Introduction

This project had the goal of automating the generation of a visual novel with AI, from a list of questions with 4 possible answers. It is built upon Xiaoyu Han's previous project with the same goal. However, he used Chainforge to query LLMs, and Unity and Fungus as a visual novel engine. His project used ChatGPT, but there is now a goal of having as much as possible running locally.

The project wasn't fully automated either. One part of the Chainforge file generated the stories and the other part was being fed the stories and generated the transitions, but the stories had to be copy-pasted into the 2nd prompt.

Workflow

I wasn't feeling like I would be fast enough using Chainforge for this project, especially to parse the outputs. This is why I decided to write a script in as-code.

I quickly saw that Fungus was not suited to generate visual novels dynamically. I decided to use Ren'Py instead, because the game is made of plain text files with a consistent format.

I used Python to write the script because, since it's used a lot in this kind of situations, I thought there would be more resources online and more modules I could use to send prompts more easily to the LLMs I used.

Text-to-text generation

I got started by using the prompts Xiaoyu Han already wrote. After some experimentation, I found that Gemma2 gave the best results in terms of «parsability», while Llama3 was better at writing chronological transitions and at text analysis. I managed to tweak the top_p[1] and the temperatures[2] in order to have a consistent format, and Xuanhui Xu helped me detect characters speaking in the story with a decent enough reliability.

To sum up, here are the text-to-text prompts I had, the models and the parameters I used, and why:

Prompt	Model	Parameters	Why
Generating the story and detecting characters	Gemma2	Top P = 0.75	Gemma2 is good at respecting the format I want, even with a high top P. This makes it a good fit for generating slightly-different stories every time that are easy to parse.
Generating transitions	LLama3	Top P = 0.8	Llama3 is good at understanding that the 1st story happens before the 2nd one. It is also decent at making natural-sounding transitions, even if they are very basic.

[1] One parameter to fine-tune the way the LLM behaves. A lower top P makes more deterministic results. The LLM will pick tokens from the smallest subset which sum of probabilities is higher than the set top P. Then, it normalises the probabilities.

[2] Another parameter to fine-tune the way the LLM behaves. Like the top P, a lower temperature means a more deterministic but reliable output, while a higher temperature makes the results more creative.

			However, it generally introduces it with a sentence like «Sure, here is the transition:». Having a lower Top P even makes it explain why it wrote it like it did.
Making a list of keywords describing the background image	Llama3	Top P = 0.1 Temperature = 0.1	I needed a very deterministic result that didn't look like natural sentences, hence the parameters. Moreover, Llama3 is good at analyzing text reliably enough.

Figure 1 - Summary of text-to-text prompts, models and parameters used

The last prompt was needed in order to generate a background image for each story.

Text-to-image

I had enough time to try to generate background images. I used Stable Diffusion 3 Medium, because it is a decent enough model that can run on the hardware I had.

In order to optimize my script, I stored the text-to-image prompts in a list, then generated them all in bulk.

I decided to generate images in a 1024x176 resolution. It was the best resolution I came up with that had acceptable results and didn't crash on the hardware I had access to. It also has an aspect ratio of 16:9, and it is divisible by 8 (which is a requirement of Stable Diffusion).

With some experimentation, I set the number of inference steps[3] to 50 (just above the default), and a guidance scale[4] of 7.

I generated anime-style backgrounds because it was one style that didn't have a lot of aberrations (and the ones I saw didn't look too bad), especially compared to a more realistic output.

I tried upscaling the image to ensure it wouldn't look pixelated or blurry in the visual novel, but the hardware I used wasn't powerful enough.

Putting everything together

In order to generate the visual novel, I pre-created an empty one and zipped it into my project. It had several pre-written elements, such as a few settings and a `transform image_upscale` block.

The script goes back over the parsed story (contained in a list of `Situation` objects). This class has a `parse()` method that uses a dynamically-generated regular expression in order to match the lines of dialogue with the actual lines in the story, even if it is cut by other parts of the sentence. Indeed, the story generated sometimes has dialogue tags in the middle of the sentence. They are not written in what the LLM says are lines of dialogue.

Said lines of dialogues are stored in `Dialogue` objects. They have a `line` attribute and a `speaker` attribute. The latter can be `None` of a string of the name of the speaker.

The character names are then stored in a characters list, an attribute of Situation. This is because I need to have a list of all the characters in order to register them in Ren'Py.

[3] Controls the number of times the generated image will be refined. Increasing the number of inference steps adds more details, but increases the generation time and can create abnormal details.

[4] Guidance scale defines how closely the LLM follows the prompt. The value must be between 0 and 20.

The script goes over every part of each situation, and writes each line of dialogue in the visual novel. It also implements the questions and the transitions to the next parts.

Once the files are written correctly, I compile and «distribute» the visual novel, i.e. I make an executable out of it.

Caveats

Since this was a prototype, I didn't take the time to have a clean code architecture. In other words, almost everything is in a `if __name__ == "__main__"` block.

Everything that uses a text-to-text LLM is in a big `try`/`except` block. This means that if there is one error, it starts over from scratch.

While it is reliable enough for a proof of concept, the character detection is not completely reliable. I think it is due to the way I dynamically create the regular expressions, but I didn't have enough time to look into it.

I could probably have better-sounding and better-looking outputs if I could use gemma2:27b and llama3:80b for text generation, and Stable Diffusion XL and an upscaler for the images. The hardware I had access to wasn't powerful enough to do so, however. The GPU I had was made for gaming, but I would have benefited from using one designed for AI.