PrepPipe: Prototyping Compiler for Attainable Visual Novel Development

Shengjie Xu
University of Toronto
Toronto, Canada
shengjie.xu@mail.utoronto.ca

***Abstract*—Visual novels are a low-cost storytelling genre in gaming. They can be developed by small teams or individuals within weeks or months. Despite their simplicity relative to other game categories, challenges persist that trap amateurs and lead to inefficiencies or even project failures. PrepPipe project seeks to accelerate visual novel prototyping, shorten development time, and reduce effort wasted due to planning errors. Our project includes a prototyping compiler that transforms story scripts and assets into game project files, supplemented by auxiliary tools and asset templates. Key features of the compiler include rich-text input handling, support for a user-guided incremental refinement process, and leniency towards erroneous inputs. We aim to make visual novel development more attainable and enjoyable for a wider audience.**

***Index Terms*—Visual Novel, Compiler**

I. Introduction

Visual novels[1] blend static graphics—typically anime-style—with narrative text, sound, and music to create immersive stories. The development barrier for visual novels is relatively low compared to other gaming genres, which has broadened the range of individuals who can engage in creating visual novels. However, visual novel development is not without challenges, particularly for amateurs and newcomers prone to planning errors. This could lead to wasted effort, delayed completion, or budget overrun.

We propose the PrepPipe project to mitigate these issues, facilitating rapid prototyping in visual novel development. Central to this effort is a *prototyping compiler* that processes stories in early iteration stages, generating project files for engines like Ren'Py [2]. This approach allows developers to produce functional game demos and collect asset usage statistics "starting from day one", facilitating better-informed planning and decision-making. At the time of writing, this project is a work-in-progress with a working prototype[2].

To better support the usage scenario above, our compiler and language contrast several key aspects with traditional visual novel engines. Firstly, our language is designed to align with how humans naturally write stories, accepting inputs such as rich-text documents and effectively handling non-formal grammar. Secondly, rather than interpreting input commands as imperative instructions, our compiler treats them as flexible constraints. This approach enables significant enhancements in the output, allowing users to incrementally improve stories or projects and delegate less critical details to the compiler. Thirdly, the compiler adopts a lenient approach to error handling; it treats most issues (typos, missing assets, etc) as warnings, minimizing disruptions during the creative process and focusing on development continuity. In addition, the compiler architecture supports analyses and transforms as extensible and reusable compiler passes. These include integrating asset usage statistics reporting and presentation improvement heuristics into the pipeline.

[1]We use the definition of Visual Novels from [1] but also include Kinetic Novels, which do not have branching storylines.

[2]Available: https://github.com/PrepPipe/preppipe-python

979-8-3503-5067-8/24/$31.00 ©2024 IEEE

II. Background

A. Visual Novel Engines

Visual novels generally share the same program logic, differing only in story and game assets. This has popularized the use of specialized visual novel engines like Ren'Py [2] for their development. These engines serve as domain-specific languages (DSLs) and compiler-interpreters tailored to visual novels. By adhering to specific grammar and formatting requirements for assets, developers can create visual novel games using these engines without writing traditional game programs. Listing 1 shows example scripts from selected free and open-source engines.

However, the direct use of engine-specific languages for writing visual novels is uncommon among developers. These languages often incorporate traditional programming syntax, which can disrupt creative writing. They usually feature fail-stop semantics, where minor typos can halt the game at runtime, forcing developers to resolve these issues to get a working prototype. Additionally, the semantics of these languages are engineered to define game behavior precisely, demanding explicit commands to achieve the desired on-screen presentation. Furthermore, these languages use half-width punctuation for commands, whereas the full-width punctuation used in Chinese, Japanese, and Korean (CJK) is not supported, necessitating frequent switching of input modes for CJK authors. Consequently, it is more common for developers to draft their stories in a writer's native style and then programmatically convert them to engine scripts through methods such as search-replace or custom scripting.

```
define e = Character("Eileen") # Declare character
label start:          # The game starts here.
    scene bg room     # background: "bg room.png/jpg"
    show eileen happy # This shows a character sprite.
    e "Hello World!"  # say statement
    return            # This ends the game.
```

(a) Ren'Py (.rpy, DSL based on Python)

```
*start|Prologue                    ; game starts here.
@bg storage="BG14a"                ; background
@fg pos="center" storage="fg01_02" ; character sprite
@dia                               ; show dialogue box
Hello World![w]                    ; say statement
@gotostart ask="false"             ; ends the game
```

(b) KRKR/KAG (.ks, DSL based on TJS2)

```
monogatari.characters ({'e': {name: 'Eileen'}});
monogatari.script ({
  'Start': [               // start label
    'show scene #f7f6f6',  // background
    'e Hello World!',      // say statement
    'end'                  // ends the game
],});
```

(c) Monogatari (.js, JSON object in Javascript)

Listing 1: Examples of Visual Novel Engine Scripts

B. Visual Novel Development Procedure

Developing a visual novel generally follows this procedure:

1) Pre-production: The story writer writes the story outline, which determines the characters, scenes, general plots, and, in turn, the asset requirements.
2) Production: The team writes the story text, produces all the required assets in parallel, and integrates them into a game project. Story text is converted to engine scripts during integration. After all dependent assets are ready, the developers adjust the presentation ("staging").
3) Post-production: The team does final checks and prepares for the game release.

Because of the lack of interactivity and the focus on storytelling, visual novel development traditionally does not involve playtesting; a team may not have a working demo until staging or the post-production stage, which can be months from the start of the project. Projects led by amateurs also frequently exceed their planned timelines, increasing the risk of contributors leaving due to personal commitments or loss of interest, which can negate earlier efforts.

Another pitfall for inexperienced teams is asset planning. An ambitious writer leading the project may write the bulk of the story text before collecting asset requirements, leading to wasted writing effort or budget overrun. The cost-effectiveness of certain assets may be questionable, such as dedicating a CG to an insignificant scene with just one or two sentences of dialogue. We believe that quantitative asset usage estimates can guide better planning on asset production and enhance resource efficiency.

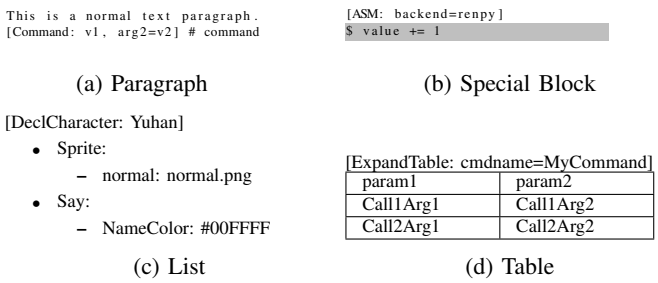
III. Design Overview

PrepPipe project consists of (1) a rich-text-based language for visual novel storywriting, (2) a compiler that generates visual novel engine project files, and (3) auxiliary tools (mainly a GUI) and art assets to support the compiler. We designed the language to suit the natural writing style of narrations and dialogues while enabling programmatic parsing and compilation. We developed the PrepPipe compiler in Python, facilitating easier plugin development and user customization. Section III-A describes the language and how the design makes it accessible to general users. Section III-B explains how the compiler is designed for compiling the language and how to use the compiler in the development workflow.

PrepPipe compiler is designed for the pre-production and production stage before staging (i.e., adjusting the presentation), allowing the writer to create prototypes and get analysis reports from the story text (and assets, optionally). The team can then use these outputs to re-iterate and improve the story and the assets. Near the end of the production stage, the team can use the compiler to generate the initial project files and start adjustment from there. We hope the availability of early prototypes will reduce team members' stress and interest loss, making the development experience more enjoyable.

A. Language Design

We designed the language on rich-text input because we observed that writers tend to write the initial story in rich-text editors like MS Word. In this subsection, we highlight our language design innovations. We explain how these innovations stem from our design goals and the specific features of rich-text editing environments.



(a) Paragraph (b) Special Block

(c) List (d) Table

Fig. 1: Structural Elements and Example Usage

```
Yuhan: "Sentence 1."
Yuhan: Sentence 2.
Yuhan "Sentence 3."
Yuhan (smile): "Sentence 4."
Yuhan: (smile) Sentence 5.
Yuhan: "Sentence 6", ignored, "continued."
"Yuhan" Sentence 7.
[Yuhan] Sentence 8.
```

Listing 2: Say Statement Formats (Non-exhaustive)

The PrepPipe language is line-oriented for error recovery and consistency with existing engine scripts. Any structured rich-text input is first read as a sequence of basic structural elements, as shown in Figure 1. A paragraph consists of text or embedded asset elements, and it does not contain paragraph styles (e.g., align center or has background color).

If the paragraph does not start with '[', it is a content paragraph. The compiler will first try to parse the content as a say statement (i.e., there is a sayer, a content string, and optionally a character state modifier for sprite change). Listing 2 lists a subset of say statement formats recognized by the implementation. If the recognition fails, the paragraph will be treated as a narration (plain text) or asset display.

If a paragraph starts with an opening square bracket '[', it is considered a command paragraph and it should contain one or more commands enclosed by square brackets. All operations other than narrations and dialogues are provided as commands, including scene changes, characters entering/leaving the scene, background music changes, etc. To support users of different native languages, all strings used in the compiler (including command and parameter names) have translations in supported languages and are customizable from configuration files. In addition, syntactic symbols like brackets and quotes support both half-width and full-width versions.

Besides paragraphs, the remaining structural elements in Figure 1 may be used as an additional parameter to the last command in the preceding command paragraph. If a paragraph contains a background color or has a center alignment, it becomes a special block. It is used for inlined engine-specific scripts (like inline assembly in traditional programming languages) or figure titles. Lists are used extensively for optional parameters and extensible declarations. Tables are useful for workflows using spreadsheets instead of texts. The language ensures that non-content texts are visually distinct from story content. When a reader inspects the content, unfamiliar command blocks can be skipped without affecting the story.

a) Incremental Refinement: PrepPipe Compiler treats input commands as constraints instead of imperative instructions. This means that even in the absence of commands, the compiler can automatically insert them, and it defaults unspecified parameters to auto-determined values. This semantics allows the compiler to automatically add placeholder assets and perform basic staging using heuristics. During the initial story writing, the author can focus on writing the story without thinking about on-screen presentation details. The compiler can add placeholder images to the screen at computed locations and set their transition effects. After the initial iterations, the author can incrementally add commands or command parameters that adjust the presentation, and the PrepPipe Compiler will observe the new constraints in the new output. Because of the language design, the additional commands or parameters can be in new paragraphs or list elements, preserving the readability of existing content. This is especially necessary in a rich-text editing environment because the line width is finite and line wrap is mandatory. Long commands exceeding line width have poor readability because monospace fonts are not defaults, therefore whitespace becomes ineffective in formatting. We do not aim to fully support staging from the input though, as it relies on engine-specific features that vary significantly across different engines.

b) Lenient to Error: PrepPipe Compiler handles errors gracefully, aligning with the objective for prototyping. It emits most error messages as a special character named "PrepPipe Error" describing the error in the game (i.e., saying "Error XXX. Please YYY. (error-code)" as a game character). "Asset not found" errors become a placeholder that replaces the referenced asset. If a scene, character, or other entities are not declared, the compiler creates an empty declaration and proceeds. This allows the author to prioritize completing the draft and making high-level decisions (e.g., about plot design) without bothering to fix the errors in the specific writing.

c) Embedded Assets: Rich-text content can embed images, audio, or other multimedia assets. Our language supports embedded assets in the content or as command parameters. While they are rare in text scripts, we believe that supporting embedded assets from inputs is necessary to explore the usability of rich-text input fully.

B. Compiler Design

To support the compilation of the visual novel language, the PrepPipe compiler uses the pipeline design shown in Figure 2.

a) Compiler Architecture: We follow the architecture of modern compiler frameworks (LLVM [3] and MLIR [4]): the compiler is decoupled into (1) *frontend* that implements file reading and language semantics handling, (2) *backend* that implements the engine-specific code generation, and (3) *middle-end* that defines an abstraction interface between a frontend and a backend, as well as implementing engine-agnostic algorithms. At any abstraction interface, primarily the middle-end, the data structure representing game or story content is known as *Intermediate Representation* (IR). This architecture enables flexible customization and seamless extension of new functionalities. For example, code generation of a new visual novel engine can be integrated into the pipeline by implementing another backend that reads the IR; all the frontend logic can be reused without change. *Analyses* like statistics collection and *transforms* like game/code improvement heuristics can be implemented in the middle-end as compiler *passes* (which reads IR and optionally produces transformed IR), and they work regardless of the combination of frontends and backends. We also incorporated the idea from MLIR [4] that a base IR provides common utilities for code reuse and ease of debugging. This allows us to further divide the pipeline and create abstraction interfaces, improving the flexibility and speed of iterating the language design.

b) Pipeline Stages: To support multiple input file formats and simplify parsing, the compiler defines a data structure called `InputModel` to abstract away differences in input file formats while preserving the structural elements in Figure 1. In this way, supporting a new file format only involves writing a new frontend component that exports the `InputModel` for this file format. Then, a command detection pass operates on the `InputModel` to replace command paragraphs with parsed command data structures. Next, the visual novel language frontend takes over and produces the abstract syntax tree (AST). Each AST node may represent certain meaningful constructs in the language, for example, a parsed say statement, a character/scene declaration, a command, or an asset reference.

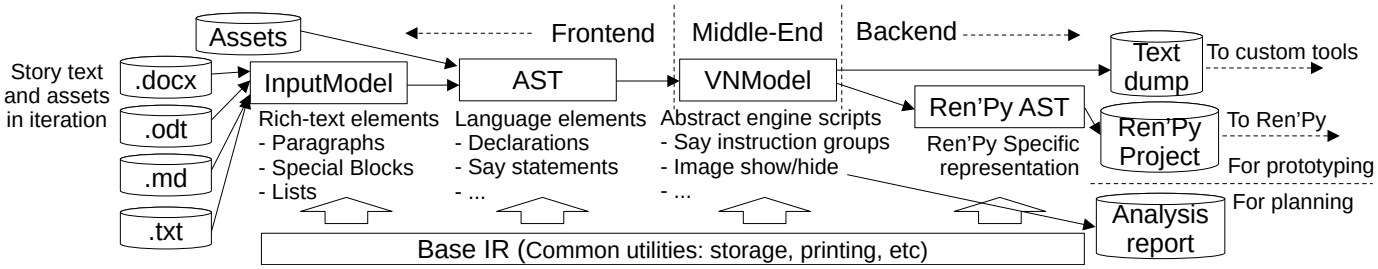


Fig. 2: PrepPipe Compiler Pipeline Overview

The AST interface abstracts away differences in *syntax* for all constructs with the same *semantics*: say statements in different forms (e.g., Listing 2) are all emitted as say statement nodes, and the same command specified in different human languages or using user-specified keywords are represented with the same AST node types. This unification allows subsequent pipeline stages to handle different forms of say statements or commands in the same way. Then, a visual novel code generator visits the AST and emits `VNModel`, an IR representing the script for an abstract visual novel engine. The code generator implements the semantics of each AST node using imperative commands from the abstract engine. For example, character entry/exit becomes character sprite show/hide, and control flow commands (e.g., menu commands that ask the player to choose a branching path) become *basic blocks* (contiguous blocks of instructions without branching in between) and *terminator* instructions (branching instructions at the end of each basic block). Say statements become say *instruction groups* that contain child instructions for showing character names, say content, side images, etc. After `VNModel` is generated, the pipeline can execute analysis and transforms, including asset usage statistics collection. We also implemented a pass that breaks long says into shorter ones so writers can write long monologue paragraphs without overflowing the resulting game's dialogue box. At the time of writing, for producing the prototype, we have (1) a Ren'Py backend that supports exporting Ren'Py project files and (2) a text dump pass in the middle-end that exports IR information in plain text. Developers using in-house engines can write custom scripts that parse the text dump to integrate the compiler into their workflows.

IV. Discussion

A. Asset Templates

Usually, a visual novel project starts with no image assets prepared in advance. Playing the output prototype without image assets will be boring and meaningless, as only a dialogue box is popping text with a black background. To generate placeholder images that are both nice-looking and general, we work with artists to draw pictures with reserved areas for customization. For example, in a template background image, there is a white screen on the wall where the user can attach an image, a text fragment, or color blocks, and there is an item (e.g., a carpet or a wall) that the user can recolor. Users can customize the templates according to their needs (e.g., writing the scene name on the screen) to create non-trivial game prototypes. We plan to add more user-customizable asset templates in the future.

B. Usable Tooling

PrepPipe project originated from the observation that many visual novel writers use rich-text editors like MS Word, often without the programming skills to convert their stories into engine scripts. They may also use a writing style that is challenging to process programmatically. Before the advent of Large Language Models (LLM), the developers needed to replace the dialogue text with corresponding engine scripts line by line, which is tedious and can take days or weeks.

We started the project by designing a rich-text-based syntax that matches how storywriters write conversations in natural languages. It later evolved into a compiler that assists visual novel prototyping on top of the story-to-script conversion. We hope this project can enhance the experience of visual novel development, enabling more creators to engage in this genre. Future extensions will maintain an emphasis on usability for the general public.

C. Relationship to Related Work

Visual novel engines and the PrepPipe compiler can be categorized as authoring tools for Interactive Digital Narratives (IDNs) [5]. We believe that PrepPipe compiler is the first work that can prototype visual novels from rich-text inputs. We plan to implement more frontends/backends to interface with other visual novel engines and IDN-related tools in the future.

References

[1] J. Camingue, E. Carstensdottir, and E. F. Melcer, "What is a visual novel?" *Proc. ACM Hum.-Comput. Interact.*, vol. 5, no. CHI PLAY, oct 2021. [Online]. Available: https://doi.org/10.1145/3474712

[2] T. Rothame, "Ren'Py," https://www.renpy.org/, accessed: Jun 9, 2024.

[3] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[4] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[5] C. Hargood and D. Green, *The Authoring Tool Evaluation Problem*. Cham: Springer International Publishing, 2022, pp. 303–320. [Online]. Available: https://doi.org/10.1007/978-3-031-05214-9_19