Sustainable Visual Editor Solution for the Development of Serious Visual Novels

An extension-based approach using a game engine

Author
José Coutinho
1628275

Supervisor
Prof. Dr. Remco Veltkamp

Examiner
Dr. Laura van der Lubbe

Game and Media Technology MSc Thesis

Department of Information and Computing Sciences
Utrecht University
Netherlands
July 22, 2026

Abstract

Researchers increasingly use Visual Novels (VNs) as interactive narrative environments for studies in behaviour and education, but often lack the technical skills to build games. A common solution is building a specialised visual editor, tailored to a specific study. However, such tools are costly to develop and hardly adaptable to new requirements, as demonstrated by the case motivating this thesis, forcing researchers to rebuild similar systems repeatedly. This thesis investigates general-purpose game engines as a more sustainable foundation for VN authoring tools in research contexts, while remaining similarly intuitive. It first conducts a comparative feature analysis of representative purpose-built tools, VN engines, and VN extensions for general-purpose engines, showing that VN extensions already match VN engines in essential feature coverage while inheriting the flexibility of their base engines, though some extensions are less mature and still trail on secondary features. It also identifies alternative player input features as a shared gap across existing tools. It then produces a proof-of-concept VN authoring solution by extending Dialogic, a VN extension for the Godot engine, with alternative input features required by the motivating case. The overall design and architecture of the solution are analysed, showing that Godot's scene-based architecture serves as a flexible foundation for developing feature extensions. Usability is also evaluated through a single-participant study with the researcher from the motivating case, showing that they perceive the tool as a coherent VN authoring environment, despite some confusion with the scene system caused by their previous knowledge of general-purpose game engines. These findings give a positive but preliminary answer to the suitability of game engines as a foundation for sustainable VN authoring tools, given the limited scope of this thesis.

Contents

1 Introduction **3**
- 1.1 Goals and Research Questions . . . 3
- 1.2 Overview . . . 4

2 Visual Novels **5**
- 2.1 Interactive Storytelling . . . 5
- 2.2 Presentation and Gameplay . . . 5
- 2.3 Alternative Interaction and Genre Blending . . . 6
- 2.4 Serious Applications . . . 6

3 VN Development Tool Landscape **8**
- 3.1 Propellant and Purpose-Built Systems . . . 8
- 3.2 What is a Game Engine? . . . 9
- 3.3 Specialised Engines . . . 10
  - 3.3.1 VN Engines . . . 10
- 3.4 General-Purpose Engines . . . 11
  - 3.4.1 VN Extensions . . . 12
- 3.5 Comparative Feature Analysis . . . 15
  - 3.5.1 Features . . . 15
  - 3.5.2 Analysis . . . 17
  - 3.5.3 Tool Choice and Development Scope . . . 17

4 Solution Description **19**
- 4.1 Godot . . . 19
- 4.2 Dialogic . . . 20
- 4.3 Numeric Input . . . 22
- 4.4 Interactive Scenes . . . 23

5 Evaluation **25**
- 5.1 Procedure . . . 25
  - 5.1.1 Pre-session . . . 25
  - 5.1.2 Authoring Session . . . 26
  - 5.1.3 Post-Session Interview . . . 26
  - 5.1.4 Data Collection and Analysis . . . 26
- 5.2 Results and Discussion . . . 27

6 Conclusion **31**

1 | Introduction

Researchers increasingly turn to interactive narrative environments for studies in behaviour and education. One popular format is the Visual Novel (VN), a distinct genre of interactive narrative games characterised by simple gameplay and visual presentation. These researchers often lack the technical skills required to develop a game, and therefore need creation tools with an accessible learning curve and an intuitive visual interface. They may also need support for any specific features their research demands.

A common solution in academic environments is to have specialised systems built from scratch, tailored to the researcher's needs. However, this approach requires significant development time and effort, while academic environments often have limited resources. Additionally, these systems are hardly adaptable across similar projects with slightly different requirements, resulting in a cycle of building new systems. These two issues (high development cost and poor reusability) together define the sustainability problem this thesis addresses.

As an example directly motivating this thesis, a researcher wanted to create VNs to study decision-making related to drug trafficking scenarios among a specific demographic group. They attempted to reuse Propellant, a web-based visual editor for creating police training scenarios, developed in a previous student project. The system needed to be extended to support certain requirements, including numeric and text responses, but the architectural complexity and tightly coupled, domain-specific design made this task prohibitively difficult.

A reusable solution for developing VNs should be flexible, both in the baseline functionality it provides and in the architectural design that supports extending it. Most modern game projects are developed using a game engine: a software framework designed for building games. Game engines are designed to be this type of reusable solution, making them especially relevant to this thesis.

The landscape of game engines shows a great variety of designs, workflows, and genre targets. Two categories are particularly relevant here: engines specialised in the VN genre, and engines designed to support any genre (general-purpose). While both types have been used to create serious VNs, they differ in the features and user experience they offer authors, as well as in the architecture, modularity, and extensibility they offer developers. However, work evaluating these tools along either dimension remains scarce.

1.1 Goals and Research Questions

The main goal of this thesis is to produce a proof-of-concept VN authoring solution, based on a game engine, that addresses specific requirements of the motivating case study (later referenced as "original requirements"), and aims to be extensible and reusable in other projects. At the same time, the leading research question is formulated as follows: *How suitable are game engines as a foundation for sustainable Visual Novel authoring tools in research contexts?*

This question is notably broad, due to the many different types of game engines, and suitability spanning both usability and extensibility. The scope of this thesis will focus on general-purpose game engines, since their use cases include but are not limited to VN development, a valuable aspect considering the experimental nature of research requirements. This thesis seeks to answer the following research questions:

RQ1: *How does the engine's general-purpose architecture shape the design and implementation of VN-related editor and gameplay features?*

RQ2: *How does the general-purpose editor environment influence the VN authoring experience?*

The first question is answered by analysing and reflecting on the design and implementation of the VN-related features of the solution. The second question is answered through user studies on the usability of the solution. This thesis contributes knowledge about the architectural and usability aspects of VN extensions for general-purpose game engines, besides the proof-of-concept solution itself.

1.2 Overview

Chapter 2 describes the VN genre to ensure familiarity with its main elements, as they are relevant for understanding the design and features of a VN authoring tool.

Chapter 3 discusses the limitations of Propellant in more detail, and explains the fundamentals of VN tools and game engines through representative examples in the landscape. It also includes a comparative analysis of the feature coverage across these VN tools, and closes by discussing the choice of tools and development scope for the solution.

Chapter 4 describes the produced solution, covering the design of the base engine, VN extension, and all implemented features. Chapter 5 covers the evaluation of the solution, describing the user-study's design and the resulting observations.

Finally, Chapter 6 concludes the thesis, and discusses its limitations and future work.

2 | Visual Novels

Being familiar with the VN genre is important for understanding the features, affordances, and workflows of VN editors, explained in the other chapters of this thesis. This chapter describes VNs, drawing from the unified VN genre definition provided by Camingue et al. [1] — derived from the analysis of a corpus of existing definitions and released VNs, capturing the most common features of the genre — as well as personal observations from current released VN titles.

2.1 Interactive Storytelling

Interactive Storytelling is a narrative model where the storyline is not completely predetermined, instead providing player with agency over certain parts of it. This results in the player experiencing a unique story, based on their decisions and interactions within the narrative world. Dialogue choice menus, seen in Figure 2.1, are a very common form of interactive storytelling, and an essential part of VNs.

Figure 2.1: Examples of dialogue choices in narrative games. Screenshots of *The Murder of Sonic the Hedgehog* (a VN) and *Baldur's Gate 3* (a Role-Playing game), respectively.

2.2 Presentation and Gameplay

The VN genre has a simplistic style of gameplay and visual presentation. In terms of visuals, the scenes usually consist of static 2D art, including a background and character portraits, as seen in Figure 2.2. Throughout the story, characters frequently switch between different portraits to convey distinct poses or expressions. VNs usually use simple visual effects, such as transition animations for smoothing changes to the scene, or speaking character highlights to help follow the dialogue between characters.

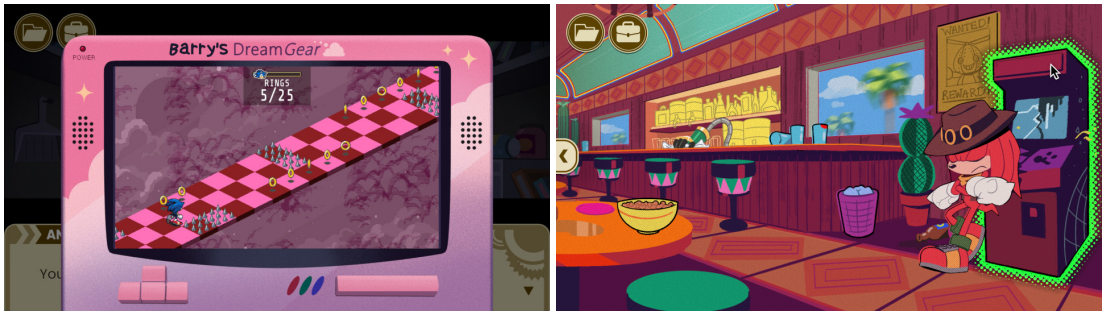
The gameplay of VNs revolves around on-click progression, where the user taps to advance the text in the textbox or select an option within a choice menu. The gameplay structure is comparable to classic narrative game genres, such as hypertext narratives or even gamebooks. However, VNs may include a variety of gameplay formats surrounding the main gameplay style.

Figure 2.2: Examples of the typical visual presentation in VNs. Screenshots of *Phoenix Wright: Ace Attorney* and *STEINS;GATE*, respectively.

2.3 Alternative Interaction and Genre Blending

While VNs mostly follow their traditional gameplay formula, there are often cases where they expand with new types of interactions, or integrate whole sections built around a different genre's gameplay, as seen in Figure 2.3. A common example is exploration in a point-and-click adventure game fashion — players interact with scene elements, triggering story events such as initiating conversation with a character — as it aligns naturally with the on-click progression in VNs. The text and numeric responses mentioned as part of the original requirements are examples of alternative interaction as well.

Essentially, these alternative interaction features and genre-blending gameplay represent the possible variety in game design, which has direct implications for the flexibility and extensibility required of development tools, and is central to the discussion in the following chapters.



(a) Minigame (b) Exploration

Figure 2.3: Examples of alternative interaction in *The Murder of Sonic the Hedgehog*. In (a), the player must succeed in an auto-runner minigame to advance the story, and in (b) they can interact with objects and characters in the scene to investigate for clues.

2.4 Serious Applications

VNs are becoming an increasingly attractive medium for serious applications, due to their simplistic gameplay, as well as being relatively easy to develop compared to other game genres. For instance, Grasse et al. [7] employed a VN to support the teaching of responsible research conduct. More broadly, the taxonomy by Camingue et al. [2] identifies multiple teaching strategies present across a range of educational VNs, suggesting that the medium has a variety of pedagogical approaches.

Besides education, VNs and other interactive narrative games are used as research in-

struments to study player behaviour across dimensions such as roleplay and decision-making [5, 11, 3]. This project's motivating case study, mentioned in Chapter 1, contains yet another example of this use case.

3 | VN Development Tool Landscape

3.1 Propellant and Purpose-Built Systems

Propellant is a police training scenario editor developed as part of a student project at Utrecht University. These scenarios use the gameplay and visual presentation format of VNs, as seen in Figure 3.1.

Figure 3.1: A training scenario as it appears to players, provided in Propellant's documentation.

The editor is designed for non-technical users, with a visual interface and a feature set streamlined for the training use case. Propellant structures stories as a collection of connected scenes. For each scene, trainers interactively place the scene elements using a canvas interface, shown in Figure 3.2, and write a narrative script using a flowchart-based interface, shown in Figure 3.3.

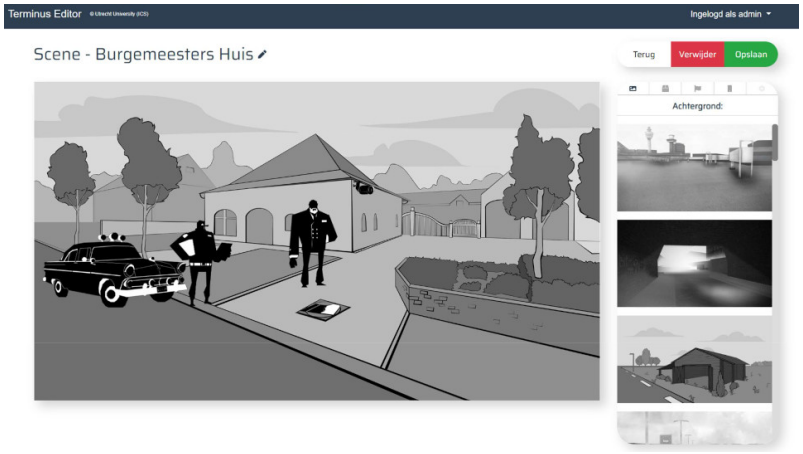


Figure 3.2: Screenshot of Propellant's visual scene composition interface, provided in its documentation. Users select or drag elements from the library on the right, placing them on the scene canvas on the left.

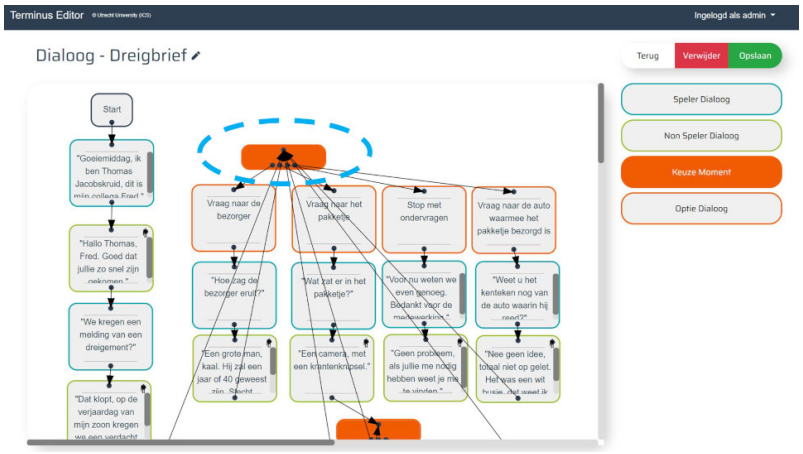


Figure 3.3: Screenshot of Propellant's narrative script authoring interface, provided in its documentation. Users compose a flowchart using a set of nodes: the blue nodes represent player speech, the green nodes represent the non-player speech, and the orange nodes represent choices.

Propellant is part of a system with two other components: a mobile game application through which trainees play the scenarios, and a server application responsible for storing scenarios and player progress. This web architecture simplifies content deployment, allowing trainers to create and update scenarios without having to rebuild and redistribute the game application.

While the system works well for its intended purpose, its domain-specific design limits its applicability in other projects. The system uses a fixed library of assets and only two characters (a player character and a single non-player character), with no support for importing new images or defining new characters. Adapting the system to make its design more general and add new features requires changes across all three components, not just the editor application. Since each component uses a distinct technology stack, these adaptations require considerable effort. Additionally, this system's web architecture adds maintenance efforts regarding the server infrastructure.

These issues with tight coupling and system architecture are not unique to Propellant. RichCast [6] is a web platform on which users can create, publish, and play voice-driven VNs. The platform's editor also features a visual interface designed for non-technical users, but it has a specialised structure for authoring voice-driven content. Since the editor is integrated into a web platform, adaptations likely require changes across the rest of the system as well.

While these systems work well for their original use cases, their domain-specific designs and complex architectures make them difficult to reuse or adapt for other projects. Consequently, this thesis explores using established foundations that are already designed with modularity and extensibility in mind.

3.2 What is a Game Engine?

Video games combine technical and creative work: developers write code that defines gameplay and player interaction, and they configure art assets such as images, 3D models, and audio files. Underneath this, a game also needs common technical infrastructure — such as rendering graphics to the screen, detecting collisions between objects, playing audio, managing input from the player — which is a substantial undertaking to build

from scratch.

Game engines are software frameworks that provide this technical infrastructure, enabling developers to focus on the creative aspects of a game rather than reimplementing this low-level functionality for every project. A game engine typically consists of two main parts: a core engine handling the functionality described above while the game is running, and an editor with a set of tools supporting content creation, such as scripting, level design, and asset management [14].

3.3 Specialised Engines

Games can be very distinct in terms of gameplay across genres, even if they share common ground on the technical side. Specialised engines focus on a specific genre, which allows their core and editor to be optimised for a narrower range of use cases. They reflect developers' common needs across the genre, streamline the development process, and provide a cohesive feature set.

3.3.1 VN Engines

VN engines support the game elements described in Chapter 2 and facilitate the authoring process. The heart of any VN engine is the narrative scripting system, which authors use to compose the story. Authors write scripts as a sequence of story events that describe scenes and character dialogue, conceptually resembling screenplay writing.

Narrative scripting systems use a dedicated narrative scripting language that abstracts away technical complexity found in programming languages, making the system accessible to authors without programming experience, and allowing them to focus on the narrative itself. These dedicated languages usually still allow authors to optionally integrate basic programming constructs within the scripts, including variables and conditional logic, which can be used to create small story variations as well as complex branching structures. Narrative scripting languages usually come in two forms, text or visual, which influence the overall engine design and authoring experience.

The most prominent example of a VN engine is Ren'Py[1]. It is by far the most widely used in the VN space, having the most releases according to VNDB[2], and being often acknowledged in academic literature. It revolves around a text-based authoring workflow, and the scripting language blends Python and screenplay syntax, designed to be powerful and flexible while remaining approachable. Figure 3.4 shows an example script, illustrating the structure and syntax.

TyranoBuilder[3] is another popular VN engine. In contrast to Ren'Py, it features a visual editor with a visual scripting language, allowing authors to write the script by dragging story event nodes from a library into a timeline, as seen in Figure 3.5. While this language is fundamentally similar to that of Ren'Py, the visual design prioritises simplicity and intuition over flexibility and writing efficiency.

Asset management is another important part of a VN engine. VN projects accumulate a large number of assets, especially images for different backgrounds and character poses. These assets need to be organised and configured for use within the narrative. VN engines usually provide specialised character management, since characters are complex

[1] https://www.renpy.org/
[2] https://vndb.org/r
[3] https://tyranobuilder.com/

```
# Define characters
define e = Character("Eileen")

label start:
    scene bg meadow
    show eileen happy at right

    e "Welcome to my project! Are you enjoying the demo?"

    menu:
        "Yes, it's great!":
            jump positive_response
        "I'm still learning.":
            e "That's okay! We all start somewhere."

label positive_response:
    e "I'm glad to hear that!"
    return
```

Figure 3.4: Example of a narrative script in Ren'Py

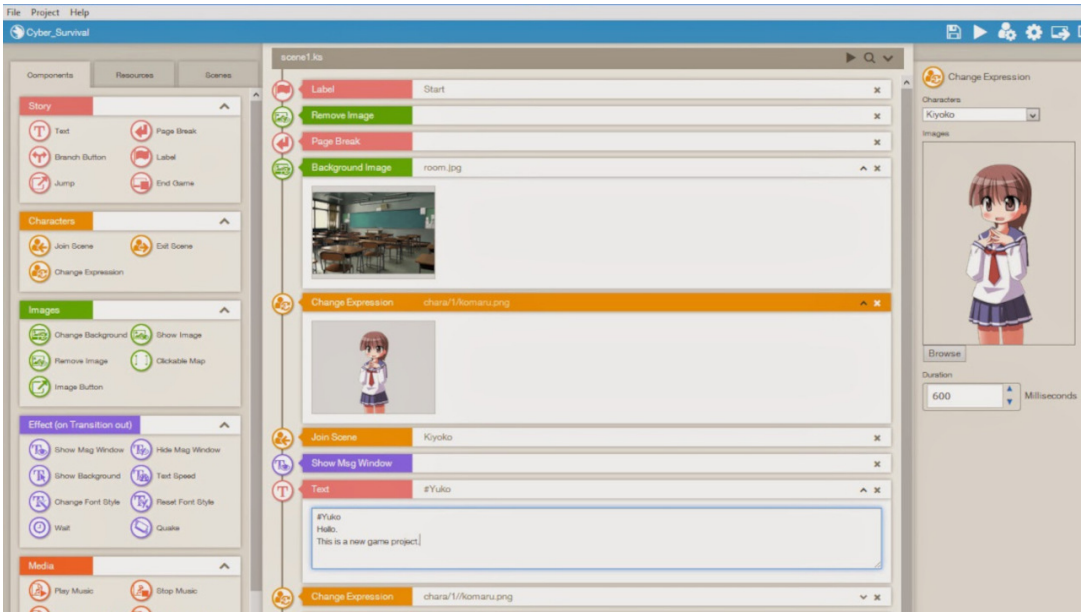


Figure 3.5: Example of the visual scripting language in TyranoBuilder

entities that often have dozens of images for different expressions and poses, along with multiple configurable properties.

However, as VN engines are optimised for VN development, their features and affordances centre around traditional VN format. As Consalvo [4] observes from their experience with Ren'Py, attempting to include non-traditional gameplay creates friction with the engine's framework. Despite already being more flexible than purpose-built systems like Propellant, this friction could pose a potential issue when experimental requirements in research contexts fall outside of the traditional format.

3.4 General-Purpose Engines

General-purpose engines support the development of virtually any genre. Their design prioritises modularity and flexibility, and provides the technical foundation shared by all

games. Unity[4] and Godot[5] are two established examples. Unity is an industry-standard commercial engine, also used outside games for applications such as training simulations and architectural visualisation. Godot is a free and open-source engine that has seen a steep rise in popularity in recent years, becoming an established alternative to Unity [10].

Composition is a core concept in general-purpose game engines: each game object is assembled from smaller, reusable components that define its behaviour. For example, an object can be made visible by attaching a component that renders an image, and made physical by attaching a component that handles collision. The specific mechanism engines use to enable this composition varies by design. These objects are organised into a scene tree: a hierarchical structure representing everything present in a game world, such as a level or a menu. The scene tree captures the state of the game at runtime, continuously processed and updated by the engine as it handles input, physics, and other systems. The engine's editor provides tools to interactively assemble game objects and arrange them within a scene, illustrated in Figure 3.6.

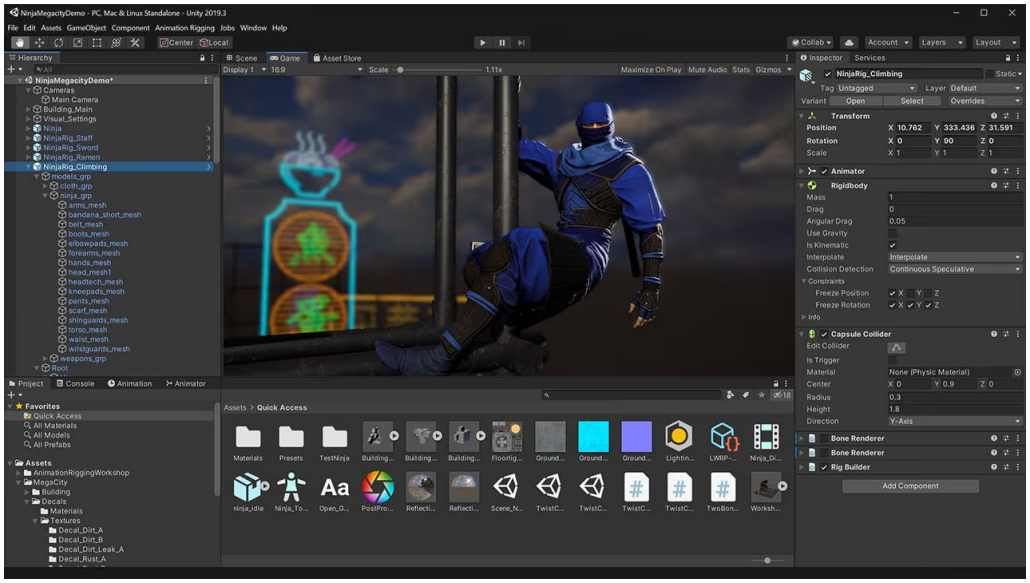
In addition to the scene system, general-purpose engines expose a programming framework for implementing custom behaviour, going beyond what the engine provides out of the box. This extensibility allows developers to add new components and editor tools for VN development, instead of building a new VN engine from scratch.

3.4.1 VN Extensions

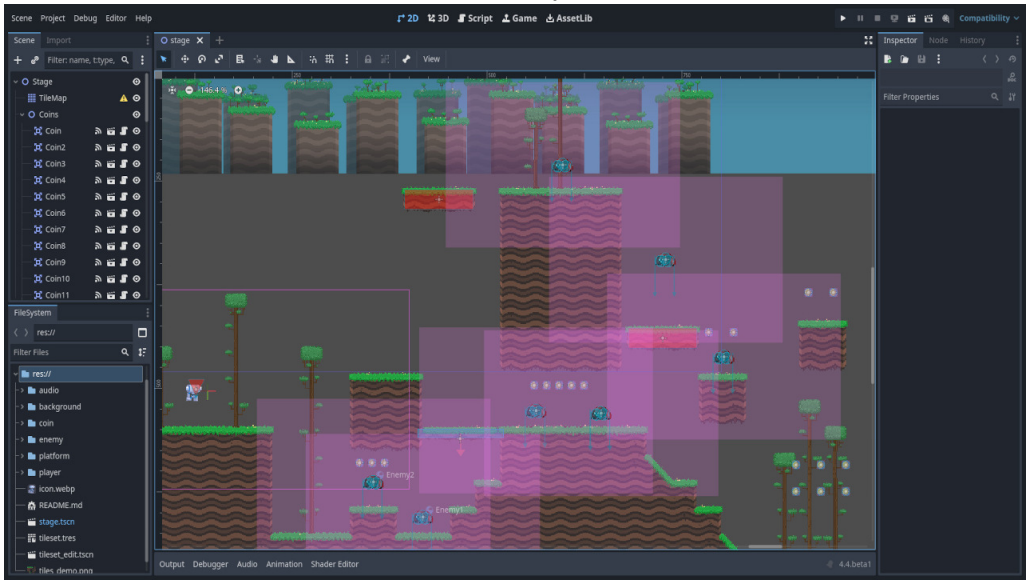
A VN extension for a general-purpose game engine adds features found in VN engines, especially the narrative scripting system essential to developing any VN. Unity and Godot each have a couple of extensions available. For Unity, the most notable options are Naninovel[6], a commercial extension, and Fungus[7], a free and open-source alternative. For Godot, the most notable option is Dialogic[8], also free and open-source. Figure 3.7 shows examples of the different authoring environments provided by these three extensions. Unlike VN engines, a general-purpose engine with a VN extension is not limited to traditional VN features, but it also exposes the author to a general-purpose editor environment which is inherently more complex and may have potential implications in usability and user experience.

This extension-based approach has already been explored in research contexts, though such work remains scarce. Anansi [12] is a Unity extension combining Ink[9] with a custom social simulation engine, used to build procedural, location-based VNs. M22 [13] similarly aims to bring a Ren'Py-like scripting system to Unity, but remains an undeveloped academic prototype. This thesis aims to further explore this space.

[4] https://unity.com/
[5] https://godotengine.org/
[6] https://naninovel.com/
[7] https://github.com/snozbot/fungus
[8] https://dialogic.pro/
[9] https://www.inklestudios.com/ink/

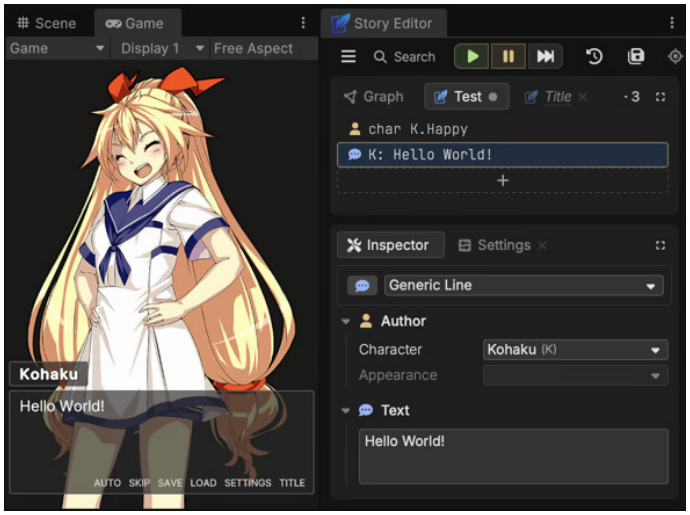


(a) Unity



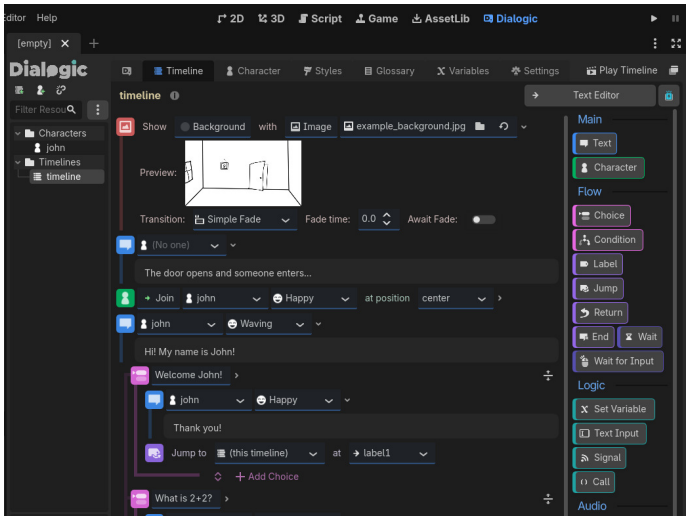
(b) Godot

Figure 3.6: Scene editor of Unity and Godot. Both show the scene canvas at the centre, the scene tree at the top left, the file system at the bottom left, and the inspector panel at the right.



(a) Naninovel

(b) Fungus



(c) Dialogic

Figure 3.7: Authoring environment of Naninovel, Fungus, and Dialogic within their respective engines.

3.5 Comparative Feature Analysis

This section further analyses and compares the representative tools — Propellant, Ren'Py, TyranoBuilder, Naninovel, Fungus, and Dialogic — in terms of VN-related feature coverage. The features and respective coverage are drawn from available documentation and community material such as tutorial videos, and shown in Figure 3.8.

	Purpose-built editor/system	*VN engine*		*VN extension*		
	Propellant	**Ren'Py**	**TyranoBuilder**	**Naninovel**	**Fungus**	**Dialogic**
Authoring Features						
Text Scripting		✓		✓		✓
Visual Scripting	✓		✓	✓	✓	✓
Graph-based Story Structure Visualisation	✓			✓	✓	
Dynamic Scene Composition		✓	✓	✓	Partial	✓
WYSIWYG Scene Editing	✓		✓		Partial (via Unity)	
Asset Management		✓	✓	✓	✓	✓
Fast-Track Testing			✓	✓		✓
Common VN Features						
Dialogue History		✓	✓	✓		✓
Glossary / Journal		✓		✓		✓
Save & Load	✓	✓	✓	✓	✓	✓
Visual Effects		✓	✓	✓	✓	✓
Specialised or Advanced Features						
Variables & Conditional Logic		✓	✓	✓	✓	✓
Localisation		✓	✓	✓		✓
Game UI Customization		✓	✓	✓	✓	✓
Show Video		✓	✓	✓		
Advanced Animation System		✓	✓	✓		
Layered Character Composition		✓		✓		✓
Interactive Scenes / Hotspots		✓	✓	✓	✓	
Text Input		✓	✓	✓		✓

Figure 3.8: Feature coverage of representative VN tools

3.5.1 Features

The features are divided into three different groups for clarity: *Authoring Features* for the main editor features; *Common VN Features* for the support of game features found commonly across VNs; and *Specialised or Advanced Features* for features that are more complex or target specific, less common use cases. The features are described as follows:

Text Scripting: text-based narrative script authoring.

Visual Scripting: visual-based narrative script authoring.

Graph-based Story Structure Visualisation: visual graph-based models of the narrative's branching structure, such as flowcharts. These provide an intuitive mental model that improved story management and experimentation [8].

Dynamic Scene Composition: display and control each scene element individually using events in the narrative script, rather than defining the scene statically as a whole.

WYSIWYG Scene Editing: editor uses "What You See Is What You Get" paradigm, providing previews or direct-manipulation tools that enable authors to visualise scene changes immediately within the editor during iteration.

Asset Management: allow importing and organising assets such as background images, character images, audio, etc.

Fast-Track Testing: component of the editor enabling the author to directly play specific sections of the narrative for more efficient testing [8].

Dialogue History: gameplay feature that records previous narrative text and choices, accessible to the player through a dedicated panel. Allows player to remember earlier information in dense stories without needing to replay content.

Glossary / Journal: gameplay feature that contains descriptions of key story concepts, accessible to the player through a dedicated panel. Similar benefits to Dialogue History. Author defines entries in editor.

Save & Load: gameplay feature allowing player to save their progress, and later restore it. Important part of games in general.

Visual Effects: includes effects, such as transition animations or speaking-character highlights, that add visual polish.

Variables & Conditional Logic: allow defining variables and conditional statements in the narrative script, similar to programming languages. Enables advanced forms of dynamic content variation and narrative branching.

Localisation: support creating and managing translations more efficiently.

Game UI Customisation: modify the present game UI apprearance, or create new layouts.

Show Video: support displaying video files as cutscenes or animated backgrounds.

Advanced Animation System: animation features such as skeletal animations enabling more complex animations, especially for characters.

Layered Character Composition: alternative advanced format for defining characters with many different portrait variations. Characters are composed of multiple controlable layers that combine to create the different expressions and poses. Reduces asset redundancy and memory usage.

Interactive Scenes / Hotspots: define a displayable layer containing interactive images or areas that trigger story events and branch the story, functioning similarly standard choice menus. Related to the exploration game pattern described in Chapter 2.

Text Input: narrative event that displays a textbox to capture text input during the narrative script, such as a custom player name.

3.5.2 Analysis

In terms of overall feature support, the table shows Propellant lack a significant number of features compared to the other tools. This is very likely tied to the domain-specific design, as discussed in section 3.1.

Ren'Py is the only fully text-based option, and naturally lacks the visual editor features that the other tools have. Naninovel and Dialogic notably support both visual and text authoring instead of committing to a single mode, giving authors the choice based on their preferred workflow.

Propellant, Naninovel, and Fungus all support story-graph visualisation, though its exact form varies with each tool's scripting system design. This feature is notably absent from both VN engines, though it is important to note that Ren'Py has third-party tools such as BranchPy[10] that provide it. Dialogic also lacks story-graph visualisation, trailing the other two VN extensions.

Propellant is the only tool fully based around individual static scenes. The other tools support dynamic scenes, which seems to be the standard approach. Fungus uses an unique hybrid approach that leverages Unity's scene tools: authors place all scene elements in the scene tree, and can then dynamically control them through the narrative script. Between the tools with dynamic scenes, TyranoBuilder is the only to support WYSIWYG scene editing, with its interactive positioning tool. Fungus does not support previewing dynamic changes, only the static elements.

Fast-track testing is absent in Propellant and Fungus. TyranoBuilder, Naninovel, and Dialogic provide a playtest button in their editor that allows playing the current open script. Ren'Py cannot provide this type of visual feature, but it provides a developer command console that allows controlling the script executing during playtesting, though it is less accessible to non-technical users.

Looking at the features in second and third groups, Dialogic and Fungus are lacking support for a significant part of the features, compared to Ren'Py, TyranoBuilder, and Naninovel. This shows their differences in maturity: Ren'Py, TyranoBuilder, and Naninovel are all actively developed and stable, while Fungus is discontinued and Dialogic is still in an early development stage. Naninovel and Ren'Py show feature parity, and TyranoBuilder is slightly behind, lacking two features.

3.5.3 Tool Choice and Development Scope

The three VN extensions all provide the essential features for VN development. Following the reusability principle of this thesis, the solution includes one of these existing VN extensions, instead of building an extension from scratch for a chosen general-purpose engine. This reduces the development scope to only the required features that are missing, while the design and implementation of the present features can still be analysed and reflected upon.

The choice of tools was mainly decided by availability aspects. Naninovel is the most feature-rich option, but its paid license poses a meaningful barrier for adoption. By contrast, Fungus and Dialogic are free and open-source, which allows the most access freedom. Fungus is based on Unity, which is still more widely used than Godot, and consequentely is more familiar and has more learning resources. However, Fungus is discontinued. Dialogic was ultimately chosen with Godot used as the base engine as a

[10] https://branchpy.com/

consequence. While it is in early stage, it is being actively developed, meaning its instability is temporary rather than a permanent limitation.

Beyond the included features, alternative interaction represents a significant gap across the field. All surveyed tools support only the two most common mechanisms in mainstream VNs: clickable scene elements for exploration gameplay, and text input for custom character naming. None of them natively supports more varied mechanisms, such as numeric input mentioned in the original requirements. Alternative input presents itself as a path for exploration in terms of tool development for research. However, the player input space in VNs is broad. Figure 3.9 provides a conceptual breakdown of the player input space in VNs, used to situate the development requirements.

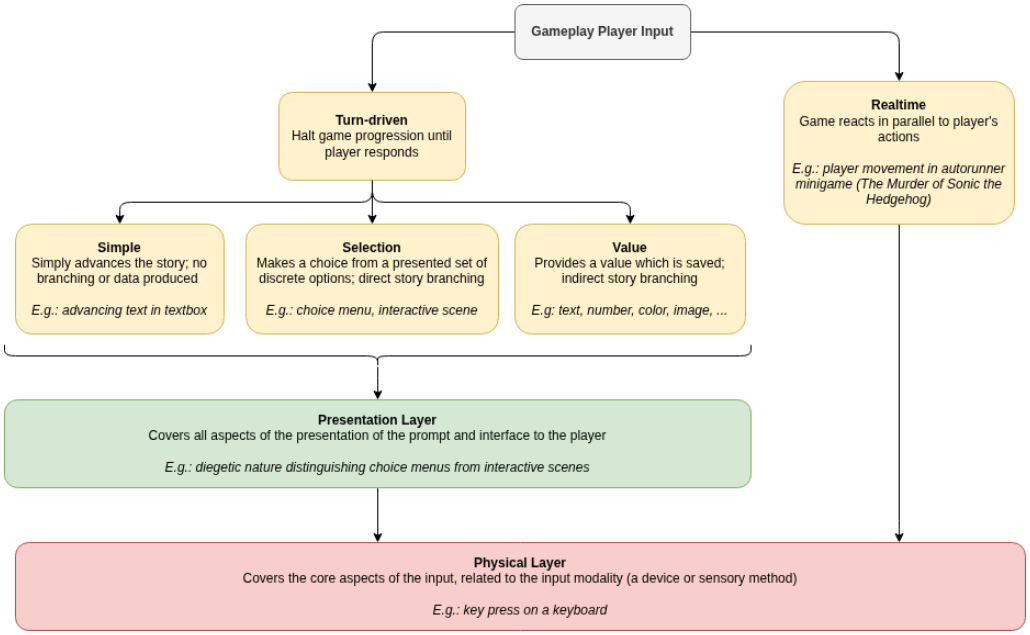


Figure 3.9: Conceptual breakdown of player input in VNs

The development scope focuses on the original requirements which include three input features: interactive scenes, text input, and numeric input. All three fall within the Turn-driven category. Within the Selection category, interactive scenes offer an alternative presentation style relative to the standard choice menu. Text and numeric input fall within the Value category. At the physical level, these requirements revolve around standard actions using the mouse and keyboard. Other value types, alternative input modalities, and further presentation variations fall outside the development scope. Dialogic already supports text input; the design and implementation of the other two is described in Chapter 4.

4 | Solution Description

This chapter first covers Godot, describing the architectural foundation shared by any project built on the engine. It then describes Dialogic, analysing how its authoring and runtime systems are built on top of this foundation. Finally, it describes the design and implementation of the two features.

4.1 Godot

Godot's architecture is built around a small number of core concepts that together define how games are structured, how behaviour is defined, and how the engine itself can be extended.

As described in Chapter 3, the scene tree and game object composition are central parts of a general-purpose game engine. Godot merges the concepts of game object and component into a single `Node` class, organised as a hierarchy of subclasses, illustrated at a high level in Figure 4.1. Composition is then achieved through the hierarchical relationship between objects in the scene tree. This means that, for example, a `Sprite2D` node (renders a 2D image) can exist as an independent game object or as a component of another object.

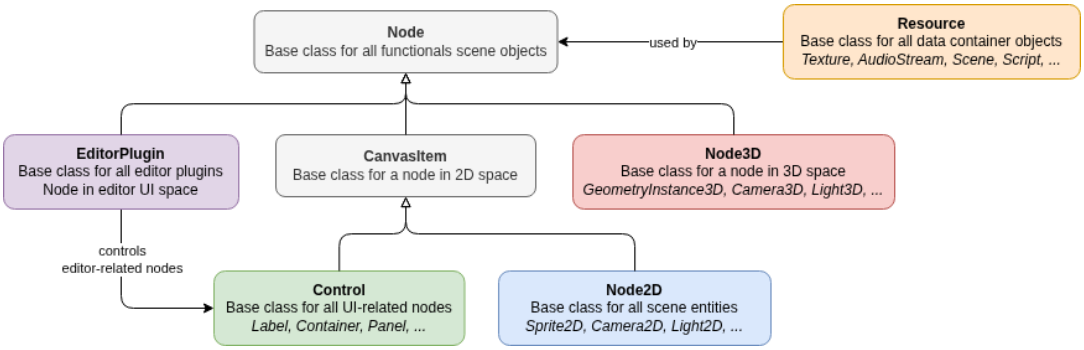


Figure 4.1: Simplified overview of the Godot's core class hierarchy

Besides nodes, the `Resource` class represents objects that serve as data containers rather than functional parts of the scene, such as textures and audio. Resources live as files within the project and are used by nodes. A particularly important and special type of resource is the `Script`, which contains code that extends or overrides a given class's functions and properties, defining a new subclass of node or resource. Scripts enable interaction with the engine's systems, such as dynamically controlling the scene tree by instantiating or removing nodes programmatically. This is especially relevant to the editor itself, since Godot's editor is technically a game running on its own underlying engine, and relies on the same scene and node architecture as any other Godot project.

Godot provides the `EditorPlugin` class for creating editor extensions. Since `EditorPlugin` inherits from `Node`, a plugin lives as a node in the editor's own scene tree, and gives the developer full access to manipulate the editor interface through the script. Any custom

UI provided by a plugin, such as panels or docks, is built as a standard scene, using `Control` nodes and the same editor tools available for building any other scene. The plugin then instantiates and adds them to the editor's scene tree.

4.2 Dialogic

Dialogic is structured around two interconnected sides: an editor side, providing the authoring tools integrated into the Godot editor, and a runtime side, managing the execution of narrative scripts during gameplay. Figure 4.2 gives a simplified overview of its main components.

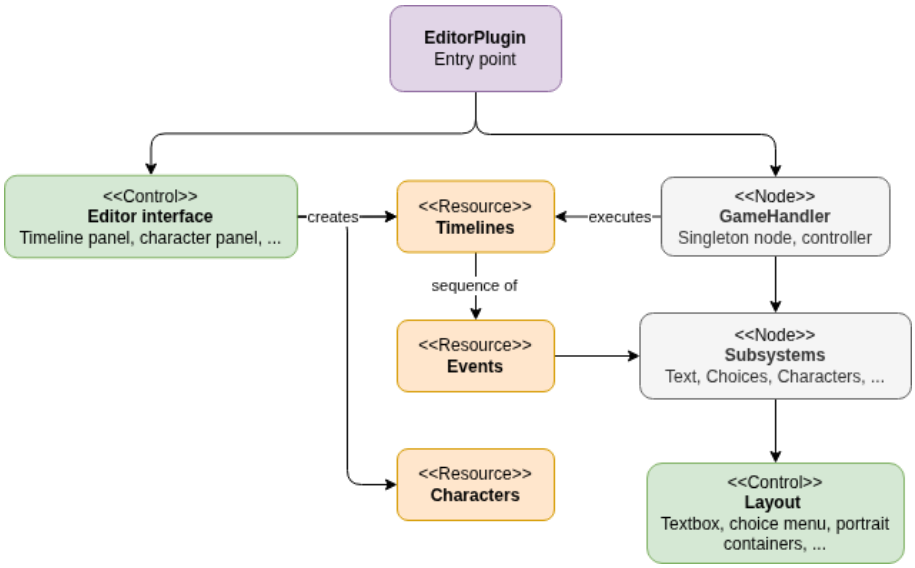


Figure 4.2: Simplified overview of Dialogic's architecture

On the runtime side, the central coordinator is the `DialogicGameHandler`, a node added to the scene tree as a globally accessible singleton by the `EditorPlugin` script. It executes the narrative scripts, which take the form of `Timeline` resources, each containing a sequence of `DialogicEvent` resources. Multiple subtypes of `DialogicEvent` exist, such as `TextEvent` for displaying dialogue text, broadly corresponding to the event types found in VN engines. Following the Command[1] design pattern, each event interacts with the relevant `DialogicSubsystem` node, present as a child of the `DialogicGameHandler` and responsible for one specific area of VN gameplay functionality. A `TextEvent`, for instance, calls into the `TextSubsystem`, which locates the textbox nodes in the scene and updates them with the new dialogue text.

All gameplay UI elements — the textbox, choice menus, portrait containers — are implemented as separate scene files, managed as layers within Dialogic's layout system. Figure 4.3 shows the default textbox's scene. Each UI component is an independently editable scene built from `Control` nodes, which developers can configure, replace, or extend using the main editor's scene-building tools.

The same principle applies to the editor side. Dialogic's authoring panel, including the timeline editor and character registry, is a standard `Control`-based scene, instantiated and managed through the `EditorPlugin` script, shown in Figure 4.4. Understanding, modifying, or extending any part of Dialogic's UI, whether gameplay or editor, requires

[1] `https://refactoring.guru/design-patterns/command`

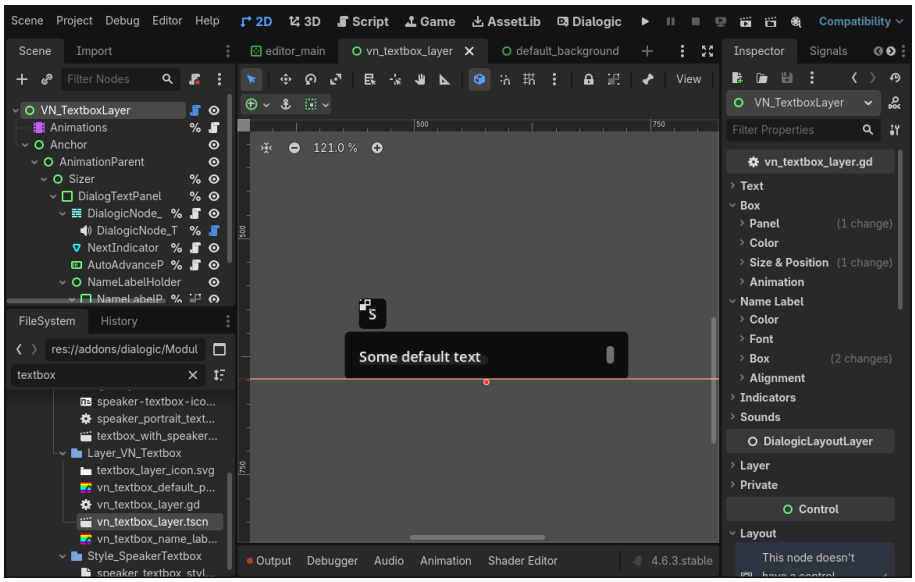


Figure 4.3: Scene of Dialogic's textbox layer

the same knowledge needed to build any Godot scene. Dialogic's extensibility is a direct consequence of Godot's scene system, rather than an entirely separate plugin framework.

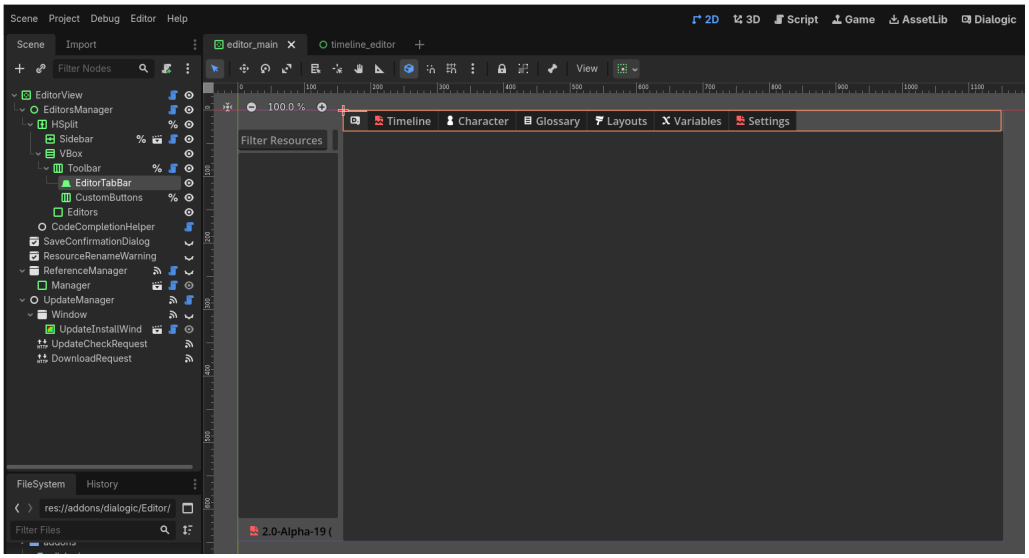


Figure 4.4: Top-level scene containing the Dialogic editor

The two core VN asset types, characters and backgrounds, are defined as resources rather than nodes. The `DialogicCharacter` resource holds a character's metadata, such as their display name and a dictionary of named portrait images. Backgrounds are plain `Texture` resources.

A particularly relevant aspect of Dialogic's design is how it communicates with systems outside itself. A further consequence of this architecture is how Dialogic communicates with systems outside itself. The `DialogicGameHandler` exposes execution control functions, and the timeline includes `Call` and `Signal` events for invoking functions in external scripts. Since Dialogic objects are nodes, they can interact with any other node in the scene tree, including nodes belonging to entirely separate gameplay systems. VN gameplay built with Dialogic can therefore be combined with non-VN gameplay without any

dedicated integration layer, simply by using the communication mechanisms Godot's scene system already provides. This is a practical advantage over VN engines, whose frameworks are built around standard VN use cases alone.

4.3 Numeric Input

Numeric Input follows the same structural pattern as Dialogic's existing Text Input feature, which works by displaying a text field to the player and storing the response in a Dialogic variable after confirmation. Reusing this pattern keeps the new feature consistent with Dialogic's existing authoring conventions, and requires no new architectural concept for the author to learn. Technically, it consists of the same three components: a timeline event, a subsystem, and a layout layer.

The timeline event, `NumericInputEvent`, defines the properties the author configures in the timeline editor: the Dialogic variable to store the result in, an optional minimum and maximum value, a step value for incrementing and decrementing, a default starting value, and a prompt text displayed to the player, as seen in Figure 4.5. The layout layer is a scene containing a spinbox and a slider, built using only Godot's built-in UI nodes, as seen in Figure 4.6. The subsystem connects the two: when the event executes, it initialises the layout layer with the event's properties and, once the player confirms a value, stores the result in the named variable. Figure 4.7 shows a diagram of the implemented classes.

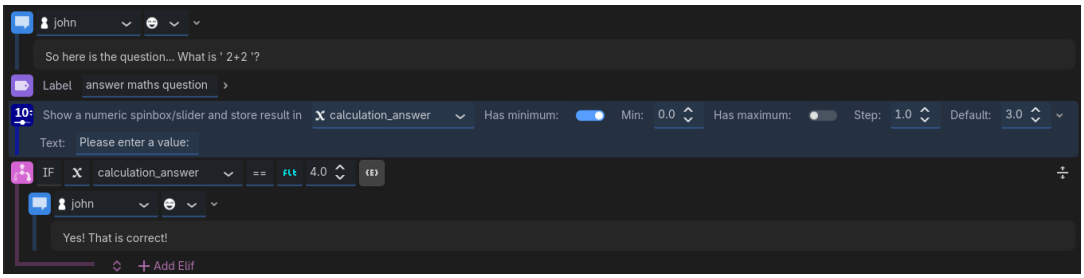


Figure 4.5: Example of NumericInput event on the timeline

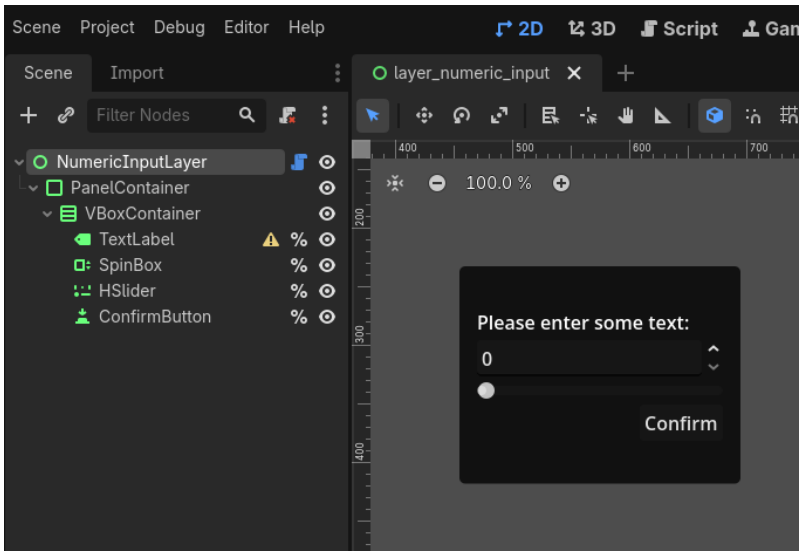


Figure 4.6: Scene of the Numeric Input layer

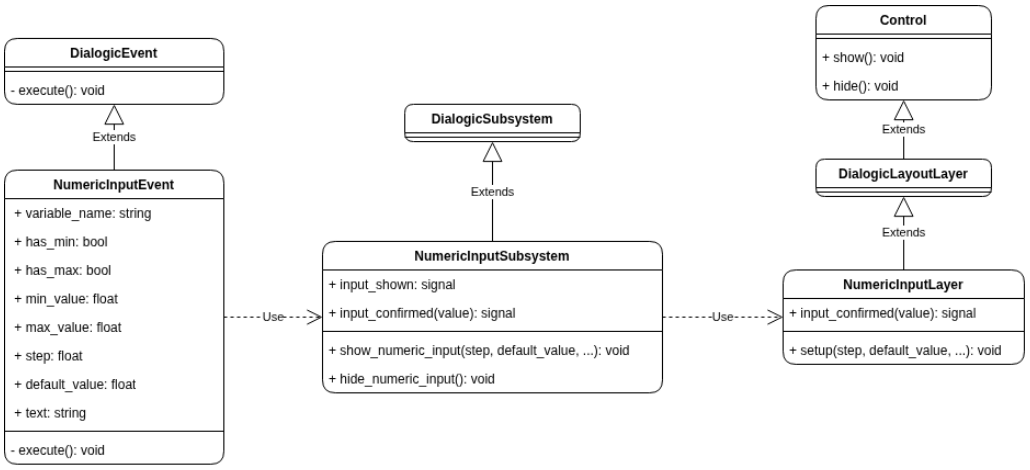


Figure 4.7: Class diagram of the numeric input feature

4.4 Interactive Scenes

Interactive scenes, also known as *hotspots*, allow the player to interact with elements placed directly within the scene, triggering story events. Unlike choice menus, which are presented as a UI overlay, hotspots are integrated into the scene itself, giving them a diegetic quality.

Two design approaches for this feature were identified across the surveyed VN tools. The first, employed by TyranoBuilder, integrates hotspots within the timeline similarly to choice menus: each hotspot is added as a node on the timeline with a configurable screen position. The second, employed by Ren'Py, Naninovel, and Fungus, keeps hotspots as a dedicated scene or UI layer that is displayed via a timeline event. This project follows the second approach, as it reuses Godot's scene system and tooling more naturally.

The feature consists of a single timeline event, `HotspotSceneEvent`, which requires the author to specify a scene resource, as seen in Figure 4.8. Unlike Numeric Input, this feature requires no dedicated subsystem, since the event only needs to instantiate and display an author-provided scene. The author creates the hotspot scene using Godot's standard scene tooling: a new scene file with a `Control` node as the root, containing one or more `Hotspot` nodes placed within the hierarchy, as seen in Figure 4.9. The `Hotspot` node is implemented by extending Godot's built-in `TextureButton` type, inheriting its interaction functionality. The texture is optional, allowing a hotspot to function as either a clickable image or an invisible clickable area. Each hotspot stores a reference to a Dialogic timeline, and automatically connects its button press signal to the start of that timeline. Figure 4.10 shows a diagram of the implemented classes.

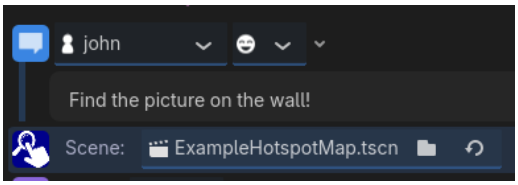


Figure 4.8: Example of HotspotScene event on the timeline

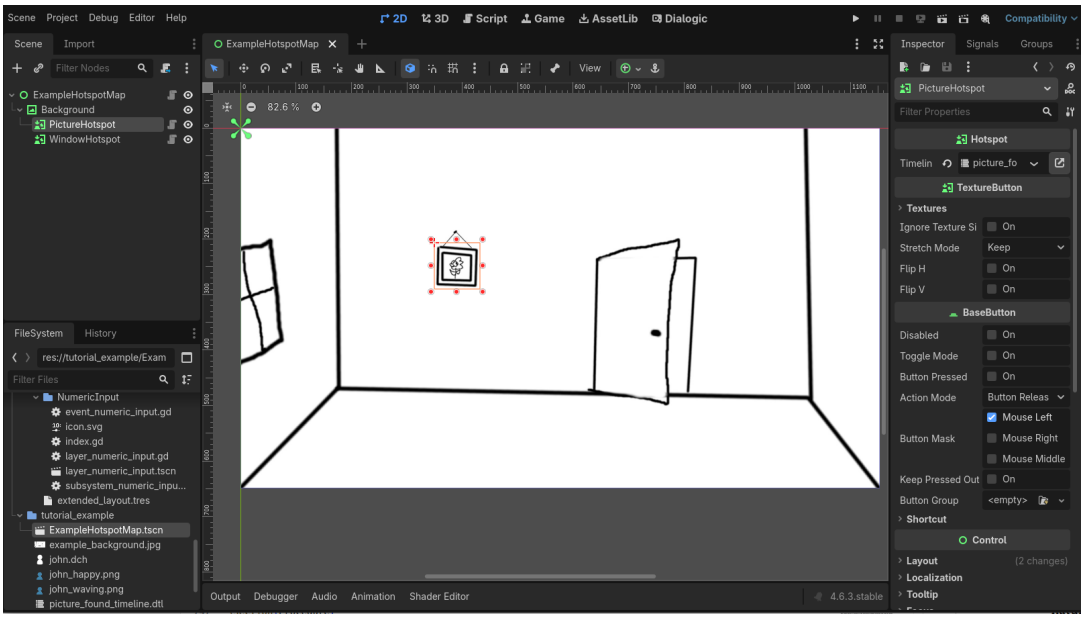


Figure 4.9: Example scene with hotspots

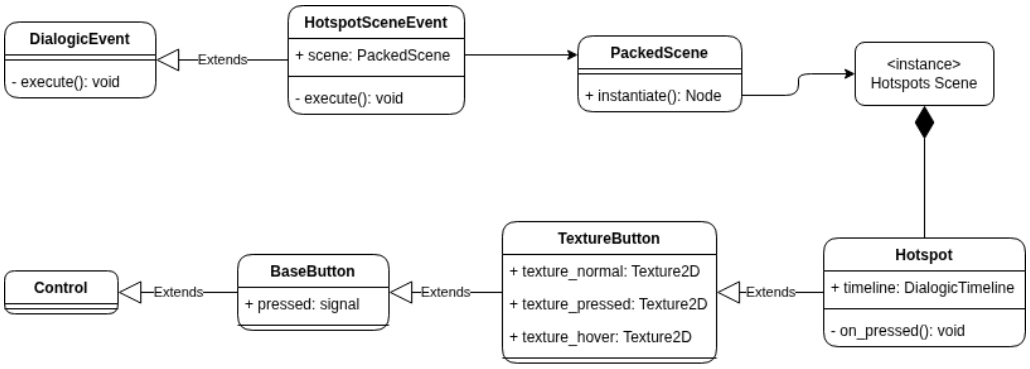


Figure 4.10: Class diagram of the hotspots feature

5 | Evaluation

5.1 Procedure

The study design is based on the evaluation method proposed and used by Green et al. [9, 8], who address the scarcity of systematic UX evaluations of Interactive Digital Narrative authoring tools. They identify a central tension in evaluating such tools: standard task-based usability testing assumes short, discrete tasks, whereas meaningful authorship is a sustained, complex process. Their proposed protocol addresses this by placing participants within an ongoing authoring task, skipping the initial setup phase, and collecting both behavioural and reflective data within a pragmatic time frame. The study is conducted in a single session consisting of multiple stages: contextualising the participant; conducting a demographic survey; a tutorial on how to use the tool; the authoring session itself; and a post-session interview.

The study reported here is a single-participant case study rather than a controlled usability study. The available participant is the researcher from the motivating case study, whose requirements directly shaped the development scope of the solution, making them a particularly well-suited evaluator of the solution's fitness for purpose. This context motivates one adaptation to Green et al.'s protocol: rather than completing a missing section of a predefined story, the participant authors a story of their own. Since they are already familiar with the content they originally intended to implement, this naturally avoids the story-setup problem that Green et al.'s partial-story approach is designed to circumvent, while also grounding the session in a more authentic authoring context.

The Ethics and Privacy Quick Scan of the Utrecht University Research Institute of Information and Computing Sciences classified this research as low-risk with no fuller ethics review or privacy assessment required.

5.1.1 Pre-session

At the beginning, the participant receives an information sheet explaining the study purpose and procedure, as well as formally asking for participation consent. They may ask any questions. Then a demographic survey is conducted, which will give grounding for the interpretation of the results. The survey includes the following questions:

1. What is your field of study?
2. What is your knowledge of Visual Novels and interactive narrative games?
3. What is your knowledge of Dialogic and Godot?
4. What is your knowledge of any other VN authoring tools?
5. What is your knowledge of any other game engines?

Finally, the participant goes through a tutorial which introduces the editor interface and demonstrates essential functionality. This tutorial takes the form of a walkthrough paired

with a video showing the creation of example content. The video and example content will remain available during the authoring session.

The first step of the tutorial is to give an overview of the Godot editor, covering the scene panel and the filesystem, followed by the Dialogic-specific panels. The focus is on naming the areas and briefly stating their purpose, rather than explaining how everything works. The next step is to demonstrate the core workflow, which includes importing assets into the engine, defining new characters, and writing a timeline with the basic events — display events for backgrounds, characters, dialogue text and choices, and flow control using labels and *jump* events — illustrated in Figure 5.1. The final step is to demonstrate the more advanced authoring features, which includes defining variables, adding conditional statements to the timeline, and using alternative input, as illustrated in Figure 5.2.

5.1.2 Authoring Session

The participant is given a hard limit of 30 minutes (matching the duration used in Green et al.'s study [8]) to create their story using the tool, along with some task guidelines covering relevant features: include non-linear pathways, an asynchronous story variation (based on something the player did earlier), and alternative player input (text, numeric, and clickable images). They are encouraged to follow the task guidelines but are given creative freedom, enabling natural behaviour to emerge during the session. They can ask questions at any time, and can finish early if satisfied.

5.1.3 Post-Session Interview

Following the authoring session, the participant takes part in a semi-structured interview. A fixed set of questions is asked in order, as follows:

1. What aspects and features of the tool felt helpful or satisfying?
2. What aspects and features of the tool felt like obstacles to your goal of implementing your vision. What was it that frustrated you?
3. If you could add or change any features to make implementing your vision easier, what would they be and why?
4. To what extent did the tool feel like a coherent VN authoring environment? What specific moments or interface elements contributed to that feeling?
5. Do you feel like your familiarity with game engines influenced your experience using the tool?

Questions 1–3 provide general feedback on the solution and overall authoring experience. Questions 4 and 5 are specific to this thesis, examining the influence of the general-purpose engine's design on the authoring experience. The observational notes may be mentioned to the participant during the interview, prompting more specific or richer responses.

5.1.4 Data Collection and Analysis

During the authoring session, the participant is encouraged to narrate their thought process. The participant's voice is recorded along with their screen for later reference. The interview is also recorded. Since this is a single-participant case study, the specific answers and incidents captured in the session are analysed, rather than performing the thematic analysis which Green et al. do across multiple participants.

5.2 Results and Discussion

The evaluation session was more time-constrained than planned, due to unforeseen personal circumstances on the participant's side, aggravated by an extended difficulty during the authoring task. As a result, only basic dialogue between two characters and a sequence of numeric inputs were authored; the intended branching portions were not reached. Corrupted audio sections in the recordings also resulted in some context being lost. The findings reported below should therefore be read as a narrow but still informative set of observations, drawn from both the authoring session and the post-session interview.

The participant works within the field of Human-Computer Interaction and Serious Games, and has strong familiarity with interactive narrative games as a medium. They reported little to no familiarity with VN authoring tools, having only encountered research-oriented tools such as Propellant. Notably, they had some experience with Unity, being familiar with its interface but not its programming side.

Question 1: Helpful and satisfying aspects

When asked what aspects or features felt helpful or satisfying, the participant highlighted two positives: 1) creating and displaying characters, together with dialogue, was described as clear and intuitive; and 2) the timeline's scrollable, vertically ordered layout was praised as easy to maintain an overview of the story with.

Regarding the second point, the participant added that they found this layout cleaner than graph-based layouts, which they had previous experience with. However, this observation is limited in an important respect: the participant did not author any branching content during the session, which is precisely the context where graph-based layouts are most advantageous. The observation is therefore presented as a tentative, domain-relevant data point worth further investigation.

Question 2: Obstacles to authoring

When asked what aspects or features felt like obstacles, the participant raised two issues: 1) no support for capturing multiple numeric inputs at once, having to structure them as sequential numeric input events; and 2) the confusion about when to use Godot scenes as backgrounds.

The background/scene confusion manifested early. The Background event presented two configuration options — a simple stanardard option for using a static image, and an advanced alternative option to use a Godot scene for complex background compositions — with no description distinguishing the two. In doubt, the participant chose the scene option, a path that quickly led to confusion and requiring directions. They intended to use a simple static image, and self-corrected after consulting the tutorial, but a noticeable portion of the session time was spent trying to navigate the Godot's scene system.

This incident illustrates how a GP engine feature can influence the VN authoring workflow without being sufficiently scoped within the interface. This relates to the design principles described by Green et al. [8], which state the relative visibility of features within the editor influences how frequently users gravitate towards them. Applied to this case, it suggests that the relative prominence of general-purpose editor affordances may lead authors away from the simpler, more appropriate options.

The numeric input constraint led to a significant disruption. The participant was informed that, with the current implementation, capturing three numeric inputs could

only be done with three sequential numeric input events in the timeline. Following this, the participant produced an unusual construct involving Choice events whose goal they could not articulate, and continued unsuccessful experimentation until intervention was required. The goal only became clear upon arriving at the final structure: alternating Choice and Numeric Input events, repurposing the Choice event as a gate button outside its intended use for branching. As the participant's thinking during the exploration phase was never made clear, the following is therefore an informed speculation rather than a supported conclusion: being forced to work around the first limitation, the participant may have converged on a vision that also relied on unsupported features (gate button), repurposing choice menus as the closest available option, and consequently encountering friction and losing orientation.

Question 3: Desired features and improvements

When asked what features they would add, the participant restated their original vision: a single event with multiple inputs, displayed simultaneously within one interface. The current implementation only supports sequential single-input events. They also proposed a design improvement regarding the Background event: separate the "background" and "scene" configurations into two distinct event types, with more explicit naming and descriptions.

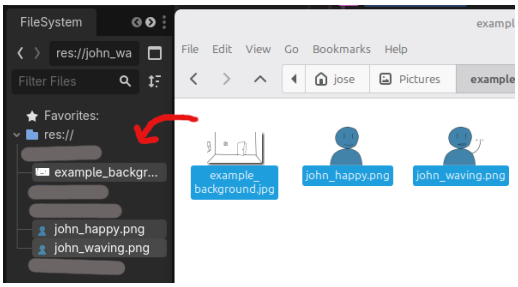
Question 4: Perceived coherence of the authoring environment

When asked whether the tool felt like a coherent story-authoring environment, the participant reported that building a timeline felt like the natural starting point of creating a story. Asked whether any specific moments or interface elements broke that feeling, they identified the background/scene confusion as only a minor disruption.

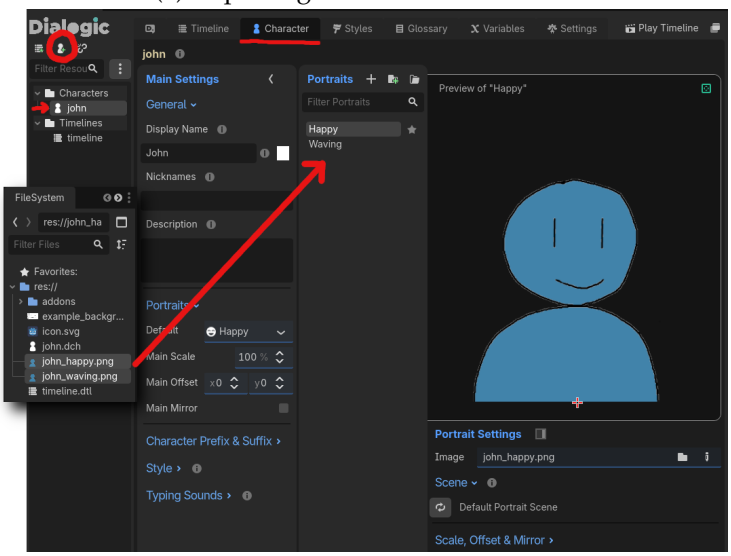
This is the most direct evidence available for RQ2. While the general-purpose engine's influence on the authoring experience surfaced at that moment, it remained localised, suggesting that it is present at specific friction points, but not disruptive to the overall authoring experience.

Question 5: Influence of prior engine familiarity

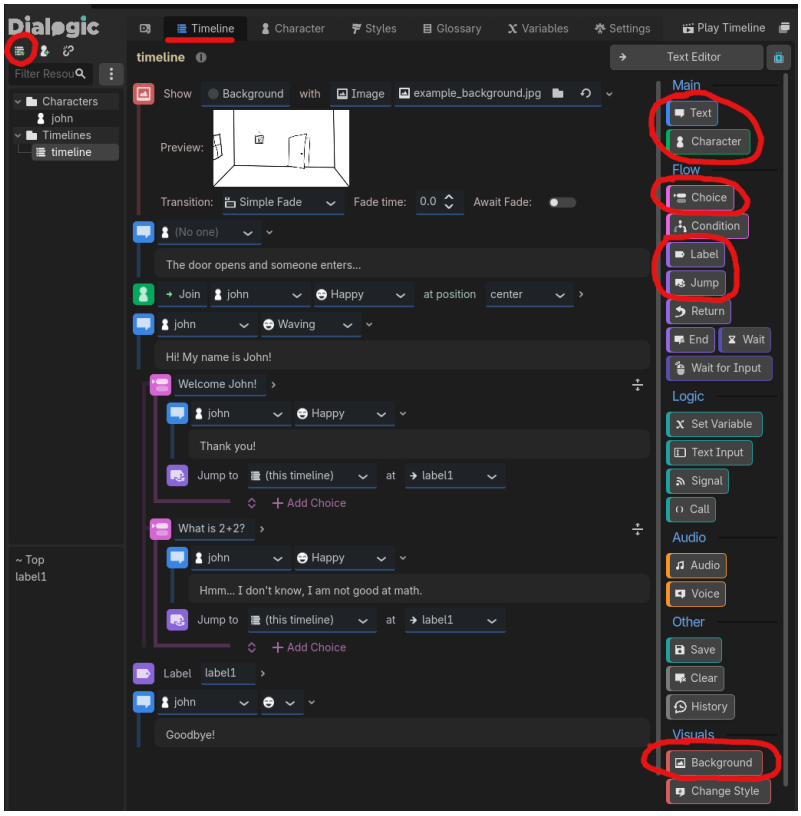
When asked whether their familiarity with game engines had influenced their experience, the participant confirmed that their predisposition towards scene-based thinking in the background event situation came from their prior experience with Unity. This directly corroborates the need for making the disctinction between both options clearer, and suggests that prior engine experience is a relevant factor in how authors navigate the interface.



(a) Importing asset files into Godot

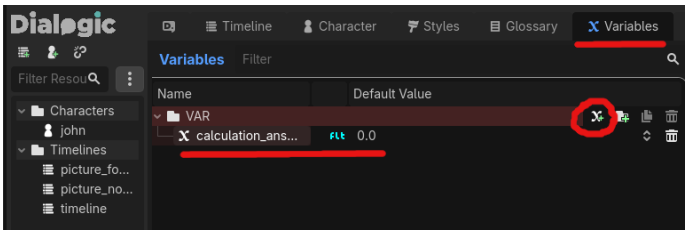


(b) Registering a new character in Dialogic

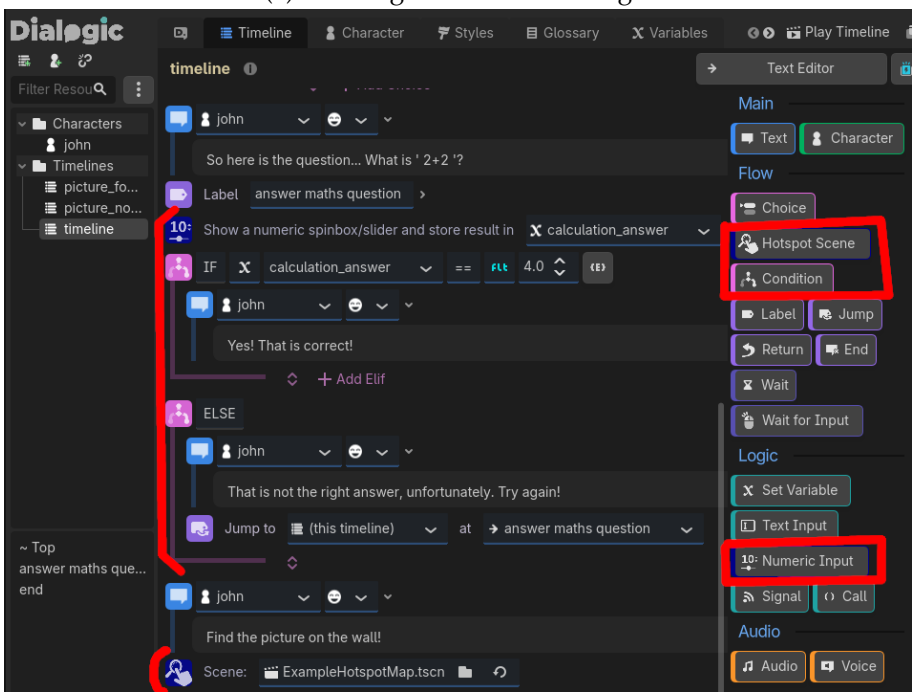


(c) Creating a timeline which includes basic events

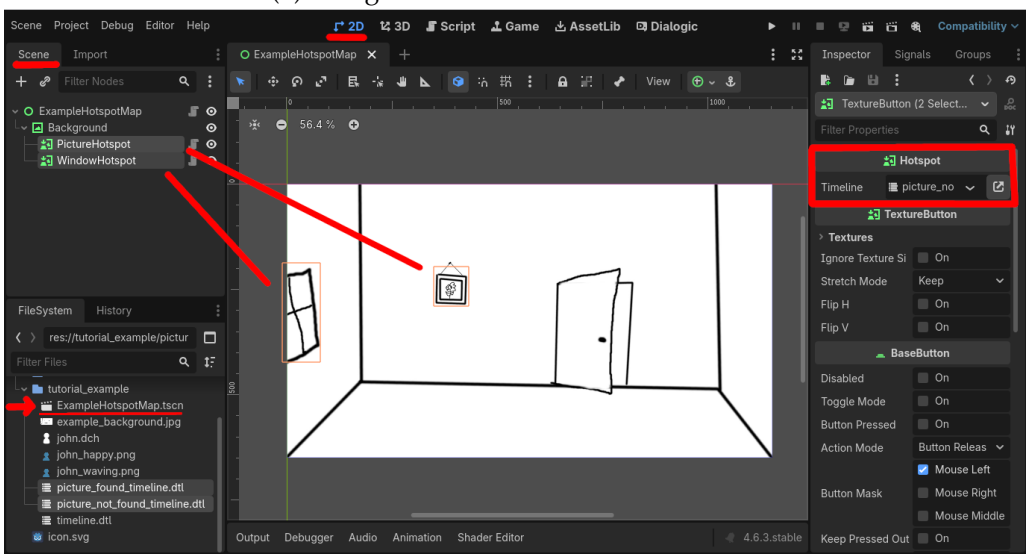
Figure 5.1: Tutorial steps demonstrating the core workflow of Dialogic



(a) Defining variables in Dialogic



(b) Using advanced events in the timeline



(c) Creating a scene with hotspots

Figure 5.2: Tutorial steps demonstrating the workflow using the more advanced authoring features

6 | Conclusion

This thesis examines whether general-purpose (GP) game engines are a suitable foundation for sustainable VN authoring tools in research contexts, motivated by the Propellant case: a purpose-built editor too costly to adapt against new requirements. The thesis combines a comparative analysis of the VN tooling landscape, a proof-of-concept solution built on Godot and Dialogic, and a usability evaluation of that extension.

The comparative analysis shows that mature GP-engine extensions, such as Naninovel, already match dedicated VN engines in feature coverage while also inheriting the flexibility and tooling of their base engine. Dialogic and Fungus still trail on secondary features, a gap tied to their development status rather than to the GP-extension approach itself; all three nonetheless provide the essential features needed for VN development. The analysis also identifies alternative player input as a gap shared across tools, which sets the scope for the proof-of-concept.

For RQ1, the architectural analysis of Dialogic shows that Godot's scene-based design applies uniformly throughout editor tooling and gameplay features, as they are all built from ordinary nodes and scenes, with no dedicated plugin framework needed. Implementing numeric input and interactive scenes confirms this, as both follow the same conventions, supporting Godot as a viable, extensible foundation for VN tooling.

For RQ2, the single-participant evaluation gives narrow but suggestive evidence. The participant's prior Unity experience briefly misdirects them toward the scene tree when configuring a simple background, showing how general-purpose concepts embedded in the interface can shape an author's mental model. Despite this, the participant still perceives the tool as a coherent VN authoring environment, with the timeline as the obvious entry point.

These findings give a cautiously positive answer to the research question motivating this thesis: Godot's architecture lowers the cost of adding new features, while its effect on the authoring experience is real but context-dependent rather than a systematic barrier. Given the limited scope of both the proof-of-concept and the evaluation, this answer remains preliminary.

The proof-of-concept implements only two player input features. Other angles, such as different input modalities or value types, remain unexplored, and would give empirical grounding to the extensibility claims made here analytically. These angles can naturally be explored as the proposed solution is used in future projects with new requirements.

The evaluation is a single-participant case study, cut shorter than planned and affected by partial audio loss. Its findings on RQ2 are illustrative, not generalisable, and are shaped by the participant's own familiarity with Unity. Larger studies, with a bigger and more varied participant pool, are needed to characterise the general-purpose environment's influence more reliably.

The landscape analysis compares tools on features alone. It does not capture the UX aspects that only hands-on testing can reveal. Little data exists for VN tools in general,

echoing Green et al.'s [9] call for more comprehensive user evaluations of both modern and older tools.

Finally, this thesis is scoped to Godot and Dialogic specifically. Extending the same development and evaluation approach to other engines and extensions, such as Unity, Naninovel, and Fungus, is needed to properly answer the research questions.

Bibliography

[1] Janelynn Camingue, Elin Carstensdottir, and Edward F. Melcer. "What is a Visual Novel?" In: *Proceedings of the ACM on Human-Computer Interaction* 5.CHI PLAY (Oct. 2021), pp. 1–18. DOI: 10.1145/3474712.

[2] Janelynn Camingue, Edward F. Melcer, and Elin Carstensdottir. "A (visual) novel route to learning: A taxonomy of teaching strategies in visual novels". In: *International Conference on the Foundations of Digital Games* (Sept. 2020), pp. 1–13. DOI: 10.1145/3402942.3403004.

[3] Daniel S. Christensen, Mette Jakobsen, and Martin Kraus. "The engagement effect of players' agency over their characters' motivation". In: *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering* (2018), pp. 184–193. DOI: 10.1007/978-3-319-76908-0_18.

[4] Mia Consalvo and Dan Staines. "Reading ren'py: Game engine affordances and design possibilities". In: *Games and Culture* 16.6 (Nov. 2020), pp. 762–778. DOI: 10.1177/1555412020973823.

[5] Ignacio X. Domínguez et al. "The mimesis effect". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (May 2016), pp. 3438–3449. DOI: 10.1145/2858036.2858141.

[6] Christopher Ferraris and Charlie Hargood. "RichCast - a voice-driven interactive digital narrative authoring system". In: *Lecture Notes in Computer Science* (2022), pp. 414–423. DOI: 10.1007/978-3-031-22298-6_26.

[7] Katelyn M. Grasse et al. "Improving undergraduate attitudes towards responsible conduct of research through an interactive storytelling game". In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (May 2021), pp. 1–8. DOI: 10.1145/3411763.3451722.

[8] Daniel Green, Charlie Hargood, and Fred Charles. "Use of Tools: UX Principles for Interactive Narrative Authoring Tools". In: *Journal on Computing and Cultural Heritage* 14.3 (July 2021), pp. 1–25. DOI: 10.1145/3458769.

[9] Charlie Hargood and Daniel Green. "The Authoring Tool Evaluation Problem". In: *Human–Computer Interaction Series* (2022), pp. 303–320. DOI: 10.1007/978-3-031-05214-9_19.

[10] Julian Holfeld. *On the relevance of the Godot Engine in the indie game development industry*. 2024. arXiv: 2401.01909 [cs.OH]. URL: https://arxiv.org/abs/2401.01909.

[11] Mette Jakobsen, Daniel Svejstrup Christensen, and Luis Emilio Bruni. "How knowledge of the player character's alignment affect decision making in an interactive narrative". In: *Lecture Notes in Computer Science* (2017), pp. 193–205. DOI: 10.1007/978-3-319-71027-3_16.

[12] Shi Johnson-Bey et al. "Building visual novels with social simulation and storylets". In: *Lecture Notes in Computer Science* (Dec. 2024), pp. 145–161. DOI: 10.1007/978-3-031-78450-7_9.

[13] Samuel Lynch et al. "M22 - A modern visual novel framework". In: *Proceedings of the 8th International Workshop on Narrative and Hypertext* (Sept. 2019), pp. 9–13. DOI: 10.1145/3345511.3349284.

[14] Marcus Toftedahl and Henrik Engström. "A taxonomy of game engines and the tools that drive the industry". In: *DiGRA Digital Library* (Jan. 2019). DOI: 10.26503/dl.v2019i1.1077.