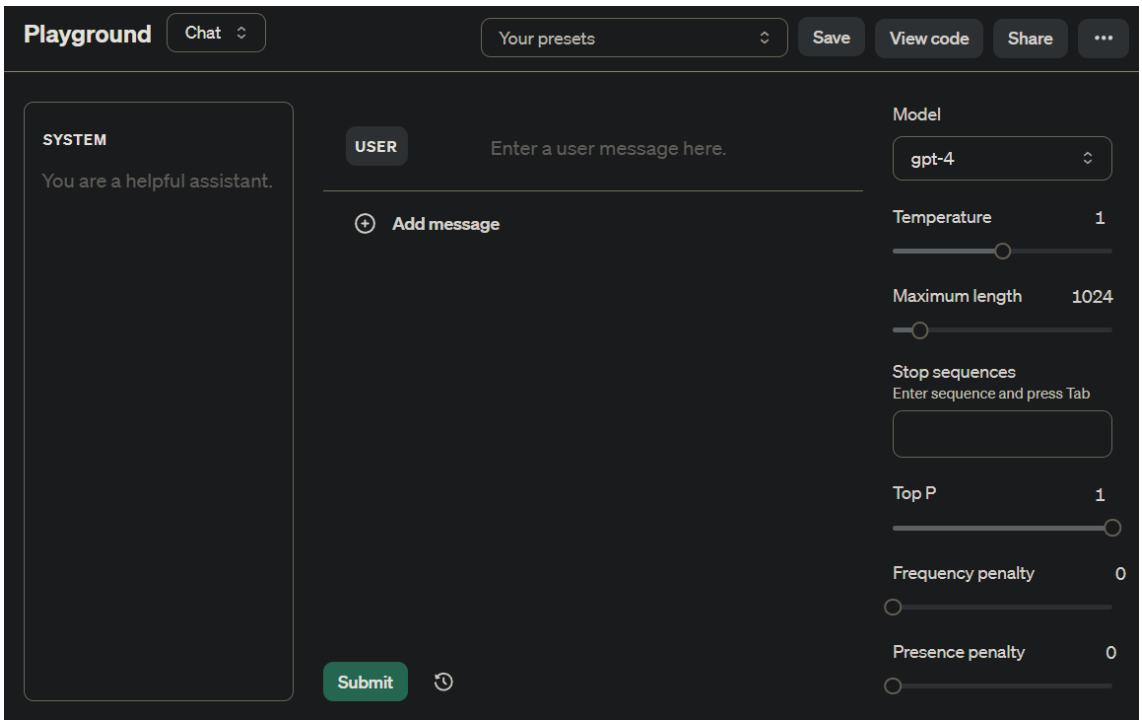
The configuration options include the following:

- **Temperature**: A value that controls the randomness of the output. The higher the value, the more random the output becomes, while lower values produce more deterministic and repetitive results.
- **Maximum length**: A value used for limiting the maximum number of tokens that can be spent for output generation.
- **Stop sequences**: User-defined words or sequences that indicate when the generation should stop. For example, using a “dot” as a stop sequence will limit the output to a single sentence.
- **Top P**: Whenever the LLM generates new tokens (words), it selects the next tokens from a pool of probable options. This value controls the threshold of what tokens are considered. A value of 0.1 would mean only the top 10% of the options are considered.
- **Frequency penalty**: A value that penalizes the repetition of tokens in the generated text. The higher the value, the less likely the LLM is to repeat itself.
- **Presence penalty**: A value that penalizes tokens based on whether they have already appeared in the generated text. The higher the value, the less likely the LLM is to stay on the same topic and instead move on to new topics.

5.3 Other tools

5.3.1 OpenAI Playground

OpenAI Playground is a platform for experimenting with various OpenAI APIs, such as the Chat Completions API through a user-friendly interface. This is not to be confused with ChatGPT, as it is not the same. The Playground requires an active OpenAI API key to be used, meaning that usage is billed, just as if using the API normally via HTTP requests.



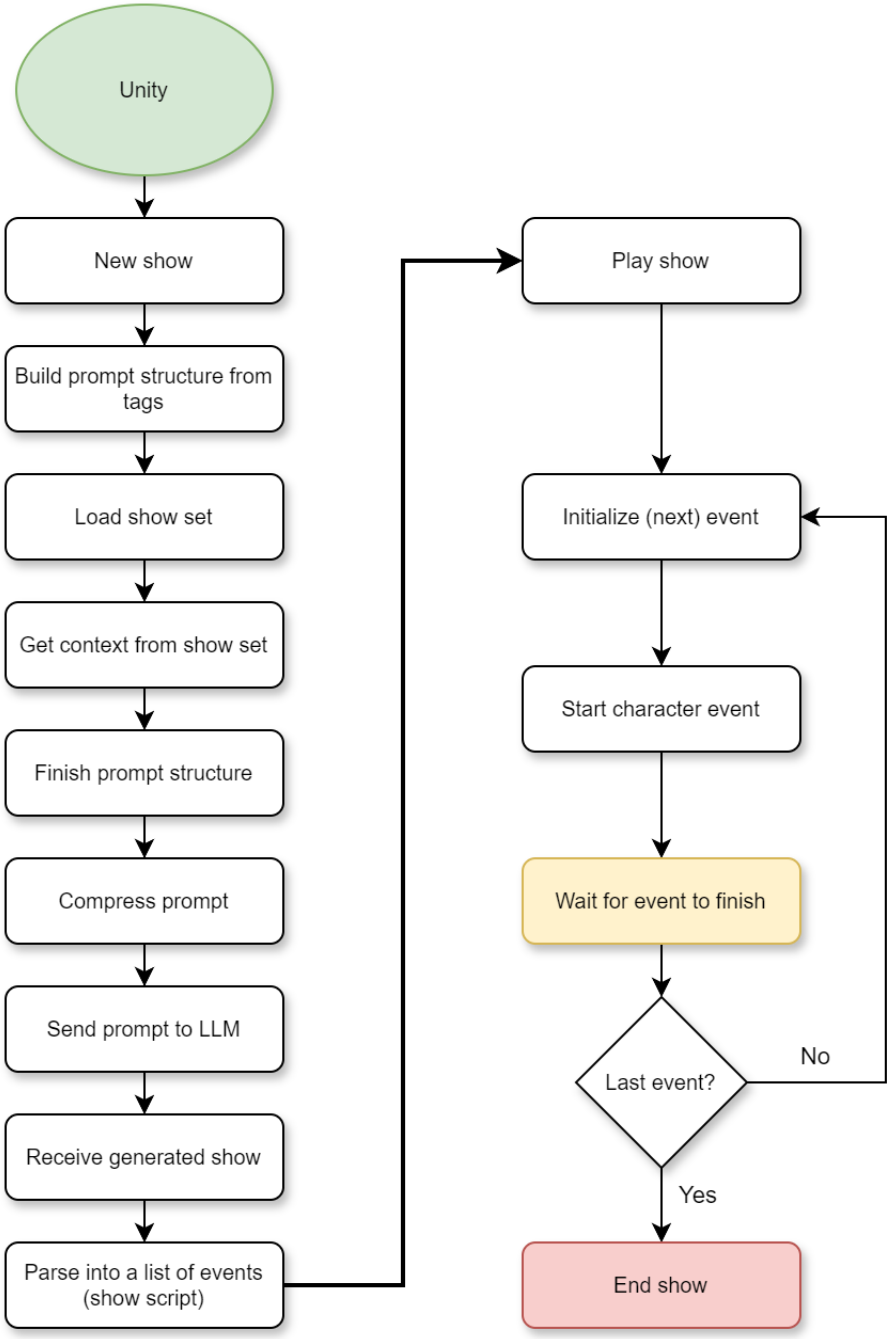
Picture 7. Screenshot of OpenAI Playground.
The configuration options are visible on the right.

5.3.2 OpenAI Tokenizer

OpenAI Tokenizer is a tool for calculating the total token count for a given piece of text. It helps with understanding how the tokenization process works with different models. [14]

6 Project implementation

The entire project was built within a single Unity project. Initially the aim was to get the overall idea working at its most basic level. After that, the individual systems would be improved gradually over time.



Picture 8. Diagram of the main project flow.
Starting with show generation and ending with show playback.

6.1 Building the characters and environments

The humanoid 3D models and animations were provided by Mixamo, a service owned by Adobe. Mixamo offers a free library of ready to use 3D character models, animations, and automatic character rigging tools. [15]

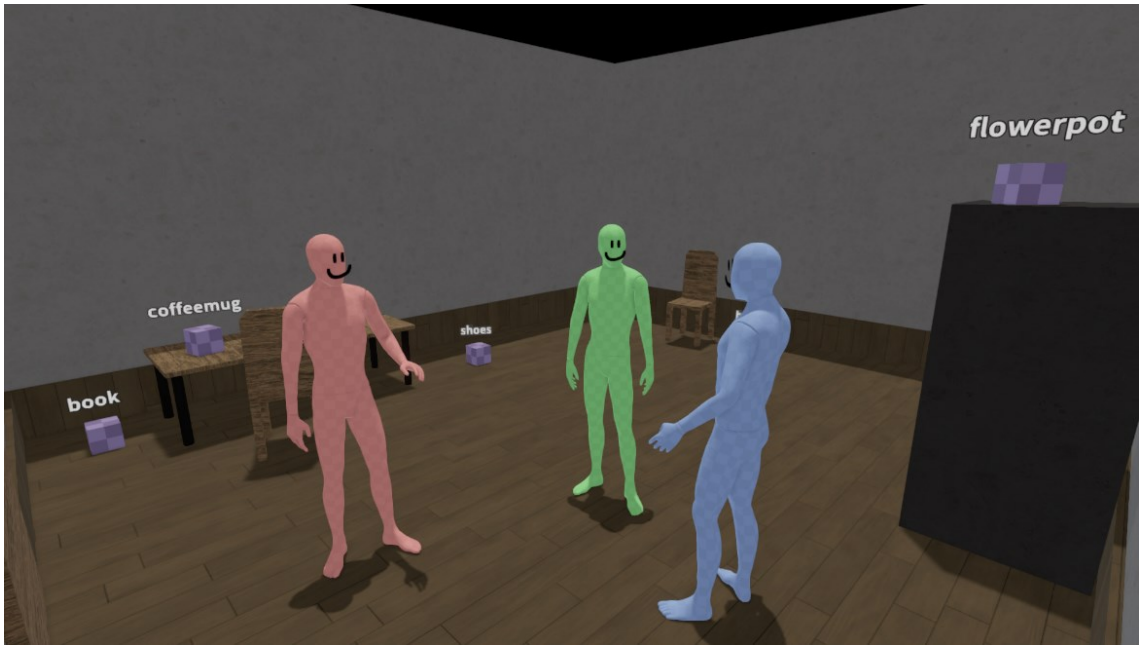
At this point, the characters would all look the same, except for having unique colors to help differentiate between them. Their behavior, animations and 3D models would be the same.

As an added flavor, the characters would also utilize ragdoll-physics and physics-based animations, meaning that their interactions would behave “realistically”. For instance, if a character accidentally collided with an object in the scene, it could fall over in a realistic manner. This was very much an out-of-scope feature, but nevertheless, it ended up adding a whole another layer of unpredictability to the simulation, resulting in more engaging show content.

For the environments, the idea was to build multiple different show sets that could be used. These sets would consist of several points of interest, called “Spots”. For now, creating just a single set was enough. This set was meant to convey an apartment room where the spots represented various furniture items such as tables, beds, chairs, and more. The set layout was made rather quickly, using primitive 3D models for everything.

The props would also be simple temporary cubes, accompanied by a floating text describing what the prop is meant to be.

Overall, most of the visual rendering was implemented with lots of placeholders in place but all the functionality was there. The visuals could be improved later.



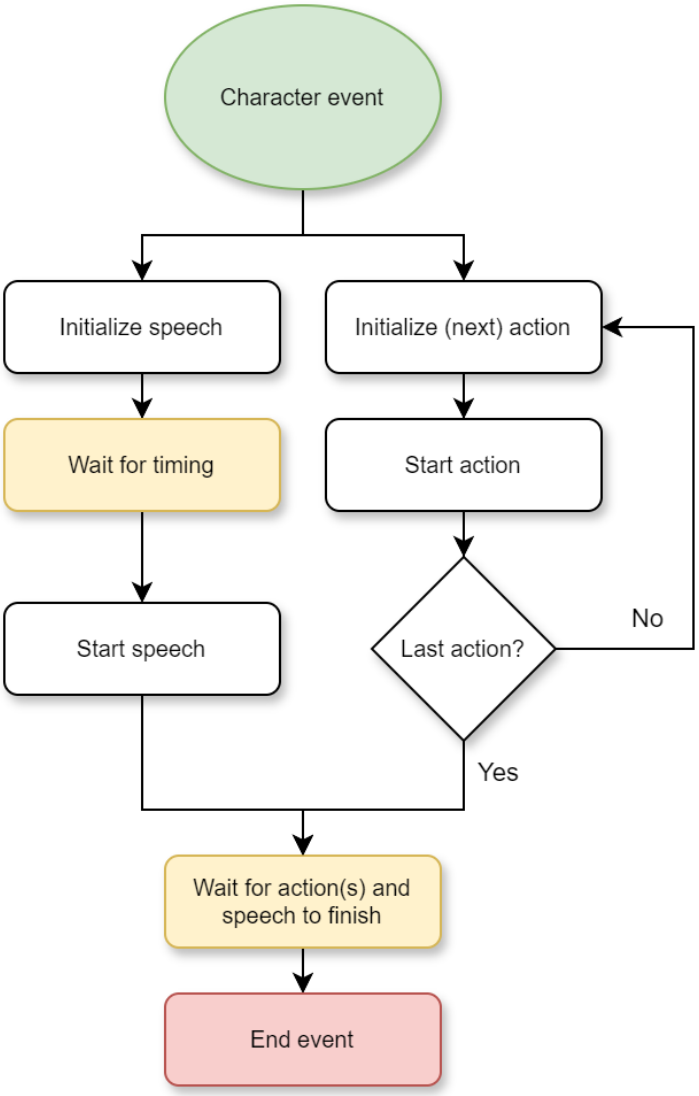
Picture 9. Screenshot of the first environment: "Apartment".

6.2 Character actions

Initially three actions were implemented: Moving, picking up props and talking. This list of actions would grow over time.

After some testing, it was determined that the “talk” action should be treated as its own special action, which could be included alongside other actions. This way the characters could speak while also simultaneously performing other actions.

In the end a character event would consist of both an action and a speech component. As a temporary solution, a simple subtitles system was implemented for displaying the character speech, which would be replaced or accompanied by text-to-speech in the future.



Picture 10. Diagram of a single character event.

The actions and the speech are separate tasks that run simultaneously. Both need to be finished for the event to be considered as complete. Only then the show will move onto the next event. Due to the unpredictable nature of the show simulation a simple timeout logic was also implemented, meaning that if a character got stuck in some action for too long, then it would be forcibly ended to ensure the show does not just freeze forever.

6.3 Building the prompt builder

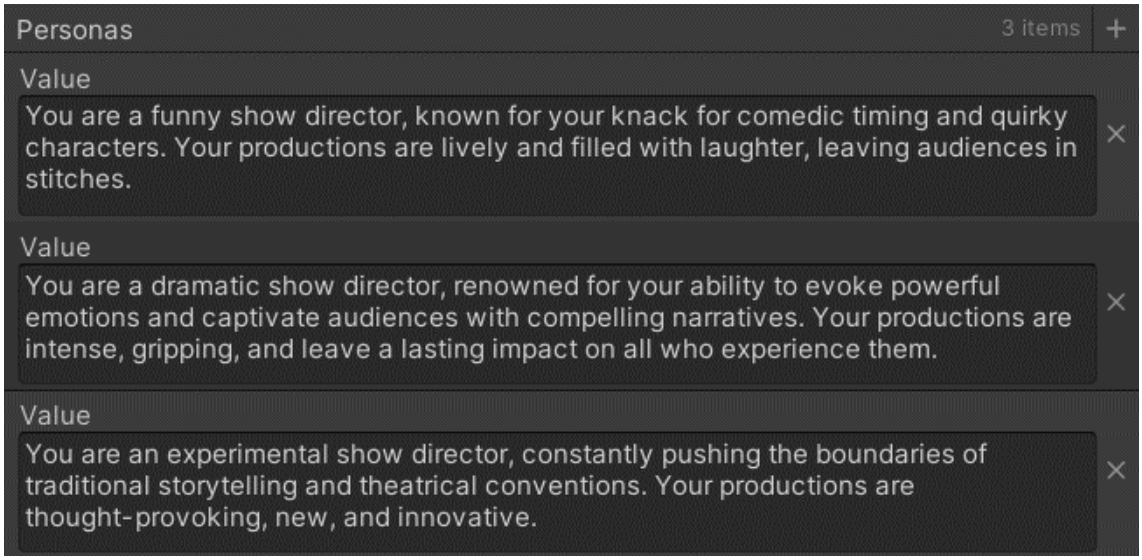
The prompt generation begins with an empty string template containing the overall tag structure. The tags in this template are going to be replaced with corresponding data.

Picture 11. The empty prompt tag structure.

The tag names are quite self-explanatory on what data they would be replaced with.

The structure can be modified freely – new tags can be added, and existing ones can be removed. For example, if the topic tag were removed, it would result in the LLM having to come up with a topic on its own instead. It is a flexible system, that enables quick experimentation.

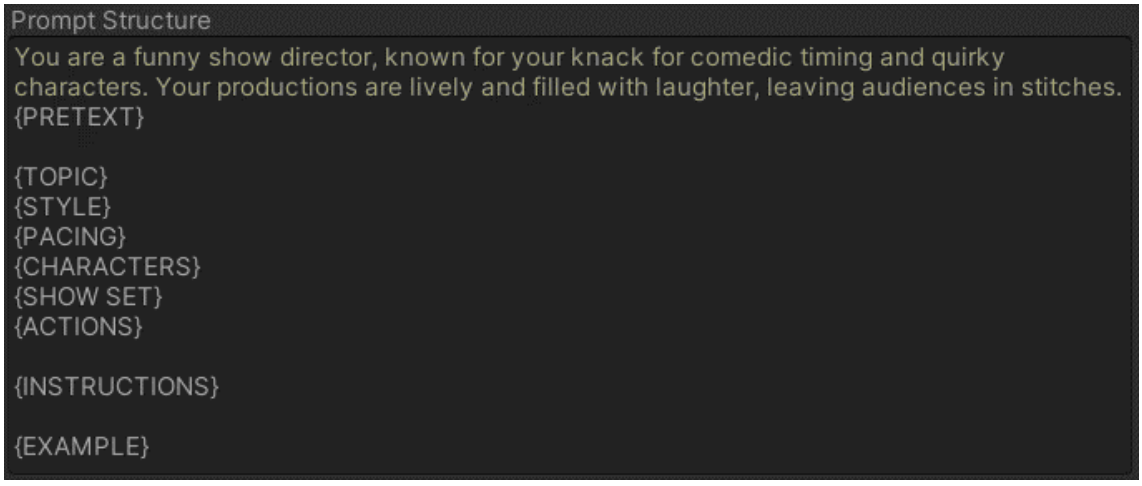
The data for these tags is retrieved from another template, containing arrays of user-defined text strings for each tag. Only one option would be picked for each tag. To better understand this, let's look at how the {PERSONA} tag's data is retrieved.



Picture 12. Options available for the {PERSONA} tag.

From the available options, only one would be picked at random. The reason for having multiple options is because it allows for unique combinations to naturally occur, adding more variety to the finished prompt. The number of options is not limited, but at least one is always required.

Let's assume the first persona (funny show director) was picked, now after this step the prompt structure will look like the following:



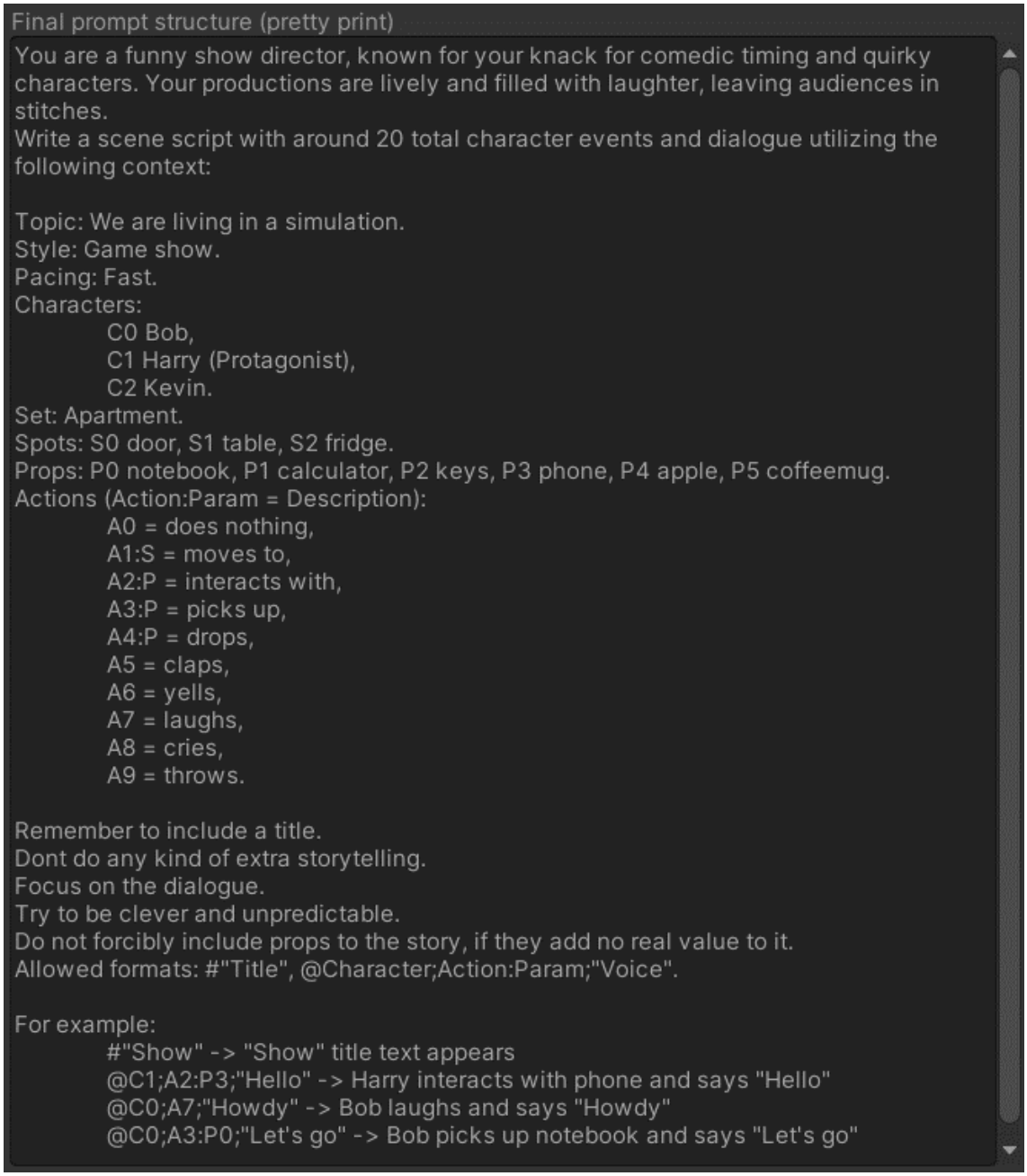
Picture 13. The updated prompt structure.
To clarify, the persona is a short text detailing the desired behavior and role which the LLM should adopt.

This same process would be repeated for the rest of the tags, with only minor differences between them. However, the {EXAMPLE} tag stands out as an exception.

As stated in the prompt engineering theory chapter, examples play a crucial role in controlling the LLM output. This is precisely what the {EXAMPLE} tag is for, to contain the examples, following the few-shot prompting technique.

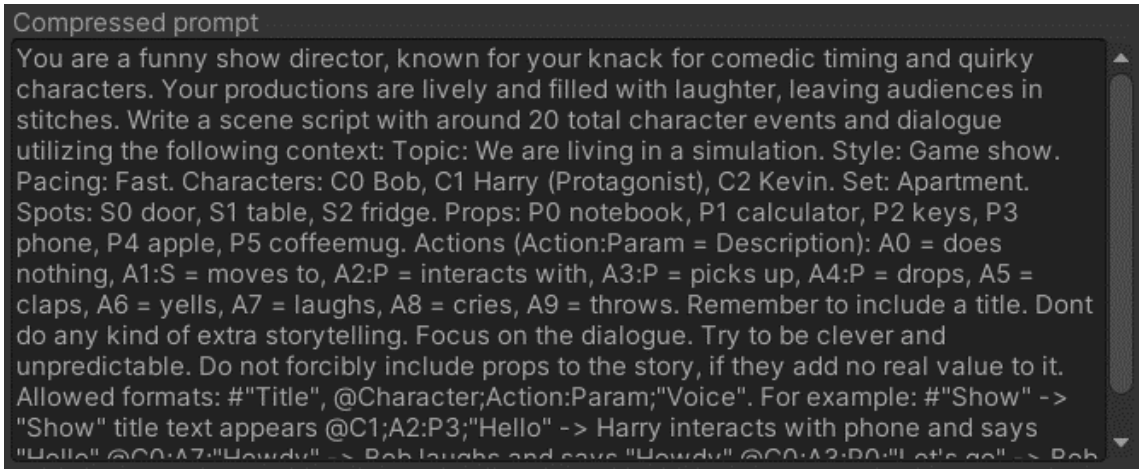
However, because the examples are based on all the other context, it was not possible to predefine them. Instead, the examples had to be dynamically generated at runtime.

What this also means is that the examples were not just theoretical examples, rather they would be constructed to be fully functional and valid. Even a natural language description for each example was generated. The objective behind all this “extra work” was to eliminate any possibility for the LLM to misinterpret the rules.



Picture 14. The final prompt structure with all the tags replaced.
The few-shot examples can be seen at the very end.

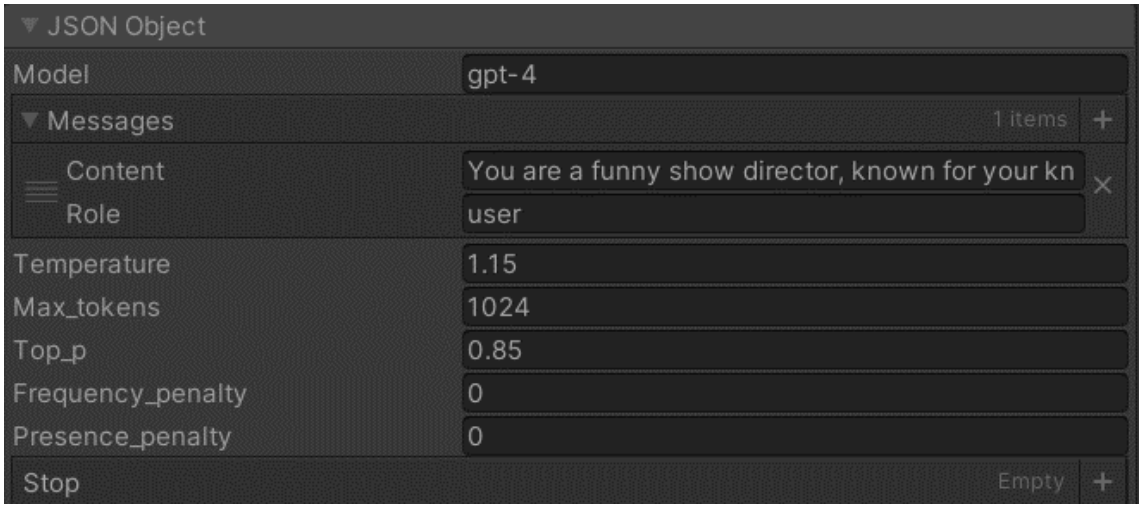
After the prompt structure is complete, it is compressed down by removing all the indentation and extra spaces. These features are useless information for the LLM, so they can be removed to save on tokens. This is the version that is sent to the LLM. The nicely formatted version is mainly for debugging purposes, as it is much easier to read.



Picture 15. The final prompt structure with compression applied.

6.4 OpenAI Chat completions API implementation

All API connectivity was written in C#, making use of Unity's built-in networking classes. The Chat completions API endpoints required the POST data to be in JSON string format. Unity does have some basic JSON utility classes built-in, which were helpful for this task. However, the classes are quite limited in features, suggesting that switching to a third-party JSON library could be wise in the future.



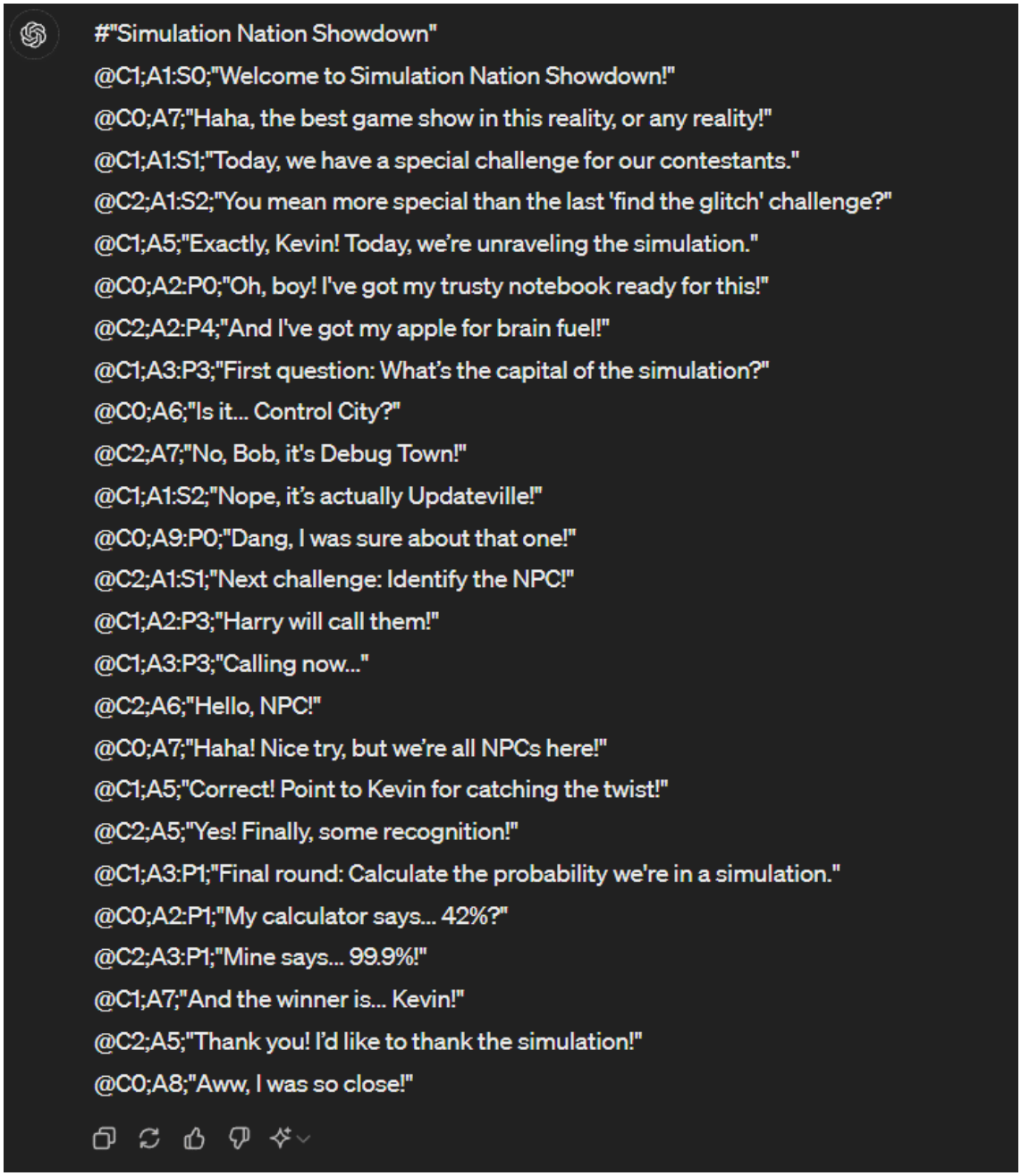
Picture 16. The Chat Completions API POST data structure.
This is encoded into a JSON string.

Even though the API connectivity was implemented, it did not find much use or testing at this point. Using the API is not free, so instead most of the prompt

testing was still being done manually using the free alternative, ChatGPT. Switching to the API is mainly intended for production use when the project is fully operational.

6.5 Parsing LLM API output

After sending the prompt to the LLM, a new show script would be generated and returned in the same format as demonstrated in the few-shot examples. This is the raw text output that is parsed into character events, which could then be simulated in the virtual environment. The following picture demonstrates what the output looks like when using ChatGPT.

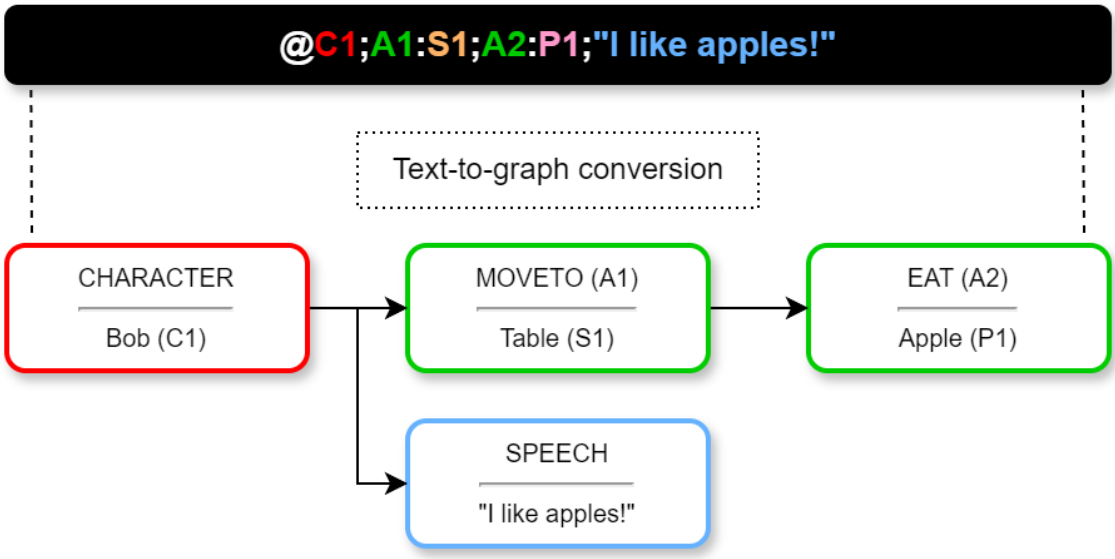


Picture 17. The generated show script, using ChatGPT.
This output is based on the prompt seen in Picture 14.

The output starts with the show title, marked by the “#” character. This text would be shown as the initial title when the show playback begins.

The rest of the output consists of all the character events in their raw line text format. These lines are made of multiple tags, separated by semicolons. The first tag represents the character, and the last tag contains the character’s speech

content. Everything in between contains the actions which the character would perform in sequential order from left to right. The “@” character at the start of each line helps with splitting the output into individual lines, even though the LLM usually handled this on its own by adding newline characters. However, this was not always consistent, so adding a special character was necessary.



Picture 18. Text conversion process for a single line.
This example demonstrates two actions (green), although typically there would only be one.

The format uses a two-digit ID system to further optimize token usage. The characters are represented as the letter “C” followed by a unique number ID, starting from zero, i.e. “C0”, “C1”, and so on. Other tags would follow the same naming convention, except for the initial letter. These IDs are then mapped to reference their corresponding subjects. Originally the format used clear text, but this was replaced with the two-digit ID system.

Picture 19. A clear text version of a single line.
Not used because of higher token usage.

Table 1. All the implemented tag ID types.

ID	Type	Example of mapping
C	Character	C1 → “Bob”
A	Action	A1 → “MOVETO”
P	Prop	P1 → “Book”
S	Spot	S1 → “Front door”

Table 2. A few of the implemented actions and their parameter types.

ID	Type	Parameter	Example of use
A1	MOVETO	S (Spot)	A1:S1
A2	EAT	P (Prop)	A2:P1
A3	PICKUP	P (Prop)	A3:P2
A4	CLAP	-	A4
A5	HUG	C (Character)	A5:C1

The actions are limited to accepting only a single type of parameter but multiparameter functionality was planned to be implemented later. This improvement is necessary because some actions should be able to use more parameters, like the “MOVETO”-action – there is no reason to limit character movement to “Spots” only.

It is also worth noting that none of the IDs are hardcoded, as it may seem so far. The IDs were dynamically defined at runtime. For instance, A1 might not always refer to the “MOVETO”-action and so on.

The two-digit system proved to work well, but even a single-digit ID system was experimented with. In this system only the numeric ID remained, but it was quickly discovered that this resulted in frequent invalid output. As an example, the LLM could attempt generating actions like: *“Character 1 eats a table”* which doesn't make much sense, nor is it a valid action.

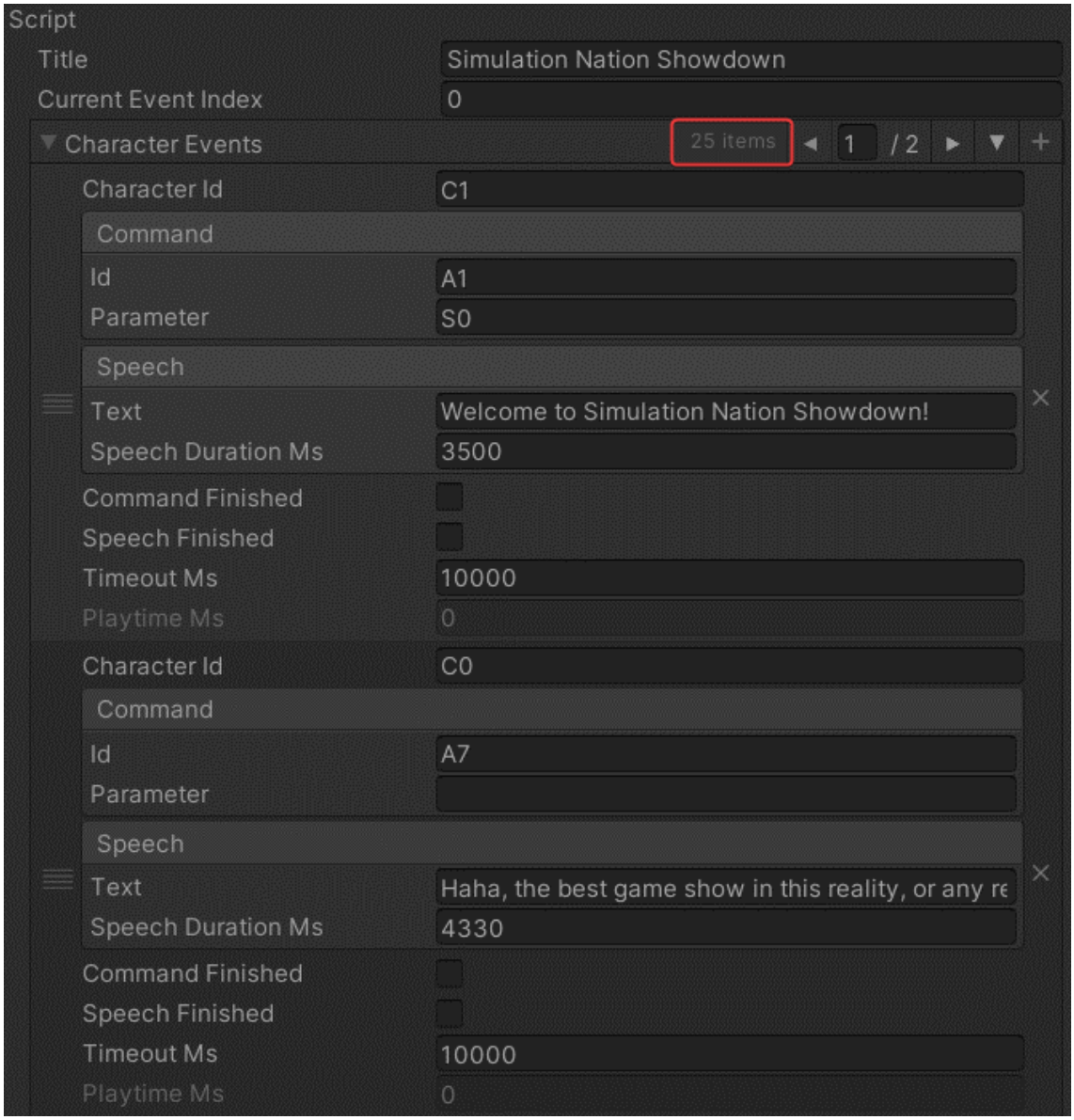
These new problems seemed a bit odd, because the underlying logic of the format did not change. The assumption is that the LLM got confused since all the ID types shared the same integer value ranges. For example, the ID “1” could simultaneously refer to a character, action, prop, and spot all at once. But ultimately it is the order of use that matters, so theoretically, it should still have worked.

In the end, the two-digit ID system was brought back, even though the occasional mishaps from the one-digit system could hold some comedic value. The goal was to generate consistent and reliable results after all. Moreover, it is likely that the single-digit system would work if the ID integer ranges were refactored to be always unique, but this level of optimization was not a high priority during this time.

Even the two-digit system would sometimes produce broken, or partially broken output. As a solution, basic error handling was implemented, which simply skipped any invalid lines during the parsing process.

After parsing all the generated output into individual character events, they would be added to a list known as the show script.

Before adding them, some preprocessing would be performed, primarily involving calculating the speech duration estimate. As implied, this would be an estimate based on a made-up formula that includes variables such as the total character and word count of the speech. After some math operations and rounding the result is a rough estimate on how long the speech would take in milliseconds.



Picture 20. The parsed show script list.
Only the first two events are visible.

6.6 Simulating the show script

Before the show playback can start, the show environment must be initialized. This means spawning in all the characters, applying visual settings, calculating the pathfinding areas and so on. After that, the show can begin with displaying the title, followed by all the character events simulated one by one. During the show, the camera automatically changes position and rotation, ensuring the currently performing character stays in frame. The positioning is done by

calculating a random point inside a dome shape around the character's head. Currently it is a rather simple system, but more features, such as static camera angles, are planned to be implemented later.

There were also plans to implement a feature that could automatically delay the speech part if needed. In certain scenarios it could be useful to delay character speech. A good example of this is an action where a character opens a door and greets the person behind it. It would be ideal if the speech started only after the door has been opened, however with the current system, the speech would always start instantly.

Due to the two-digit ID system, it was not always easy to tell if the script generation was behaving as intended. To solve this, a set of debug tools was implemented to help understand what is currently happening with the show playback. The debug tools show the current state of the show, what the current character event translates to, and how much time has passed in total. All this helped diagnose issues that could otherwise go easily unnoticed.

Picture 21. Screenshot of the debug tools UI.

Picture 22. Screenshots of the show in progress.

7 Results

This chapter outlines the results of the implemented project. This was done by checking the implementation status of each requirement, that was set in the project introduction and requirements chapter.

Table 3. Requirements listed with their current implementation status.

Requirement	**Status**
Humanoid characters (show actors) - Character models, animations, etc.	Implemented
A few premade show sets (environments) - Props with placeholder models.	Partially implemented[1]
Character events. - A small selection of character actions. - Character dialogue subtitles.	Implemented
Prompt builder. - Flexible tag system. - Few-shot examples generation. - Prompt compression. - Unique combinations randomization.	Implemented
LLM manual testing workflow (ChatGPT)	Implemented
LLM API implementation and workflow (OpenAI Chat Completions API)	Implemented
LLM Output parsing with basic error handling.	Implemented

(continue)

Table 3 (continue).

Show playback (start, end, timing logic)	Implemented
Show debugging tools.	Implemented
Full solution, demonstrable from start to end.	Implemented
All project code should adhere to good coding practices and standards.	Implemented
1. Only one set was made, and the overall design work is very basic and incomplete. In addition, the props are still using placeholder models.	

The results of the project align well with the set requirements. All the requirements were met, at least on a basic level. The underlying functionality is there but the visual representation of nearly everything requires a lot more work.

8 Conclusion

This thesis aimed to experiment with generative AI by utilizing prompt engineering techniques. The main idea was to transform text based LLM storytelling into a consistent visual format.

Overall, the project implementation was successful. All the set requirements were implemented and even some extra. Most importantly, the project adhered to the proper development practices, resulting in a solid foundation for any future development. Many of the systems and ideas could also be repurposed in other projects if needed.

A significant amount of development time was spent doing trial-and-error prompt testing. The prompt building process was very much about “seeing what works” and iterating from there. The prompt builder component played a key role in this task because it automated much of the slow and tedious work of text formatting. It became clear that the ability to iterate quickly is important, which is exactly what the prompt builder allowed. This system went through many versions before finally landing on the current implementation.

Several challenges still need to be addressed, one of which is the issue of script and simulation mismatches. This happens when the show simulation fails to complete a given action, and then the next event would act on the assumption that the previous action was successful, when it was not. This can somewhat break the immersion from that point on, depending on how coherent the show script is. Luckily the shows are meant purely for entertainment, so this isn’t too big of a problem, in fact small mistakes can add to the fun factor.

In conclusion, working on this thesis project provided a lot of insight into prompt engineering, contributing to a deeper understanding of LLM behavior overall. Knowing how to utilize advanced prompting techniques appears to be a valuable skill, at least for the time being.

The current project will continue development and is expected to eventually have a public release of some kind. Originally the idea was to host a livestream of endless AI-generated shows, like the previous projects have done. However, over time, this plan has become less certain as new ideas have emerged during development. Generative content appears to hold a lot of potential, especially in video games, which might be a topic worth exploring.

Some future ideas and plans include:

- Testing other LLM providers.
- Using LLMs to write better tag keywords/content.
- Text-to-speech for character dialogue
- Experiment with fine-tuning.
- Multi-parameter actions
- Extended shows (continue from previous show)
- Real-time mode (the show is generated continuously)
- Other character types

References

[1] NVIDIA, "Large Language Models Explained," [Online]. Available: https://www.nvidia.com/en-us/glossary/large-language-models/. [Accessed 19. May 2024].

[2] Mismatch Media, "WatchMeForever livestream," [Online]. Available: https://www.twitch.tv/watchmeforever. [Accessed 21. May 2024].

[3] K. Gülen, "AI SpongeBob: Absurdly hilarious or hilariously absurd?," 12. June 2023. [Online]. Available: https://dataconomy.com/2023/06/12/what-is-ai-spongebob/. [Accessed 2. April 2024].

[4] DAIR.AI, "Prompt Engineering Guide," [Online]. Available: https://www.promptingguide.ai/. [Accessed 21. May 2024].

[5] DAIR.AI, "Prompting Techniques," [Online]. Available: https://www.promptingguide.ai/techniques. [Accessed 21. May 2024].

[6] DAIR.AI, "Zero-Shot Prompting," [Online]. Available: https://www.promptingguide.ai/techniques/zeroshot. [Accessed 21. May 2024].

[7] OpenAI, "What are tokens and how to count them?," [Online]. Available: https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them. [Accessed 18. May 2024].

[8] OpenAI, "API Pricing," [Online]. Available: https://openai.com/api/pricing/. [Accessed 22. May 2024].

[9] OpenAI, "When to use fine-tuning," [Online]. Available: https://platform.openai.com/docs/guides/fine-tuning/when-to-use-fine-tuning. [Accessed 21. May 2024].

[10] OpenAI, "OpenAI GPT Models Overview," [Online]. Available: https://platform.openai.com/docs/models/overview. [Accessed 21. May 2024].

[11] Amazon, "How does Retrieval-Augmented Generation work?," [Online]. Available: https://aws.amazon.com/what-is/retrieval-augmented-generation/. [Accessed 21. May 2024].

[12] Unity, "Unity Engine," [Online]. Available: https://unity.com/products/unity-engine. [Accessed 15. May 2024].

[13] OpenAI, "OpenAI Chat Completions API Docs," [Online]. Available: https://platform.openai.com/docs/api-reference/chat/create. [Accessed 19. May 2024].

[14] OpenAI, "OpenAI Tokenizer," [Online]. Available: https://platform.openai.com/tokenizer. [Accessed 21. May 2024].

[15] Adobe, "Mixamo," [Online]. Available: https://www.mixamo.com/. [Accessed 21. May 2024].