МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ІНТЕРАКТИВНОЇ ВІЗУАЛЬНОЇ НОВЕЛИ З ВИКОРИСТАННЯМ REACT ТА ELECTRON»**

Виконав: студент 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
О.В. Коростильов
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, PhD Чопорова О.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри фундаментальної та прикладної математики, доцент, к.ф.-м.н. Панасенко Є.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Коростильову Олегу Владиславовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка інтерактивної візуальної новели з використанням React та Electron

керівник роботи Чопорова Оксана Володимирівна, PhD
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 04.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз предметної області та існуючих рішень.
2. Проєктування архітектури та компонентів системи.
3. Програмна реалізація та впровадження.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	14.03.2025	
2.	Збір вихідних даних.	14.04.2025	
3.	Обробка методичних та теоретичних	12.05.2025	
	джерел.		
4.	Розробка першого та другого розділу.	29.08.2025	
5.	Розробка третього розділу.	24.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	18.12.2025	

Студент _____ О.В. Коростильов
(підпис) (ініціали та прізвище)

Керівник роботи _____ О.В. Чопорова
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка інтерактивної візуальної новели з використанням React та Electron»: 70 с., 44 рис., 23 джерела.

ВІЗУАЛЬНА НОВЕЛА, ДІАЛОГОВА СИСТЕМА, ІГРОВИЙ РУШІЙ, КРОСПЛАТФОРМЕНА РОЗРОБКА, УПРАВЛІННЯ СТАНОМ, ELECTRON, REACT.

Об’єкт дослідження – процес розробки кросплатформенних інтерактивних візуальних новел з використанням сучасних вебтехнологій.

Предмет дослідження – методи та інструменти створення візуальних новел на базі React та Electron, включаючи архітектурні рішення, системи управління діалогами та станом гри.

Мета роботи: розробити кросплатформенну інтерактивну візуальну новелу з використанням React та Electron, що забезпечить сучасний користувацький інтерфейс та ефективне управління складними діалоговими системами і розгалуженням сюжету.

Методи дослідження – аналіз існуючих рішень та технологій, моделювання архітектури системи, методи програмної реалізації, експериментальне тестування та емпіричний аналіз результатів.

Виконано аналіз предметної області та інструментів розробки візуальних новел. Обґрунтовано доцільність використання стеку React та Electron. Спроєктовано модульну архітектуру додатку, визначено функціональні вимоги та структуру системи управління станом гри.

Розроблено кросплатформенний десктопний застосунок візуальної новели. Системну частину та управління життєвим циклом програми реалізовано засобами фреймворку Electron, а інтерактивний інтерфейс та логіку відображення – засобами бібліотеки React.

SUMMARY

Master’s qualifying paper “Development of an Interactive Visual Novel Using React and Electron”: 70 pages, 44 figures, 23 references.

CROSS-PLATFORM DEVELOPMENT, DIALOGUE SYSTEM, ELECTRON, GAME ENGINE, REACT, STATE MANAGEMENT, VISUAL NOVEL.

The object of study is the process of developing cross-platform interactive visual novels using modern web technologies.

Subject of study: methods and tools for creating visual novels based on React and Electron, including architectural solutions, dialogue and game state management systems.

The aim of the study is to develop a cross-platform interactive visual novel using React and Electron to provide a modern user interface and effective management of complex dialogue systems and plot branching.

The methods of research are analysis of existing solutions and technologies, system architecture modeling, software implementation methods, experimental testing, and empirical analysis of results.

The analysis of the subject area and visual novel development tools was performed. The feasibility of using the React and Electron stack was substantiated. The modular application architecture was designed, and functional requirements along with the structure of the game state management system were determined.

A cross-platform desktop visual novel application was developed. The system part and application lifecycle management were implemented using the Electron framework, and the interactive interface and display logic were implemented using the React library.

ЗМІСТ

Завдання на кваліфікаційну роботу ……..…..2
Реферат 4
Summary 5
Вступ..... 8
1 Аналіз предметної області та існуючих рішень..... 10
1.1 Огляд жанру візуальних новел 10
1.2 Аналіз існуючих рішень та технологій..... 11
1.3 Обґрунтування вибору технологічного стеку 12
1.3.1 React..... 12
1.3.2 Electron 13
1.3.3 TypeScript..... 13
1.3.4 Zustand..... 14
1.3.5 TailwindCSS 14
1.4 Функціональні вимоги до програмного продукту..... 15
1.5 Нефункціональні вимоги до програмного продукту..... 17
1.6 Висновки до розділу 1 19
2 Проєктування архітектури та компонентів системи 20
2.1 Загальна архітектура додатку 20
2.1.1 Шарова структура додатку 20
2.1.2 Структура та організація бази даних 22
2.2 Проєктування DSL для скрипту візуальної новели..... 23
2.2.1 Ідентифікація та класифікація сцен 23
2.2.2 Система ефектів 26
2.2.3 Система умов та логічних операторів..... 27
2.2.4 Макроси 28
2.3 Рушій виконання сцен та управління переходами 29
2.4 Система збереження та завантаження прогресу..... 31

2.5 Система візуалізації персонажів 32
2.5.1 Менеджер спрайтів персонажів 32
2.5.2 Структура даних та композиція спрайтів 33
2.6 Проєктування архітектури клієнтського додатку (Renderer Process) 35
2.6.1 Компонентна структура та глобальний стан 35
2.6.2 Потік керування та цикл оновлення інтерфейсу 37
2.7 Висновки до розділу 2 39
3 Програмна реалізація та впровадження 40
3.1 Конфігурація середовища та структура проєкту 40
3.2 Реалізація системного процесу та механізмів IPC 42
3.2.1 Ініціалізація головного процесу та конфігурація IPC 42
3.2.2 Менеджер конфігурації спрайтів 43
3.2.3 Менеджер збереження прогресу 44
3.2.4 Реалізація шару доступу до даних сценарію 46
3.3 Програмна реалізація рушія 47
3.3.1 Реалізація контролера сценарію 47
3.3.2 Організація глобального стану 52
3.3.3 Система обробки умов та ефектів 54
3.3.4 Інтеграція рушія з реактивним інтерфейсом 56
3.4 Реалізація клієнтської частини додатку 57
3.4.1 Структура та функціональні елементи інтерфейсу користувача 57
3.4.2 Реалізація системи композитного рендерингу спрайтів 62
3.5 Автоматизація процесів розробки та розгортання (CI/CD) 63
3.6 Висновки до розділу 3 65
Висновки 67
Перелік посилань 69

ВСТУП

Інтерактивні візуальні новели займають важливе місце в індустрії відеоігор завдяки своїй здатності поєднувати глибокі наративи з інтерактивністю, де кожне рішення гравця може суттєво вплинути на розвиток сюжету. Це дозволяє створювати унікальні й персоналізовані досвіди для користувачів, що робить жанр надзвичайно популярним серед різних аудиторій. Візуальні новели вже давно стали значущою частиною глобальної ігрової індустрії, зокрема в Японії та на Західному ринку, де вони постійно здобувають популярність завдяки своїм інноваційним підходам до оповідання.

Традиційно для створення візуальних новел використовуються спеціалізовані движки, такі як Ren’Py, TyranoBuilder або Visual Novel Maker. Хоча ці інструменти добре підходять для створення класичних візуальних новел, вони мають певні обмеження щодо гнучкості інтерфейсу, інтеграції з сучасними вебтехнологіями та можливостей кастомізації. З іншого боку, сучасні вебтехнології, зокрема React, надають широкі можливості для створення динамічних та інтерактивних інтерфейсів, а фреймворк Electron дозволяє створювати кросплатформенні десктопні додатки на основі вебтехнологій.

Використання React та Electron для розробки візуальних новел відкриває нові можливості: компонентний підхід React забезпечує модульність та повторне використання коду, сучасні інструменти стану дозволяють ефективно управляти складними діалоговими системами та розгалуженням сюжету, а Electron надає доступ до файлової системи та можливість створення справжніх десктопних додатків, які працюють на Windows, macOS та Linux без додаткової адаптації. Це особливо актуально в контексті зростаючого попиту на кросплатформенні рішення та інструменти, що дозволяють розробникам швидко створювати та підтримувати складні інтерактивні додатки.

Розробка візуальної новели з використанням React та Electron демонструє можливості застосування сучасних вебтехнологій для створення ігрових продуктів та дозволяє дослідити переваги та виклики цього підходу порівняно з традиційними інструментами розробки візуальних новел.

Практичне значення одержаних результатів. Результати роботи мають практичне значення для розробників, які розглядають можливість використання сучасних вебтехнологій для створення ігрових продуктів. Розроблений додаток демонструє життєздатність підходу з використанням React та Electron для створення візуальних новел та може слугувати основою для подібних проєктів.

Архітектурні рішення, застосовані в роботі, можуть бути використані для створення інших типів інтерактивних додатків, що потребують складного управління станом та розгалуженою логікою. Досвід інтеграції React з Electron може бути корисним для розробників, які планують створювати кросплатформенні десктопні додатки на основі вебтехнологій.

Реалізована система управління діалогами та сюжетом може бути адаптована для використання в освітніх додатках, інтерактивних презентаціях або інших проєктах, що потребують складної логіки взаємодії з користувачем.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Огляд жанру візуальних новел

Візуальні новели – це жанр інтерактивних комп’ютерних ігор, що поєднує елементи літературної оповіді та графічного подання сюжету [1]. Такі ігри зазвичай використовують статичні або анімовані зображення персонажів, фонові ілюстрації та текстові діалоги, які визначають перебіг історії. Центральним елементом жанру є розповідь, а гравець взаємодіє з сюжетом переважно через читання тексту та вибір варіантів дій [2].

Жанр сформувався в Японії в середині 1980-х років та швидко здобув популярність серед молодіжної аудиторії. Візуальні новели дозволяють зануритися у глибокі та емоційно наповнені історії, у яких гравець часто впливає на розвиток подій через систему виборів. Важливу роль відіграє взаємодія з персонажами, їхній психологічний та сюжетний розвиток, а також відображення емоцій за допомогою змін спрайтів та художніх сцен.

У межах жанру існують як лінійні історії (так звані kinetic novels), так і твори з багатогілковим наративом, де вибори користувача визначають подальшу послідовність подій і фінал гри [3]. Сучасні візуальні новели активно використовують розгалужені діалогові структури, мультифінальні шляхи, а також графічні й звукові ефекти для підсилення атмосфери.

З розвитком інтернет-платформ та появою спеціалізованих інструментів, таких як Ren’Py, жанр отримав широку міжнародну популярність [4]. Це зробило його доступним не лише для великих студій, але й для незалежних розробників, які можуть створювати власні інтерактивні історії.

Таким чином, візуальні новели посідають помітне місце в індустрії відеоігор, оскільки поєднують у собі літературний підхід до розповіді та інтерактивність, дозволяючи реалізовувати складні, варіативні та емоційно виразні сюжети.

1.2 Аналіз існуючих рішень та технологій

Розробка візуальних новел може здійснюватися на основі різних технологічних платформ, які відрізняються функціональністю, гнучкістю, продуктивністю та складністю освоєння. Сучасний ринок пропонує широкий спектр інструментів, серед яких найпоширенішими є Ren'Py, рушії на базі Unity, а також різноманітні вебрішення, що використовують сучасні фронтенд-технології. Нижче подано аналіз основних можливостей цих платформ, їхні переваги та обмеження.

Ren'Py. Ren'Py – це спеціалізований рушій, розроблений саме для створення візуальних новел та інтерактивних історій. Він базується на Python і використовує власну сценарну мову, що робить його простим для освоєння.

Переваги [6]:

– низький поріг входу, можливість створення проєкту без знання глибокого програмування;
– наявність великої спільноти та кількості навчальних матеріалів;
– підтримка різних платформ (Windows, Linux, macOS, Android);
– вбудована система збережень, історії тексту, автоперемотування та інших стандартних VN-механік.

Недоліки:

– обмежена гнучкість для складної логіки та нестандартних інтерфейсів;
– застарілий інтерфейс та класичний підхід до UI-компонентів;
– складність інтеграції сучасних технологій (онлайн-функцій, анімацій, нестандартних UI-рішень).

Unity. Unity є універсальним 2D/3D рушієм, який активно використовується для створення різних типів ігор, у тому числі й візуальних новел через спеціалізовані фреймворки та плагіни [5].

Переваги:

– високий рівень графічних можливостей (анімації, шейдери, 3D-ефекти);

– велика екосистема плагінів та магазин контенту;
– гнучкість у побудові користувацьких інтерфейсів;
– широкий спектр платформ: PC, консолі, мобільні пристрої, WebGL.

Недоліки:

– вищий поріг входу та більша складність у розробці;
– потреба в оптимізації, особливо для десктопних проєктів;
– більша вага фінального застосунку порівняно з Ren'Py чи вебрішеннями.

Вебрішення (HTML/CSS/JS, React, Vue тощо). Вебзастосунки є гнучким підходом до створення візуальних новел, особливо коли потрібен сучасний інтерфейс або інтеграція з інтернет-сервісами. Проєкти можуть працювати як у браузері, так і у форматі desktop-додатків через Electron.

Переваги:

– повна свобода у дизайні інтерфейсу та логіці;
– можливість використання сучасних вебінструментів (аналітика, хмарні сервіси, API);
– кросплатформність через браузер або Electron;
– швидке оновлення контенту без необхідності перевипуску застосунку.

Недоліки:

– потреба в більшому обсязі розробки «з нуля»;
– складніша система збережень порівняно з готовими рушіями;
– вища відповідальність за оптимізацію продуктивності.

1.3 Обґрунтування вибору технологічного стеку

1.3.1 React

React – це популярна бібліотека для побудови інтерфейсів користувача, що дозволяє створювати динамічні та реактивні компоненти [7].

Основні причини вибору React для проєкту:

– компонентна архітектура, що забезпечує легку масштабованість та повторне використання коду;
– ефективне управління станом для реалізації діалогів та системи виборів;
– велика екосистема бібліотек для анімацій, роутінгу та управління станом;
– можливість кросплатформеної інтеграції через веб або Electron.

1.3.2 Electron

Electron – це фреймворк для створення кросплатформених десктопних застосунків на основі вебтехнологій (HTML, CSS, JS) [8]. Причини його використання у проєкті:

– кросплатформене середовище для створення desktop-додатків;
– прямий доступ до файлової системи для реалізації збережень і завантаження прогресу;
– використання сучасних вебтехнологій (HTML, CSS, JS, React) без обмежень;
– простота пакування та розповсюдження готового застосунку.

1.3.3 TypeScript

TypeScript – це надбудова над JavaScript, яка додає статичну типізацію та покращує передбачуваність роботи коду [9]. Причини використання TypeScript у проєкті:

– строгий контроль типів, що зменшує ймовірність помилок під час розробки;

– підвищення надійності та масштабованості коду, особливо в складних логічних системах;
– краща читабельність і підтримуваність проєкту;
– сумісність із сучасними фреймворками, зокрема React та інструментами екосистеми JavaScript.

1.3.4 Zustand

Zustand – це легкий та продуктивний менеджер стану, який використовується для керування логікою та даними в інтерфейсі [11]. Причини його застосування:

– простий та зрозумілий API, що дозволяє швидко організувати стан додатку;
– висока продуктивність завдяки відсутності надмірних перерендерів;
– зручність у реалізації системи діалогів, виборів та прогресу користувача;
– мінімальний обсяг коду та незалежність від складної конфігурації, властивої іншим менеджерам стану.

1.3.5 TailwindCSS

TailwindCSS – це утилітний CSS-фреймворк для швидкого та гнучкого створення інтерфейсів [10]. Причини вибору TailwindCSS:

– можливість швидкого стилізування елементів без написання додаткових CSS-файлів;
– зручний набір утилітних класів, що спрощує створення адаптивного та сучасного UI;
– легкість у підтримці та модифікації стилів у процесі розробки;

– хороший баланс між продуктивністю та гнучкістю завдяки оптимізації стилів у фінальній збірці.

1.4 Функціональні вимоги до програмного продукту

Функціональні вимоги визначають поведінку системи та перелік функцій, які програмний продукт повинен виконувати для задоволення потреб користувача. Вони охоплюють логіку взаємодії гравця з інтерфейсом, правила обробки сценарію та управління даними.

Структурну класифікацію основних функціональних вимог зображено на рисунку 1.1.

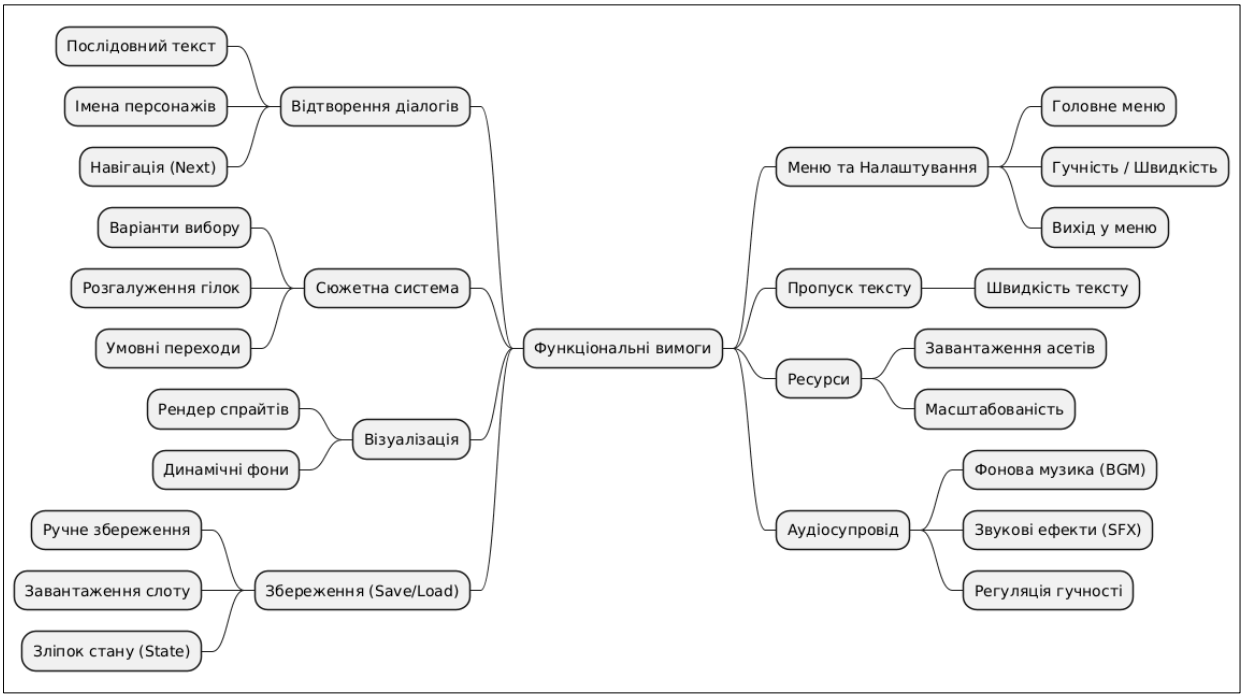


Рисунок 1.1 – Класифікація функціональних вимог до програмного продукту

Детальний опис вимог за категоріями наведено нижче.

Відтворення діалогів:

– система повинна забезпечувати послідовне відображення тексту відповідно до сценарію;

- система повинна відображати ім’я активного персонажа, якщо це передбачено сценою;
- користувач повинен мати можливість переходити до наступної репліки натисканням кнопки або кліком.

Вибір дій та розгалуження сюжету:

- система повинна дозволяти користувачу обирати один із кількох варіантів дій;
- після вибору система повинна переходити до відповідної сюжетної гілки;
- система повинна підтримувати умовні переходи залежно від попередніх рішень.

Відображення персонажів та сцен:

- система повинна відображати спрайти персонажів у визначених позиціях та станах;
- система повинна змінювати фонові зображення згідно зі сценарієм.

Збереження та завантаження прогресу:

- користувач повинен мати можливість створювати ручні збереження;
- система повинна дозволяти завантаження будь-якого створеного збереження;
- система повинна зберігати поточний сюжетний стан, вибори та службові дані.

Пропуск тексту: користувач повинен мати можливість налаштовувати швидкість відтворення тексту.

Меню та навігація:

- система повинна містити головне меню з можливістю запуску нової гри та продовження збереженої;
- користувач повинен мати доступ до меню налаштувань (гучність, швидкість тексту);
- система повинна забезпечувати вихід до головного меню з активної сцени.

Робота з ресурсами:

– система повинна завантажувати графічні ресурси (фони, спрайти) під час запуску або за потреби;
– система повинна дозволяти розширення списку ресурсів без зміни основної логіки.

Аудіосупровід:

– система повинна підтримувати відтворення фонової музики (BGM), що відповідає сцені;
– система повинна підтримувати звукові ефекти (SFX);
– користувач повинен мати можливість регулювати гучність музики та звукових ефектів у меню налаштувань.

1.5 Нефункціональні вимоги до програмного продукту

Для забезпечення високої якості кінцевого рішення та комфортної взаємодії користувача з системою було визначено ряд нефункціональних вимог. Вони регламентують експлуатаційні характеристики програмного продукту, такі як швидкодія, стабільність та зручність експлуатації.

Структурну класифікацію основних нефункціональних вимог зображено на рисунку 1.2.

Детальний опис вимог за категоріями наведено нижче.

Продуктивність:

– система повинна забезпечувати плавне відтворення діалогів та переходів між сценами;
– завантаження ресурсів (фонів, спрайтів) повинно відбуватися без помітних затримок для користувача;
– інтерфейс повинен реагувати на дії користувача протягом не більше ніж 100 мс.

Надійність: застосунок повинен коректно відновлювати роботу після перезапуску на основі останнього збереженого прогресу.

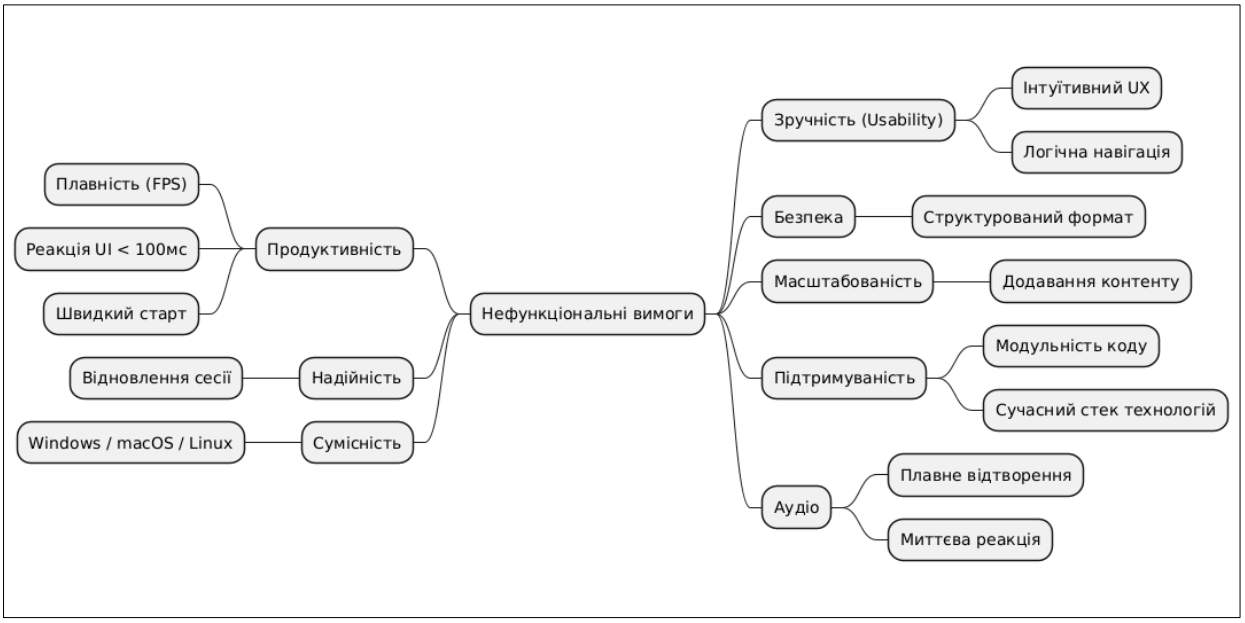


Рисунок 1.2 – Класифікація нефункціональних вимог до програмного продукту

Зручність використання (Usability):

– інтерфейс повинен бути інтуїтивно зрозумілим для користувача без необхідності вивчення документації;
– елементи управління (кнопки, меню, вибори) повинні бути доступними та логічно розташованими.

Сумісність: застосунок повинен коректно працювати на Windows, Linux та macOS.

Безпека: збереження повинні зберігатися у структурованому форматі.

Масштабованість: архітектура повинна дозволяти розширення проєкту (нові сцени, персонажі, механіки) без значних змін в існуючому коді.

Підтримуваність:

– кодова база повинна бути структурованою та розділеною на модулі для спрощення майбутніх оновлень;
– використання сучасних технологій (React, TypeScript, Zustand) повинно полегшувати супровід та рефакторинг.

Аудіо: система повинна забезпечувати плавне відтворення музики та звукових ефектів без затримок.

1.6 Висновки до розділу 1

Було здійснено комплексний аналіз предметної області та інструментарію для створення інтерактивних візуальних новел. Встановлено, що цей жанр є перспективним напрямом ігрової індустрії, успіх якого залежить від гармонійного поєднання літературного наративу та сучасної візуальної складової. В ході порівняльного огляду технічних рішень виявлено суттєві відмінності у підходах до розробки: спеціалізовані рушії (Ren’Py) забезпечують низький поріг входу, але обмежують кастомізацію, тоді як універсальні ігрові рушії (Unity) надають потужний функціонал ціною високої складності освоєння та необхідності значної оптимізації.

На основі проведеного аналізу обґрунтовано доцільність розробки власного програмного забезпечення з використанням зв’язки React та Electron. Такий технологічний стек дозволяє нівелювати недоліки готових рішень, поєднуючи повну свободу в дизайні UI з продуктивністю нативних десктопних застосунків. Підсумком етапу стало формування чітких функціональних та нефункціональних вимог, що заклало необхідний теоретичний фундамент для подальшого проєктування архітектури системи.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА КОМПОНЕНТІВ СИСТЕМИ

2.1 Загальна архітектура додатку

2.1.1 Шарова структура додатку

Архітектура застосунку побудована на багаторівневій моделі, що забезпечує чітке розділення відповідальностей та безпечну взаємодію між частинами системи. Основою слугує парадигма Electron-додатку [12], у якій інтерфейс користувача виконується в окремому Renderer-процесі, а низькорівневі операції та взаємодія з файловою системою здійснюються у Main-процесі. Між цими рівнями функціонує Preload-шар, який формує захищений місток для взаємодії за допомогою механізму IPC [13] (Inter-Process Communication). Шарову структуру додатку та взаємодію її компонентів зображено на рисунку 2.1.

На найнижчому рівні розташовано сховище даних, яке містить структури візуальної новели, сформовані на основі спеціалізованої DSL [14]. Цей рівень включає сценарії, сцени, вибори, умови переходів, ефекти, описи персонажів та їхні графічні ресурси. Дані зберігаються в локальній базі VN.db, що дозволяє виконувати запити без звернення до зовнішніх сервісів.

Взаємодія з базою даних здійснюється з Main-процесу. Саме цей процес відповідає за читання скриптів, завантаження сцен, збереження та відновлення прогресу користувача, роботу з файловою системою, імпорт ресурсів персонажів та інші операції, яким потрібні розширені привілеї. Main-процес не взаємодіє з інтерфейсом напряму, а лише надає чітко визначений API, що експортується швидким IPC-каналом у Preload-шар.

Preload-рівень виступає проміжною безпековою ланкою. Він інкапсулює IPC-виклики, формує обмежене середовище для доступу до функціональності Main-процесу та запобігає виконанню довільного коду в

Renderer-просторі. Усі функції, які використовуються з боку інтерфейсу, оголошені явно та типізовані, що робить взаємодію контрольованою та прозорою.

Renderer-рівень реалізує основну логіку взаємодії з користувачем та візуального представлення новели. Саме тут розташовані механізми відтворення сцен, відображення тексту, анімацій, спрайтів персонажів, обробки переходів та виборів. Логіка виконання сюжету винесена у спеціальний рушій, що отримує дані через API та застосовує правила DSL. Renderer отримує лише дозволені операції, такі як завантаження сцени, отримання стану прогресу чи доступ до графічних ресурсів.

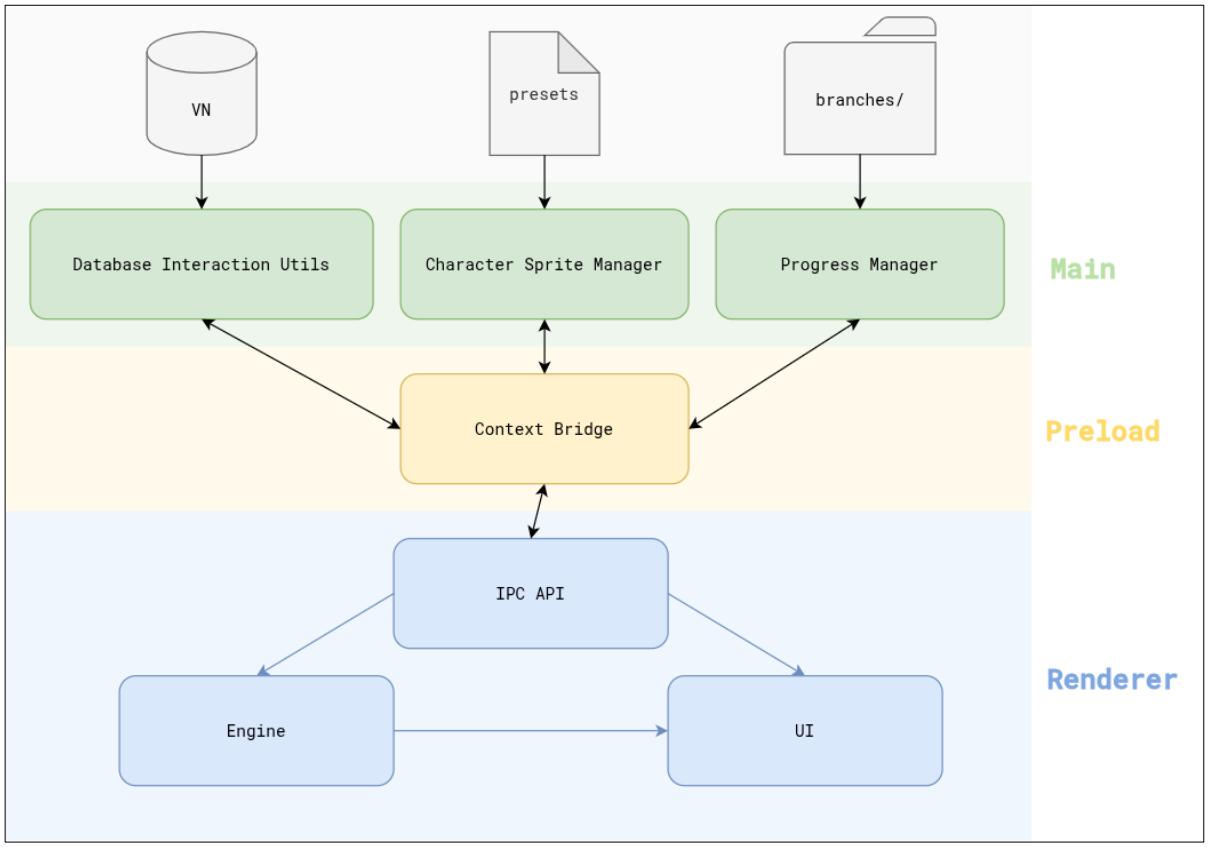


Рисунок 2.1 – Структура шарів додатку та їх взаємодія

Така архітектура забезпечує модульність, безпеку, ізоляцію логічних компонентів та передбачуваність поведінки системи. Обмеження доступу до даних на рівні Renderer ефективно унеможливлює небажану модифікацію файлової системи чи прямі запити до бази даних. Як результат, застосунок

поєднує гнучкість локального зберігання даних із захищеною взаємодією між процесами, що робить його стійким, передбачуваним і легко розширюваним.

2.1.2 Структура та організація бази даних

База даних додатку використовується для зберігання сцен та варіантів вибору візуальної новели. Вона реалізована у вигляді реляційної структури, що відповідає підготовленому DSL для скрипта сценарію. Основна мета використання бази даних – забезпечити зручну та масштабовану вибірку гілок сюжету під час виконання гри.

База даних складається з двох основних таблиць: scenes, choices. Діаграму структури та зв’язків цих таблиць зображено на рисунку 2.2.

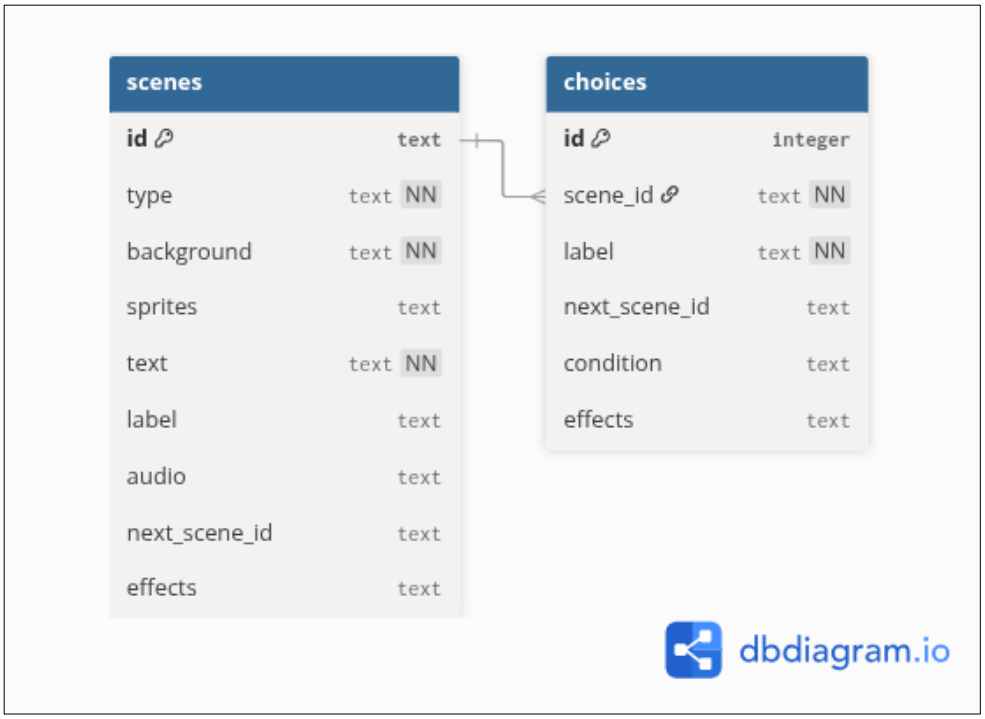


Рисунок 2.2 – Діаграма структури та зв’язків бази даних

Scenes – зберігає інформацію про окремі сцени візуальної новели. Кожна сцена містить дані про її тип (лінійна, з вибором або кінцева), фонове зображення, спрайти персонажів, текстове наповнення, підпис, що вказує на

дійову особу для кожної репліки, аудіо та додаткові ефекти, які застосовуються під час відображення сцени. Таблиця дозволяє організувати послідовність сцен, підтримувати логіку ефектів під час проходження сцени та зберігати зв’язок із наступною сценою через поле next_scene_id.

Choices – містить варіанти вибору, доступні для конкретних сцен. Кожен запис у таблиці зв’язує сцену з можливими переходами до наступних етапів сюжету, визначає умови доступності вибору та ефекти, які застосовуються після прийняття рішення. Завдяки цій таблиці реалізується гнучке гілкування історії та контроль сценарних переходів.

2.2 Проєктування DSL для скрипту візуальної новели

2.2.1 Ідентифікація та класифікація сцен

У розроблені системі візуальної новели кожна сцена має унікальний ідентифікатор (Scene ID), який забезпечує однозначну ідентифікацію та керування порядком виконання сцен у межах різних сюжетних гілок. Ідентифікатор складається з двох компонентів:

- Branch ID – унікальний хеш, що генерується за допомогою бібліотеки nanoid [15] (використання branch ID дозволяє уникнути колізій між сценами гілок і полегшує масштабування сценарію);
- Scene number – порядковий номер сцени у межах конкретної гілки, який визначає послідовність відтворення сцен.

Кожна сцена містить набір обов’язкових та опціональних полів, які визначають її візуальний, аудіо та логічний контент:

- id – ідентифікатор сцени (обов’язкове);
- background – зображення фону сцени (обов’язкове);
- sprites – масив зображень персонажів, що відображаються у сцені (обов’язкове);

– text – репліка персонажа, що відображається користувачу (обов’язкове);
– label – підпис, що вказує на дійову особу для кожної репліки (опціональне);
– audio – звуковий супровід сцени (опціональне);
– effects – набір ефектів, що виконуються під час сцени (опціональне).

Опис структури даних сцени (інтерфейсу SceneContent) представлено на рисунку 2.3.

```
// 1. The core data every scene shares:
interface SceneContent {
  id: string
  background: string
  sprites: string[]
  text: string
  label?: string
  audio?: string
  effects?: Effect[]
}
```

Рисунок 2.3 – Інтерфейс SceneContent, що визначає загальні поля сцени у DSL візуальної новели

Сцени класифікуються на три основні типи, і кожен тип має додаткові поля, специфічні для його функцій.

Лінійна сцена (linear) – призначена для послідовного відтворення сюжету без вибору користувача. Має додаткове поле nextSceneId, що вказує на наступну сцену. Використовується для стандартних сюжетних переходів та інтерактивних вставок.

Сцена з вибором (choice) – дозволяє користувачу впливати на розвиток сюжету через альтернативні варіанти дії. Має поле choice, яке містить об’єкти вибору з label, nextSceneId, effects та condition. Інтегрується з логікою умов для створення розгалужених сюжетів.

Кінцева сцена (end) – завершує конкретну гілку сюжету. Не містить полів для наступних сцен, може містити ефекти та аудіо для фінальної демонстрації результату.

Структури даних для лінійної сцени, сцени з вибором та кінцевої сцени представлені на рисунках 2.4, 2.5 та 2.6 відповідно.

```
// 2. A scene where you just go straight to the next one:
interface LinearScene extends SceneContent {
  type: 'linear'
  nextSceneId: string
}
```

Рисунок 2.4 – Структура типу LinearScene, що описує лінійну сцену з автоматичним переходом до наступної

```
// 3. A branching scene:
export interface Choice {
  label: string
  nextSceneId: string

  condition?: Condition
  effects?: Effect[]
}

interface ChoiceScene extends SceneContent {
  type: 'choice'
  choices: Choice[]
}
```

Рисунок 2.5 – Структура типу ChoiceScene та об’єкта Choice, щоописують сцену з вибором і можливими гілками переходів

```
// 4. A terminal scene with no outgoing links:
interface EndScene extends SceneContent {
  type: 'end'
}
```

Рисунок 2.6 – Структура типу EndScene, що описує кінцеву сцену без переходів до інших елементів сценарію

2.2.2 Система ефектів

Система ефектів у розробленій DSL відповідає за зміни стану гри, активацію подій та виконання допоміжних дій, пов’язаних із перебігом сцени. Ефекти дозволяють сценаристу впливати на внутрішні змінні, запускати медіа, ініціювати додаткову логіку або навіть змінювати напрямок проходження новели. Таким чином, система ефектів є ключовим механізмом, що поєднує опис сцени зі станом гри та забезпечує інтерактивність.

Усі ефекти подаються у вигляді структурованих об’єктів у масиві effects, який може бути присутній у будь-якій сцені чи виборі. Формат ефекту визначається за полем type, після чого в залежності від типу вказуються необхідні параметри. Таке рішення забезпечує розширюваність DSL та дозволяє легко додавати нові типи ефектів без зміни загального синтаксису.

Основні типи ефектів та їх структура в рамках розробленої DSL наведені на рисунку 2.7.

```
// FlagEffect: set a boolean flag
export interface FlagEffect {
  type: 'flag'
  key: string
  value: boolean
}

// VariableEffect: set a value directly
export interface VariableEffect {
  type: 'variable'
  key: string
  value: boolean | number | string
}

// IncrementEffect: increment a numeric variable
export interface IncrementEffect {
  type: 'increment'
  key: string
  by: number
}
```

Рисунок 2.7 – Приклад визначення типів ефектів у DSL (TypeScript)

У рамках DSL підтримуються такі типи ефектів:

– flag – встановлює або скидає логічний прапорець;
– variable – задає значення змінної (число або булеве);
– increment / decrement – збільшує або зменшує значення змінної;
– playSound – відтворює аудіо (короткі ефекти або фоновий звук);
– showEffect – активує візуальний ефект;
– goto – переходить до іншої сцени, може містити умову;
– useMacro – виконує заздалегідь визначений набір ефектів.

2.2.3 Система умов та логічних операторів

Система умов у DSL визначає, коли ефекти виконуються та коли можливий перехід до наступної сцени. Вона забезпечує динамічну і адаптивну логіку сценарію, дозволяючи створювати нелінійні розгалуження та керувати доступністю виборів у choice-сценах. Умови можуть застосовуватися безпосередньо у сцені для контролю виконання ефектів, для визначення доступності варіантів вибору у choice-сценах, а також у goto-ефектах для умовного переходу між сценами.

У рамках DSL підтримуються такі типи умов:

– variable – порівнює значення змінної з еталонним;
– flag – перевіряє стан булевого прапорця;
– not – логічне заперечення умови;
– and – поєднання кількох умов, які повинні виконатися одночасно;
– or – поєднання кількох умов, при якому достатньо виконання будь-якої.

Кожна умова представлена як структурований об’єкт із чітко визначеними полями, що дозволяє комбінувати їх у складні логічні конструкції.

Основні типи умов та їх структура в рамках розробленої DSL наведені на рисунку 2.8.

```ts
// Basic comparison operators
type ComparisonOperator = '==' | '!=' | '>' | '>=' | '<' | '<=';

// Condition: variable comparison
export interface VariableCondition {
  type: 'variable';
  key: string;
  operator: ComparisonOperator;
  value: number | string | boolean;
}

// Condition: flag check (shortcut for variable of boolean type)
export interface FlagCondition {
  type: 'flag';
  key: string;
  value: boolean;
}

// Condition: NOT logic
export interface NotCondition {
  type: 'not';
  condition: Condition;
}
```

Рисунок 2.8 – Приклад визначення типів умов у DSL (TypeScript)

2.2.4 Макроси

Макроси у DSL – це іменовані набори ефектів, які можна повторно використовувати у різних сценах. Вони дозволяють уникати дублювання коду та спрощують підтримку сценаріїв. Для зручності макроси зберігаються у окремому JSON-файлі і активуються через ефект useMacro, що забезпечує гнучку та масштабовану організацію сценаріїв.

Приклад визначення макросів у рамках розробленої DSL наведено на рисунку 2.9.

```json
{
  "healPlayer": [
    { "type": "variable", "key": "playerHealth", "value": 100 },
    { "type": "playSound", "sound": "healing.mp3" }
  ]
}
```

Рисунок 2.9 – Приклад визначення макросів у JSON-файлі

2.3 Рушій виконання сцен та управління переходами

Рушій виконання сцен є центральним програмним компонентом, який забезпечує інтерпретацію DSL-сценарію, обробку логіки виборів, виконання ефектів та навігацію між сценами. Він виступає зв’язувальною ланкою між сценарною структурою, станом прогресу та інтерфейсними компонентами, забезпечуючи послідовний і передбачуваний перебіг візуальної новели.

Розглянемо основні функції рушія.

Завантаження та оновлення сценарію: рушій отримує список сцен для поточної гілки за її ідентифікатором та за потреби перезавантажує сценарний набір, якщо перехід здійснюється до сцени з іншої гілки. Це дозволяє використовувати поділ сценарію на незалежні гілки, що спрощує масштабування проєкту.

Ініціалізація середовища виконання: рушій переходить у робочий стан лише після завершення початкового завантаження сцен. Ініціалізація включає мінімальну затримку для коректної роботи анімацій та уникнення різких змін інтерфейсу. Після ініціалізації рушій гарантує стабільність переходів та логічної обробки.

Пошук і підготовка сцен до показу: під час запиту конкретної сцени рушій виконує її попередню обробку. Це включає фільтрацію недоступних варіантів у сценах вибору на основі умов, виконання ефектів для лінійних сцен, а також обробку можливих умовних переходів через goto. Завдяки цьому сцена завжди повертається у вже обчисленому, логічно коректному стані.

Обробка ефектів пов’язаних зі сценою або вибором: рушій застосовує всі ефекти, пов’язані зі сценою або вибраним варіантом, включаючи зміну змінних, оновлення прапорців, відтворення аудіо, запуск візуальних ефектів та виконання макросів. Усі ефекти обробляються послідовно, що забезпечує передбачувану поведінку системи.

Управління переходами між сценами: рушій відповідає за визначення цільової сцени після виконання логіки поточної. У лінійних сценах перехід

зазвичай відбувається автоматично через nextSceneId, але може бути змінений умовними ефектами goto. У сценах вибору перехід визначається вибором користувача, якщо ефект goto не перевизначає цей маршрут.

Актуалізація прогресу користувача: після визначення наступної сцени рушій оновлює глобальний стан прогресу через відповідний store. Це гарантує, що поточний стан синхронізований із логікою рушія та доступний іншим компонентам додатку, включно з інтерфейсними та модулем збереження.

Підтримка адаптивної логіки проходження: завдяки інтеграції умов, ефектів і сценарної структури рушій дозволяє будувати складні розгалуження, динамічні зміни доступності виборів та альтернативні завершення. Це забезпечує гнучкість у створенні нелінійних історій та глибоку інтеграцію сценарної логіки зі станом гри.

UML-діаграма класу Engine наведена на рисунку 2.10.

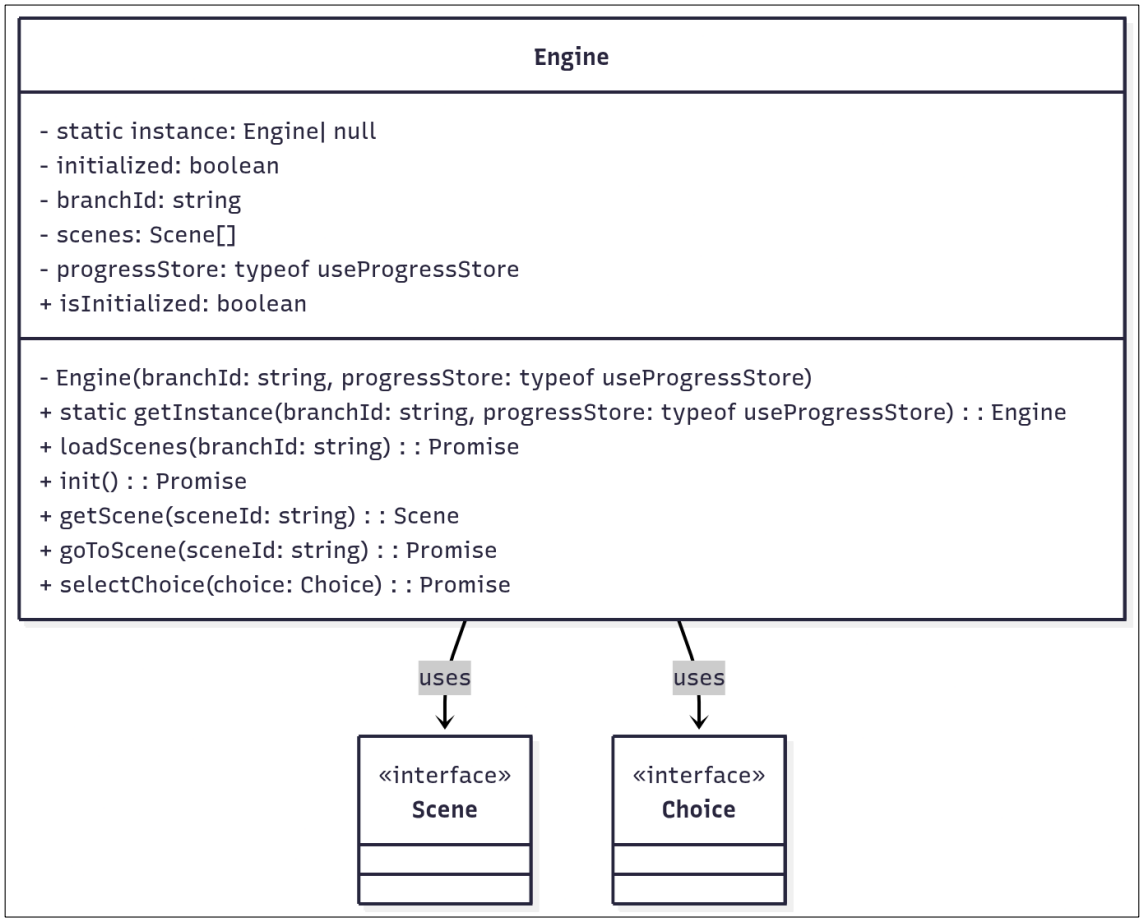


Рисунок 2.10 – UML діаграма класу Engine

2.4 Система збереження та завантаження прогресу

Система збереження прогресу відповідає за фіксацію стану проходження новели та забезпечує можливість повернення до будь-якого моменту. У розробленій архітектурі збереження реалізовано на основі локальних JSON-файлів, які створюються у спеціальній директорії користувача. Це робить систему простою, прозорою та незалежною від зовнішніх сервісів.

Для роботи із збереженнями використовується окремий менеджер, який відповідає за створення, читання, оновлення та видалення файлів. Кожен запис містить слот, часову позначку та повний стан прогресу, включно з активною сценою, значеннями змінних, прапорцями та іншими даними, необхідними для коректного відновлення сценарію. Перед записом система перевіряє коректність імені слоту та забезпечує існування робочої директорії.

Завантаження прогресу здійснюється шляхом читання відповідного JSON-файла і відтворення всіх збережених параметрів. Крім того, передбачено отримання списку доступних збережень та можливість видалення окремих слотів. Комунікація між Renderer-процесом і логікою збережень відбувається через IPC-канали, що забезпечує безпечний доступ до файлової системи та чіткий розподіл відповідальностей між частинами застосунку.

UML-діаграма класу ProgressManager наведена на рисунку 2.11.

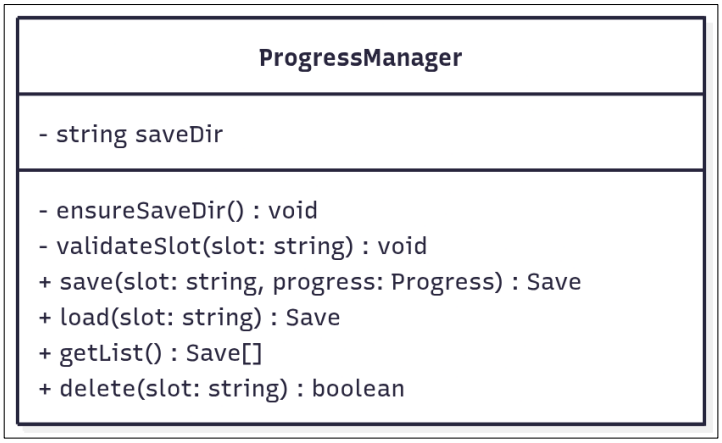


Рисунок 2.11 – UML діаграма класу ProgressManager

2.5 Система візуалізації персонажів

2.5.1 Менеджер спрайтів персонажів

Менеджмент спрайтів у системі забезпечує централізоване завантаження та доступ до графічних ресурсів персонажів. Відповідальний за це клас CharacterSpriteManager керує пресетами, що містять налаштування відображення персонажів у сценах, включаючи положення, зображення та інші властивості.

UML-діаграма класу CharacterSpriteManager наведена на рисунку 2.12.

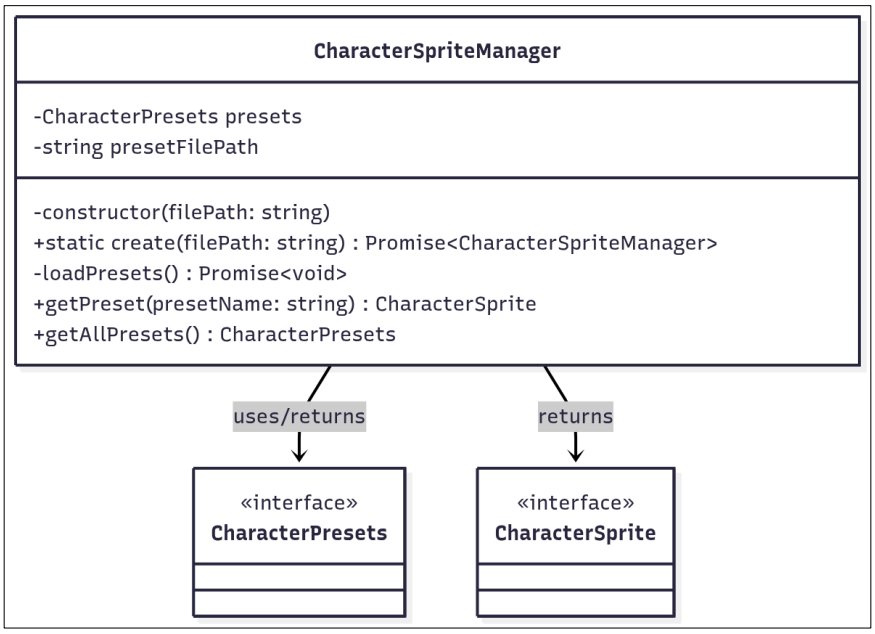


Рисунок 2.12 – UML діаграма класу CharacterSpriteManager

Пресети зберігаються у зовнішньому файлі (JSON), який завантажується при ініціалізації менеджера. Це дозволяє сценаристу та рушію отримувати необхідний спрайт за назвою пресету без дублювання даних і додаткової логіки. Менеджер надає методи для отримання конкретного пресету та повного списку доступних пресетів.

Таке рішення забезпечує:

– централізоване управління ресурсами персонажів;
– безпечний доступ до даних спрайтів з обробкою відсутніх пресетів;

– легке розширення та зміну графічних елементів без втручання у рушій чи сцени;
– підготовку до інтеграції з інтерфейсом користувача та системою виконання сцен.

Менеджер спрайтів є окремим компонентом архітектури, який тісно взаємодіє з рушієм новели та інтерфейсом користувача, забезпечуючи узгоджене та ефективне відображення персонажів у різних сценах.

2.5.2 Структура даних та композиція спрайтів

З метою оптимізації використання пам’яті та забезпечення високої варіативності відображення персонажів, у системі реалізовано підхід композитних спрайтів. На відміну від класичного методу, де для кожної емоції зберігається окреме повне зображення, розроблена архітектура оперує графічними шарами, які накладаються один на одного під час рендерингу (тіло, очі, рот, брови тощо).

Конфігурація візуальних станів персонажів винесена у зовнішні JSON-файли. Це дозволяє відділити дані від логіки відображення. Структура файлу побудована за принципом Key-Value сховища, де:

– ключ – це унікальний серіалізований рядок-ідентифікатор, що кодує набір атрибутів. Такий підхід забезпечує доступ до необхідного пресету за константний час O(1);
– значення – об’єкт конфігурації, що містить шляхи до окремих текстур та параметри позиціонування.

На рисунку 2.13 наведено фрагмент JSON-конфігурації, що демонструє структуру збереження пресету.

Для формалізації структури даних та встановлення суворого контракту взаємодії між модулем зберігання ресурсів та графічним ядром, спроєктовано об’єктну модель спрайту. Ця модель визначає набір правил та типів даних,

які гарантують цілісність візуального образу при його передачі на шар рендерингу.

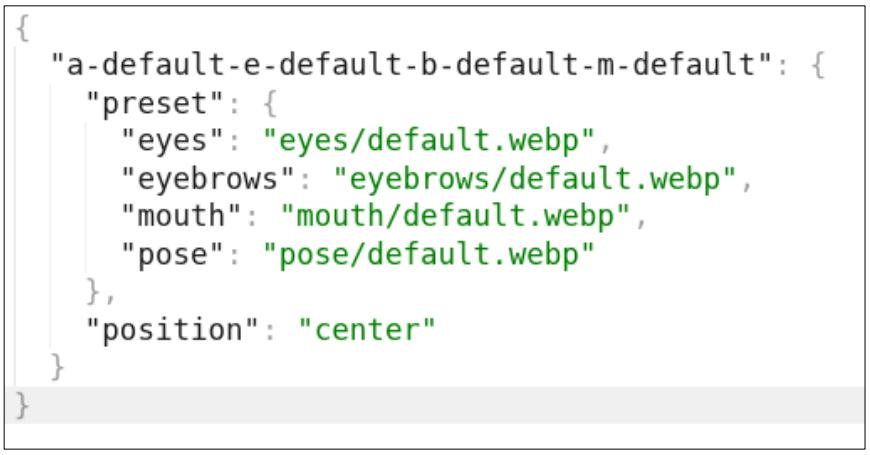


Рисунок 2.13 – Структура JSON-пресету персонажа

Система базується на двох ключових абстракціях:

– композитна модель (CharacterLayers): визначає структуру розкладання персонажа на незалежні графічні шари, що є обов’язковими для формування валідного зображення;
– контекстна модель (CharacterSprite): розширює композитну модель, інтегруючи її з параметрами сцени (просторові координати, стилізація).

Важливим архітектурним рішенням є проєктування поліморфної системи позиціювання. Поле position спроєктовано таким чином, щоб підтримувати два типи координатних систем:

– семантичні аліаси (наприклад, 'center', 'left') – для швидкого сценарного розміщення згідно з сіткою шаблону;
– об'єкт координатних відступів (структура { left, right, bottom }) – для точного позиціювання персонажа відносно меж сцени, що дозволяє реалізувати нестандартні композиційні рішення та адаптивне розміщення.

Специфікацію інтерфейсів, що описують візуальну модель персонажа, наведено на рисунку 2.14.

```
interface CharacterLayers {
  pose: string; // Базовий шар тіла
  eyes: string; // Шар емоції очей
  mouth: string; // Шар емоції рота
  eyebrows: string; // Шар емоції брів
}

interface CharacterSprite {
  // Композитна частина (Шари)
  preset: CharacterLayers;

  // Контекстна частина (позиціонування)
  // Підтримка поліморфних типів: семантичні константи або точні координати
  position?: 'left' | 'center' | 'right' | {
    bottom?: string;
    left?: string;
    right?: string
  };

  style?: CSSProperties; // Додаткові стильові перевизначення
}
```

Рисунок 2.14 – Специфікація інтерфейсів візуальної моделі персонажа

2.6 Проєктування архітектури клієнтського додатку (Renderer Process)

2.6.1 Компонентна структура та глобальний стан

Клієнтська частина системи реалізована як односторінковий додаток (SPA [16]), що виконується в процесі рендерингу Electron. Архітектура базується на компонентному підході бібліотеки React, додаток чітко розділено на три логічні рівні: рівень представлення (UI), рівень даних (Store) та логічне ядро (Engine).

На рисунку 2.15 зображено загальну схему архітектури та організацію компонентів.

Розглянемо основні елементи архітектури.

Application layer (рівень додатку): це «каркас» додатку, що формує кореневу структуру компонентів та керує глобальною поведінкою системи. Зокрема, Loading Provider відповідає за централізоване попереднє

завантаження ключових ресурсів (спрайтів, аудіофайлів та внутрішнього рушія), забезпечуючи коректний старт додатку. Він блокує відображення основного інтерфейсу до моменту повної готовності ресурсів, показуючи екран завантаження. У свою чергу, Modal Provider організовує рендеринг шару модальних вікон поверх основного контенту.

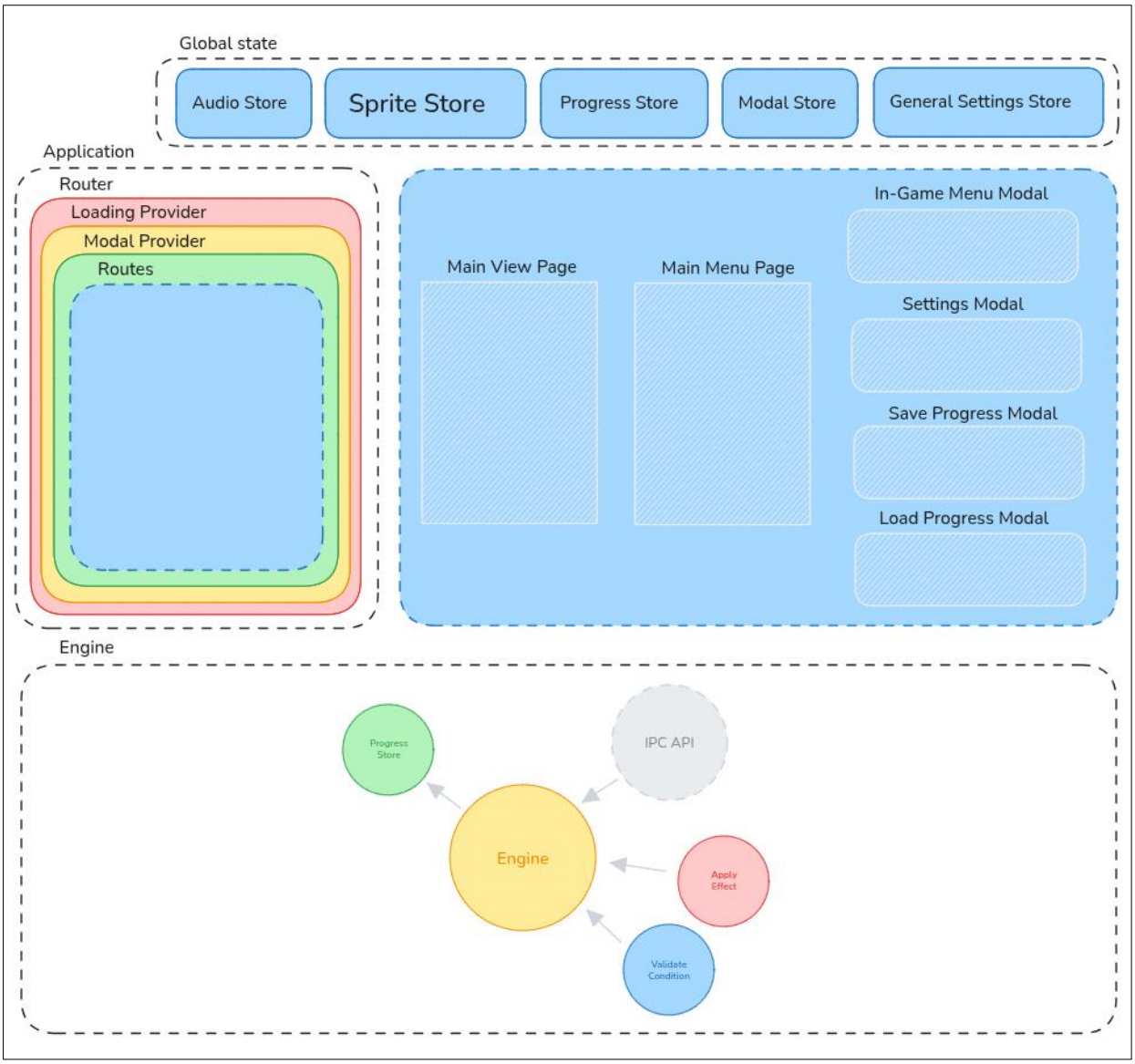


Рисунок 2.15 – Архітектура клієнтського додатку та організація сховищ даних

Routing & Pages (навігація та сторінки): система маршрутизації (Routes) відповідає за перемикання між основними контекстами: головним меню (Main Menu Page) та основним ігровим простором (Main View Page).

Global state (глобальний стан): реалізований на базі бібліотеки Zustand. Для уникнення проблеми “prop drilling” (передачі даних через велику кількість проміжних компонентів) стан винесено у зовнішні атомарні сховища. Sprite Store відповідає за кешування та доступ до графіки, Progress Store зберігає динамічні дані проходження, необхідні для відновлення ігрової сесії, Audio Store керує відтворенням фонової музики, Modal Store контролює видимість та стан перекриваючих вікон, а General Settings Store зберігає загальні налаштування, зокрема швидкість відтворення тексту.

Modals (модальні вікна): компоненти налаштувань, збереження та завантаження винесені в окремий шар поверх основного інтерфейсу. Це архітектурне рішення дозволяє відкривати меню паузи, не розмонтовуючи (unmount) компоненти активної ігрової сцени, що зберігає їх поточний стан та забезпечує безшовну взаємодію.

Engine (ядро): цей модуль інкапсулює бізнес-логіку новели. Він виступає єдиною точкою входу для комунікації з системними процесами Electron (через IPC) та відповідає за виконання ігрової логіки перед тим, як зміни вплинуть на візуальну частину.

2.6.2 Потік керування та цикл оновлення інтерфейсу

Взаємодія користувача з системою побудована на принципі прямого, однонаправленого потоку даних. Інтерфейс транслює наміри користувача, викликаючи відповідні методи зміни стану. Діаграма ілюструє спрощену модель цього процесу, де потік виконання розділяється в залежності від типу події. На рисунку 2.16 наведено схему алгоритму обробки подій.

Цикл оновлення складається з трьох концептуальних етапів:

– user action (дія користувача): компонент інтерфейсу фіксує взаємодію, наприклад, клік мишкою по варіанту відповіді або натискання функціональної клавіші;

– instruction routing (маршрутизація інструкцій): відбувається розподіл потоку виконання залежно від типу події (інструкції, що впливають на хід сюжету (наприклад, вибір репліки), передаються безпосередньо модулю Engine для оновлення глобального стану гри, тоді як суто візуальні маніпуляції (навігація меню, анімації) обробляються локально всередині компонентів або через глобальні сховища);
– state reconciliation (оновлення відображення): будь-яка зміна даних (глобальних чи локальних) автоматично ініціює реактивний цикл оновлення React, який перемальовує лише змінені частини інтерфейсу.

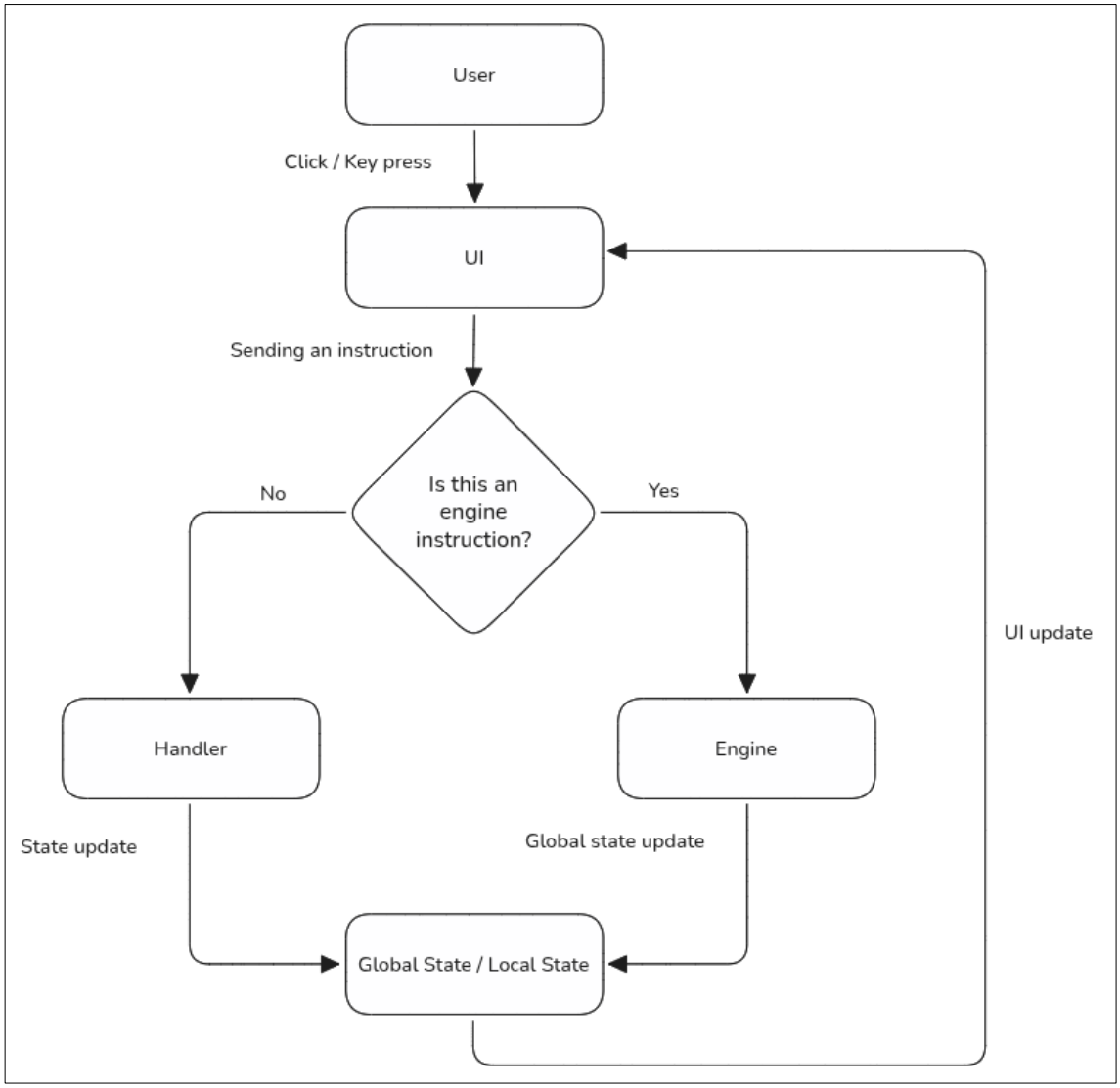


Рисунок 2.16 – Алгоритм обробки подій та оновлення UI

2.7 Висновки до розділу 2

Виконано проєктування архітектури програмного забезпечення, в основу якої покладено модульний підхід на базі фреймворку Electron. Реалізовано чіткий поділ на системний рівень (Main Process) та рівень інтерфейсу (Renderer Process), взаємодія між якими здійснюється через захищений IPC-місток, що гарантує безпеку та стабільність додатку. Для забезпечення гнучкості сценаріїв розроблено структуру бази даних та власну предметно-орієнтовану мову (DSL), яка дозволяє описувати складні нелінійні сюжети з різними типами сцен та умов.

Ключовим елементом системи визначено модуль Engine, який відповідає за обробку ігрової логіки. Систему управління глобальним станом побудовано на основі бібліотеки Zustand із використанням ізольованих сховищ для ігрового прогресу, налаштувань та UI-станів. Такий підхід забезпечує централізований контроль потоків даних та їх миттєву синхронізацію з клієнтською частиною, побудованою на React. Додатково спроєктовано підсистеми управління ресурсами, зокрема механізм збереження прогресу у JSON-файлах та менеджер композитних спрайтів для оптимізації роботи з графікою. Спроєктована архітектура повністю відповідає сформованим вимогам і готова до етапу програмної реалізації.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ВПРОВАДЖЕННЯ

3.1 Конфігурація середовища та структура проєкту

Програмна реалізація системи виконана у середовищі Node.js [17] (версія 20.x) із використанням фреймворку Electron (версія 35.x) та бібліотеки React (версія 19.x). Обґрунтування вибору цих технологій наведено у підрозділі 1.3.

Для ініціалізації програмного оточення та конфігурування процесу збірки застосунку використано інструментарій electron-vite [18] у поєднанні з утилітами @electron-toolkit [19]. Застосування цього стеку забезпечило автоматизоване налаштування робочого середовища, включно з підтримкою Hot Module Replacement (HMR), що пришвидшує ітерації розробки. Крім того, інтеграція з electron-builder [20] надала можливість формувати оптимізовані збірки застосунку для різних операційних систем (через скрипти build:win, build:linux) без потреби у складних ручних конфігураціях.

Структура проєкту розгорнута на основі стандартного шаблону electron-vite та архітектурно доповнена додатковим шаром типізації. Файлова ієрархія продемонстрована на рисунку 3.1.

Основні директорії проєкту виконують такі функції:

– src/main/ – реалізація головного процесу (Main Process) створення вікон застосунку, обробка системних подій, керування життєвим циклом;
– src/renderer/ – рендер-процес (Renderer Process): React-компоненти, стилізація інтерфейсу за допомогою TailwindCSS, керування станом;
– src/preload/ – preload-скрипти, що виконуються до завантаження рендер-процесу (вони формують безпечний IPC-місток між Renderer та Main, інкапсулюючи доступ до внутрішніх API (робота зі сценами, персонажами, прогресом, системними функціями) та

експонуючи його в глобальний контекст за допомогою contextBridge);

– src/shared/ – спільний між процесами шар, призначений для розміщення елементів, що використовуються одночасно в різних шарах (каталог виконує роль централізованого сховища спільних ресурсів, що забезпечує узгодженість взаємодії між процесами; основним його вмістом є TypeScript-типи та інтерфейси, які визначають спільні структури даних і підтримують єдиний контракт між частинами застосунку).

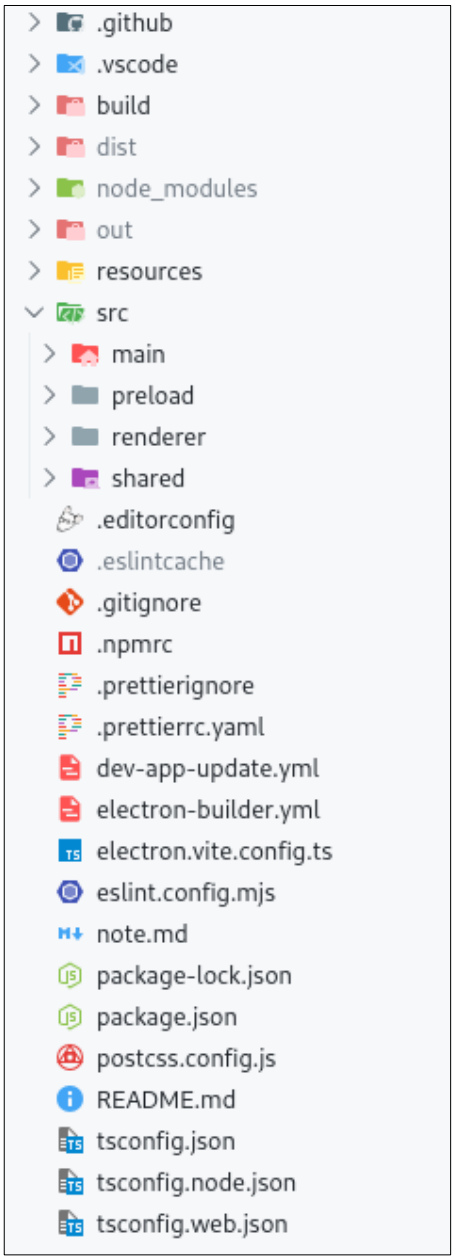


Рисунок 3.1 – Файлова структура проєкту в середовищі розробки

Конфігурацію середовища розробки та збірки системи зосереджено у файлі electron.vite.config.ts, який визначає ключові параметри роботи Vite [21], Electron і пов’язаних модулів.

3.2 Реалізація системного процесу та механізмів IPC

3.2.1 Ініціалізація головного процесу та конфігурація IPC

Точкою входу в додаток є файл /src/main/index.tsx, що виконується в середовищі Node.js (Main Process). Його ключовими задачами є управління життєвим циклом додатку, створення вікна браузера та маршрутизація запитів від інтерфейсу до системної логіки.

Програмна реалізація точки входу, яка демонструє конфігурацію вікна та реєстрацію обробників, зображено на рисунку 3.2.

```
let characterManager: CharacterSpriteManager;
let progressManager: ProgressManager;

function createWindow(): void {
  const mainWindow = new BrowserWindow({
    // ... window config
    webPreferences: {
      preload: join(__dirname, '../preload/index.js'),
      sandbox: false
    }
  });
}

// 1. App Lifecycle & Initialization
app.whenReady().then(async () => {
  // Initialize managers
  characterManager = await CharacterSpriteManager.create(presetsJsonPath);
  progressManager = new ProgressManager();

  createWindow();
});

// 2. API Routing (Handlers registered at module level)

// --- System ---
ipcMain.handle('app:quit', () => app.quit());

// --- Scene Data ---
ipcMain.handle('scene:get-branch', (_event, branchId: string) => {
  return getBranchById(branchId);
});

// ... other handlers
```

Рисунок 3.2 – Ініціалізація головного процесу та реєстрація IPC (спрощено)

У блоці app.whenReady() відбувається ініціалізація сервісів-менеджерів та створення графічного інтерфейсу.

Архітектура файлу побудована за принципом оркестратора: замість безпосереднього виконання операцій з файловою системою чи базою даних у тілі обробників, index.ts виконує роль маршрутизатора. Він приймає сигнали через ipcMain.handle і делегує виконання конкретним спеціалізованим модулям (Managers), що забезпечує чистоту коду та розподіл відповідальності.

3.2.2 Менеджер конфігурації спрайтів

Клас CharacterSpriteManager відповідає за доступ до конфігураційних даних візуальних образів персонажів. Він оперує набором пресетів – попередньо визначених схем, що описують комбінацію шарів (поза, емоції) та параметри позиціонування.

Оскільки операції читання з диска є відносно повільними, постійне звернення до файлової системи при кожній зміні емоції персонажа є неефективним. Для оптимізації продуктивності застосовано підхід попереднього завантаження даних. Весь масив конфігурації зчитується в оперативну пам’ять (у змінну this.presets) один раз – під час ініціалізації додатку.

Технічною особливістю реалізації є використання асинхронного фабричного методу (static create). Оскільки конструктор класу має бути синхронним, логіку читання файлу винесено у статичний фабричний метод. Він створює екземпляр класу, чекає завершення завантаження даних і лише тоді повертає готовий до роботи об’єкт.

Програмний код, що реалізує описаний підхід до ініціалізації та доступу до даних, зображено на рисунку 3.3.

```
export class CharacterSpriteManager {
    private presets: CharacterPresets = {};
    private presetFilePath: string;

    private constructor(filePath: string) {
        this.presetFilePath = filePath;
    }

    static async create(filePath: string): Promise<CharacterSpriteManager> {
        const manager = new CharacterSpriteManager(filePath)
        await manager.loadPresets()
        return manager
    }

    private async loadPresets(): Promise<void> {
        try {
            const data = await fs.readFile(this.presetFilePath, 'utf-8')
            this.presets = JSON.parse(data)
        } catch (error) {
            console.error('Failed to load presets:', error)
        }
    }

    public getPreset(presetName: string): CharacterSprite {
        const preset = this.presets[presetName]
        if (!preset) {
            throw new Error(`Character preset '${presetName}' not found`)
        }
        return preset
    }

    public getAllPresets(): CharacterPresets {
        return { ...this.presets }
    }
}
```

Рисунок 3.3 – Реалізація менеджера спрайтів

3.2.3 Менеджер збереження прогресу

Клас ProgressManager забезпечує надійне зберігання ігрового прогресу у файловій системі. Він виступає посередником між грою та диском, реалізуючи повний набір операцій для управління слотами збережень (створення, читання, видалення).

Важливі технічні аспекти реалізації:

– кросплатформність: шлях до директорії збережень формується динамічно через app.getPath('userData') – це гарантує, що файли будуть створені у правильній системній папці незалежно від операційної системи користувача (Windows, macOS чи Linux);

– валідація та безпека: метод validateSlot використовує регулярні вирази для перевірки імені файлу – це запобігає використанню заборонених символів та захищає від помилок при записі;
– метадані: під час запису до об’єкта прогресу автоматично додається часова мітка (timestamp), що дозволяє інтерфейсу коректно сортувати збереження за датою.

Ключову логіку класу, що демонструє алгоритм запису даних та валідації, зображено на рисунку 3.4.

```
export class ProgressManager {
    private saveDir: string;

    constructor() {
        const userDataPath = app.getPath('userData')
        this.saveDir = path.join(userDataPath, 'save')
    }

    private ensureSaveDir(): void {
        if (!fs.existsSync(this.saveDir)) {
            fs.mkdirSync(this.saveDir, { recursive: true })
        }
    }

    private validateSlot(slot: string): void {
        if (!/^[a-zA-Z0-9_\-]+$/.test(slot)) {
            throw new Error('Invalid slot name. Use only letters, numbers, hyphens, and underscores.')
        }
    }

    save(slot: string, progress: Progress): Save {
        this.validateSlot(slot)
        this.ensureSaveDir()

        const const save: Save = {
            slot,
            timestamp: new Date().toISOString(),
            ...progress
        }

        const filePath = path.join(this.saveDir, `${slot}.json`)
        fs.writeFileSync(filePath, JSON.stringify(save, null, 2))

        return save
    }

    // ... load(), getList(), and delete() methods are implemented similarly using fs module
}
```

Рисунок 3.4 – Реалізація менеджера прогресу (скорочено)

3.2.4 Реалізація шару доступу до даних сценарію

Робота з даними сценарію реалізована через гібридний підхід. Для зручності написання та версіонування сюжету використовується формат JSON (текстові файли), тоді як для забезпечення швидкодії під час виконання гри (Runtime) дані переносяться у реляційну базу даних SQLite [22]. Цей процес розділено на два етапи: підготовка даних (Seeding) та вибірка (Fetching). Для перенесення сценарію з JSON-файлів у базу розроблено скрипт міграції. Його основна задача – трансформувати ієрархічну структуру об’єктів у плоскі реляційні таблиці.

Критично важливим аспектом реалізації є використання транзакцій (db.transaction). Це гарантує атомарність операції: сценарій гілки завантажується або повністю, або не завантажується зовсім. Також скрипт виконує серіалізацію складних типів (масиви спрайтів, об’єкти ефектів) у JSON-рядки, оскільки SQLite не підтримує ці типи даних нативно.

Реалізацію скрипта імпорту даних зображено на рисунку 3.5.

```
const insertScene = db.prepare(`
    INSERT INTO scenes (id, type, background, sprites, text, label, audio, next_scene_id, effects)
    VALUES (@id, @type, @background, @sprites, @text, @label, @audio, @nextSceneId, @effects)
`)

const insertChoice = db.prepare(`
    INSERT INTO choices (sceneId, id, label, next_scene_id, condition, effects)
    VALUES (@sceneId, @id, @label, @nextSceneId, @condition, @effects)
`)

const insertScript = db.transaction((scenes: Scene[]) => {
    for (const scene of scenes) {
        insertScene.run({
            id: scene.id,
            type: scene.type,
            background: scene.background,
            sprites: JSON.stringify(scene.sprites),
            text: scene.text,
            label: scene.label ?? null,
            audio: scene.audio ?? null,
            nextSceneId: scene.type === 'linear' ? scene.nextSceneId : null,
            effects: scene.effects ? JSON.stringify(scene.effects) : null
        })

        if (scene.type === 'choice') {
            for (const choice of scene.choices) {
                insertChoice.run({
                    sceneId: scene.id,
                    id: choice.id,
                    label: choice.label,
                    nextSceneId: choice.nextSceneId,
                    condition: choice.condition ? JSON.stringify(choice.condition) : null,
                    effects: choice.effects ? JSON.stringify(choice.effects) : null
                })
            }
        }
    }
})
```

Рисунок 3.5 – Реалізація скрипта імпорту даних

На етапі виконання програми функція getBranchById виконує зворотну операцію. Вона діє як Data Mapper, зчитуючи нормалізовані дані з кількох таблиць та реконструюючи вкладену структуру об’єктів, яку очікує ігровий рушій.

Для оптимізації продуктивності використано підготовлені запити, які компілюються один раз при запуску додатку.

Реалізацію вибірки з бази даних зображено на рисунку 3.6.

```
export function getBranchById(branchId: string): Scene[] {
    // Prepare cached SQL statements
    const getScenes = db.prepare('SELECT * FROM scenes WHERE id LIKE ?')
    const getChoicesForScene = db.prepare('SELECT * FROM choices WHERE sceneId = ?')

    const pattern = `${branchId}%`
    const rawScenes = getScenes.all(pattern) as SceneDB[]

    const scenes = rawScenes.map((rawScene) => {
        const { id, background, text, label, audio } = rawScene
        const sprites = JSON.parse(rawScene.sprites) as string[]
        const effects = rawScene.effects ? (JSON.parse(rawScene.effects) as Effect[]) : undefined

        // Map raw DB rows to domain objects
        if (rawScene.type === 'end') {
            const scene: Scene = { /* ... map end scene ... */ }
            return scene
        } else if (rawScene.type === 'choice') {
            // Map choices
            const rawChoices = getChoicesForScene.all(rawScene.id) as ChoiceDB[]
            const choices: Choice[] = rawChoices.map((rawChoice) => ({ /* ... map choice ... */ }))

            // Map choice scene
            const scene: Scene = { /* ... map choice scene ... */ }
            return scene
        } else { // Map linear scene
            const scene: Scene = { /* ... map linear scene ... */ }
            return scene
        }
    })

    return scenes
}
```

Рисунок 3.6 – Реалізація вибірки даних (скорочено)

3.3 Програмна реалізація рушія

3.3.1 Реалізація контролера сценарію

Ядром логіки виконання візуальної новели виступає клас Engine. Він інкапсулює методи завантаження ресурсів, управління потоком сценарію та взаємодію з глобальним станом.

Для реалізації контролера обрано породжувальний патерн проєктування Singleton (Одинак). Це архітектурне рішення обумовлене необхідністю мати єдину точку доступу до екземпляра рушія з будь-якого компонента системи, а також унеможливити створення дубльованих контролерів, що могло б призвести до розсинхронізації ігрового процесу.

Ініціалізація класу передбачає передачу параметрів конфігурації: початкового ідентифікатора гілки (branchId) та посилання на сховище прогресу (progressStore). Це дозволяє встановити необхідний зв'язок між логікою рушія та глобальним станом гри безпосередньо в момент створення екземпляра.

Реалізацію патерну Singleton для класу Engine наведено на рисунку 3.7.

```
export class Engine {
    private static instance: Engine | null = null
    private initialized = false

    private branchId: string
    private scenes: Scene[]
    private progressStore: ProgressStore

    private constructor(branchId: string, progressStore: ProgressStore) {
        this.branchId = branchId
        this.progressStore = progressStore
        this.scenes = []
    }

    public static getInstance(branchId: string, progressStore: ProgressStore): Engine {
        if (!Engine.instance) {
            Engine.instance = new Engine(branchId, progressStore)
        }
        return Engine.instance
    }

    // ...
}
```

Рисунок 3.7 – Реалізація патерну Singleton та ініціалізація класу Engine

Для забезпечення коректного запуску ігрового процесу реалізовано асинхронний метод init. Його головна задача – підготувати рушій до роботи до того, як інтерфейс користувача почне рендеринг сцени.

Окрім завантаження початкових даних, метод реалізує UX-патерн "Minimum Loading Time". Він розраховує час, витрачений на завантаження

даних, і, якщо він менший за встановлений поріг (наприклад, 2000 мс), примусово утримує заставку. Це дозволяє уникнути візуального дефекту «миготіння» (flickering), коли екран завантаження зникає занадто швидко, що може дезорієнтувати користувача.

Реалізацію методу ініціалізації рушія наведено на рисунку 3.8.

```
public async init(): Promise<void> {
    if (this.initialized) return

    const minDuration = 2000
    const startTime = Date.now()

    await this.loadScenes(this.branchId)

    const elapsed = Date.now() - startTime
    const remaining = minDuration - elapsed
    if (remaining > 0) {
        await new Promise((resolve) => setTimeout(resolve, remaining))
    }

    this.initialized = true
}
```

Рисунок 3.8 – Метод ініціалізації рушія

Важливим аспектом реалізації є механізм роботи з пам’яттю. Сценарій гри розділено на логічні частини – «гілки» (branches). Алгоритм динамічного завантаження гілок сценарію та навігації між сценами наведено на рисунку 3.9.

```
async loadScenes(branchId: string): Promise<void> {
    this.scenes = await window.api.scene.getBranch(branchId)
}

async goToScene(sceneId: string): Promise<void> {
    const branchId = sceneId.split('-')[0]

    if (this.branchId !== branchId) {
        this.branchId = branchId
        await this.loadScenes(branchId)
    }

    const sceneExists = this.scenes.some(scene => scene.id === sceneId)

    if (!sceneExists) throw new Error(`Scene with ID "${sceneId}" not found.`)

    this.progressStore.setCurrentSceneId(sceneId)
}
```

Рисунок 3.9 – Алгоритм динамічного завантаження гілок сценарію та навігації

Замість завантаження всього скрипту в оперативну пам’ять під час старту, реалізовано метод loadScenes, який виконує асинхронний запит до головного процесу через IPC-канал (window.api.scene.getBranch) для отримання масиву лише необхідних сцен, реалізуючи підхід Lazy Loading. Метод goToScene автоматично визначає, чи потрібно підвантажити нову гілку перед переходом.

Метод getScene виконує ключову роль у забезпеченні динамічності сюжету. Він не просто повертає статичні дані з JSON-файлу, а адаптує об’єкт сцени та глобальний стан під поточний контекст гри, виконуючи три послідовні операції:

– фільтрація виборів: якщо сцена передбачає розгалуження (тип choice), система перевіряє умову (condition) кожного варіанту (варіанти, які не відповідають поточному прогресу виключаються з видачі);
– мутація глобального стану: для сцен лінійного типу виконується негайне застосування ефектів, що не пов’язані з навігацією – це дозволяє змінювати ігрові змінні автоматично в момент завантаження сцени;
– обробка умовних переходів: перевіряється наявність ефектів типу goto – якщо умова такого ефекту істинна, рушій автоматично підміняє ідентифікатор наступної сцени (nextSceneId), направляючи гравця на альтернативну сюжетну гілку без його прямого втручання.

Алгоритм попередньої обробки та адаптації сцени наведено на рисунку 3.10.

Реакція на дії користувача реалізована у методі selectChoice. Цей метод відповідає за застосування наслідків вибору (мутацію стану) та ініціацію переходу до наступного епізоду.

Важливою особливістю реалізації є пріоритетність операцій: спочатку застосовуються всі побічні ефекти через утиліту applyEffect, і лише після

цього виконується навігація. Це гарантує, що наступна сцена завантажиться вже з оновленим станом.

Метод обробки вибору гравця selectChoice наведено на рисунку 3.11.

```
getScene(sceneId: string): Scene {
    const scene = this.scenes.find((scene) => scene.id === sceneId)

    if (!scene) throw new Error(`Scene with ID "${sceneId}" not found.`)

    if (scene.type === 'choice' && scene.choices) {
        scene.choices = scene.choices.filter((choice) =>
            choice.condition ? validateCondition(choice.condition) : true
        )
    }

    if (scene.type === 'linear' && scene.effects) {
        for (const effect of scene.effects) {
            if (effect.type !== 'goto') {
                applyEffect(effect)
                continue
            }

            if (effect.condition && validateCondition(effect.condition)) {
                scene.nextSceneId = effect.sceneId
            }
        }
    }

    return scene
}
```

Рисунок 3.10 – Алгоритм попередньої обробки та адаптації сцени

```
async selectChoice(choice: Choice): Promise<void> {
    if (!choice.effects || choice.effects.length === 0) {
        await this.goToScene(choice.nextSceneId)
        return
    }

    for (const effect of choice.effects) {
        if (effect.type !== 'goto') {
            applyEffect(effect)
        }
    }

    for (const effect of choice.effects) {
        if (effect.type === 'goto') {
            const shouldGo = !effect.condition || validateCondition(effect.condition)

            if (shouldGo) {
                await this.goToScene(effect.sceneId)
                return
            }
        }
    }

    await this.goToScene(choice.nextSceneId)
}
```

Рисунок 3.11 – Метод обробки вибору гравця

3.3.2 Організація глобального стану

Сховище useProgress виконує роль центральної бази даних ігрової сесії в оперативній пам’яті. Його головна задача – утримувати та синхронізувати дані, які формують поточний контекст історії та підлягають серіалізації.

Структура стану включає три ключові компоненти:

– контекст сцени**:** ідентифікатор активного вузла сценарію;

– змінні стану: динамічні дані, що змінюються в результаті виборів гравця;

– збереження: колекція повних зліпків (snapshots) ігрового стану.

Важливою архітектурною особливістю модуля є інтеграція з IPC-мостом. Асинхронні методи getSaveSlot, saveSlot, deleteSaveSlot та loadSaves інкапсулюють логіку обміну даними між Renderer-процесом та файловою системою, делегуючи збереження, видалення та отримання списку збережень головному процесу Electron.

Структуру інтерфейсу сховища ігрового прогресу наведено на рисунку 3.12.

```
interface ProgressStore {
    // --- Core State Data ---
    currentSceneId: string
    flags: Record<string, boolean>
    variables: Record<string, string | number | boolean>
    saves: Save[] | null

    // --- Synchronous Mutations ---
    setCurrentSceneId: (sceneId: string) => void
    setFlag: (key: string, value: boolean) => void
    setVariable: (key: string, value: string | number | boolean) => void
    setProgressState: (progressState: ProgressState) => void
    // Resets state to initial configuration
    reset: () => void

    // --- Persistence / Async Actions (IPC) ---
    // Serializes current state to file
    saveSlot: (slot?: string) => Promise<void>
    // Retrieves a specific save slot
    getSaveSlot: (slot?: string) => Promise<Save>
    // Retrieves list of available saves
    loadSaves: () => Promise<void>
    // Deletes a specific save slot
    deleteSaveSlot: (slot: string) => Promise<void>
}
```

Рисунок 3.12 – Інтерфейс сховища ігрового прогресу (спрощено)

Для забезпечення звукового супроводу ігрового процесу створено окреме сховище useAudioStore. На відміну від сховища прогресу, цей модуль не містить персистентних даних (які зберігаються на диск), а керує життєвим циклом об’єктів HTMLAudioElement у середовищі браузера.

При реалізації модуля застосовано ряд технічних рішень для покращення користувацького досвіду:

– попереднє завантаження (preloading): метод preloadAudio ініціалізує та кешує аудіо-об’єкти до початку сцени – це усуває затримки, які могли б виникнути при спробі відтворити звук «на льоту»;
– поліфонія звукових ефектів: реалізація методу playSFX використовує нативний DOM-метод cloneNode – це дозволяє створювати незалежні копії аудіо-елемента для відтворення одного і того ж звуку (наприклад, ефекту друку тексту) багаторазово з накладанням, не перериваючи попередній екземпляр;
– управління фоновою музикою: забезпечує безперервне відтворення саундтреку з підтримкою зациклення.

Структуру сховища стану аудіо супроводу наведено на рисунку 3.13.

```
interface AudioStore {
    // --- Browser Audio Instances (Volatile State) ---
    // Active background music element
    bgm: HTMLAudioElement | null
    // Registry of pre-loaded sound effects
    sfx: Record<string, HTMLAudioElement>
    // UI blocking flag during asset fetching
    isLoading: boolean

    // Current volume levels (normalized 0.0 to 1.0)
    bgmVolume: number
    sfxVolume: number

    // --- Asset Management ---
    // Async preloader with audio context warm-up strategy
    preloadAudio: (bgmSrc: string, sfxList: Record<string, string>) => Promise<void>
    // Stops playback and clears memory
    reset: () => void

    // --- Playback Control ---
    playBGM: () => void
    stopBGM: () => void
    toggleBGM: () => void
    // Plays effect using cloneNode() for polyphony support
    playSFX: (name: string) => void

    // --- Volume Control ---
    // Updates background music volume immediately
    setBGMVolume: (volume: number) => void
    // Updates global volume multiplier for sound effects
    setSFXVolume: (volume: number) => void
}
```

Рисунок 3.13 – Структура сховища стану аудіо супроводу

3.3.3 Система обробки умов та ефектів

Шар бізнес-логіки рушія відповідає за інтерпретацію правил сценарію. Він пов’язує статичні дані JSON-сценарію з динамічним станом гри, забезпечуючи виконання скриптових подій та перевірку логічних умов. Реалізація виконана у функціональному стилі з використанням функцій-обробників.

Функція applyEffect виступає центральним диспетчером для будь-яких змін стану гри. Вона аналізує тип переданого ефекту (effect.type) та маршрутизує виконання до відповідної спеціалізованої функції (applyFlagEffect, applySoundEffect тощо).

Реалізація функції-диспетчера, яка забезпечує маршрутизацію викликів та дотримання архітектурних обмежень, зображено на рисунку 3.14.

```
import type { Effect } from '@shared/types/effects'
import { applyFlagEffect } from './applyFlagEffect'
import { applyVariableEffect } from './applyVariableEffect'
import { applyIncrementEffect } from './applyIncrementEffect'
import { applyDecrementEffect } from './applyDecrementEffect'
import { applySoundEffect } from './applySoundEffect'

export function applyEffect(effect: Effect): void {
    switch (effect.type) {
        case 'flag': {
            applyFlagEffect(effect)
            break
        }
        case 'variable': {
            applyVariableEffect(effect)
            break
        }
        case 'increment': {
            applyIncrementEffect(effect)
            break
        }
        case 'decrement': {
            applyDecrementEffect(effect)
            break
        }
        case 'goto': {
            throw new Error(
                'GotoEffect is not allowed inside applyEffect(). ' +
                'Navigation must be handled by Engine.selectChoice() or Engine.getScene().'
            )
        }
        case 'playSound': {
            applySoundEffect(effect)
            break
        }
    }
}
```

Рисунок 3.14 – Реалізація диспетчера ефектів

Критично важливим аспектом реалізації є чітке розмежування відповідальності: функція обробляє виключно побічні ефекти сцени (мутації змінних стану, відтворення аудіо), але не керує потоком виконання. Обробка навігаційного ефекту goto всередині цієї функції явно заборонена, що гарантує контроль над перемиканням сцен виключно з боку класу Engine.

Функція validateCondition відповідає за обчислення булевого результату логічних виразів, що визначають доступність виборів або спрацювання подій. Вона забезпечує єдиний інтерфейс для валідації будь-яких типів умов, розподіляючи їх на дві категорії:

– базові умови: перевірка конкретних даних (прапорів або змінних) делегується окремим допоміжним функціям, які безпосередньо взаємодіють зі станом;
– логічні оператори: типи and, or та not слугують контейнерами, які об’єднують результати перевірки кількох вкладених умов в один підсумковий результат.

Програмна реалізація цього механізму зображена на рисунку 3.15.

```
export function validateCondition(condition: Condition): boolean {
    switch (condition.type) {
        case 'flag': {
            return validateFlagCondition(condition)
        }
        case 'variable': {
            return validateVariableCondition(condition)
        }
        case 'not': {
            return !validateCondition(condition.condition)
        }
        case 'and': {
            return condition.conditions.every((condition) => validateCondition(condition))
        }
        case 'or': {
            return condition.conditions.some((condition) => validateCondition(condition))
        }
        default: {
            return false
        }
    }
}
```

Рисунок 3.15 – Реалізація диспетчера перевірки умов

3.3.4 Інтеграція рушія з реактивним інтерфейсом

Інтеграція імперативного класу Engine з декларативними компонентами React реалізована через кастомний хук useEngine. Цей модуль виконує роль сполучної ланки, інкапсулюючи логіку ініціалізації та надаючи інтерфейсу спрощений API для отримання даних і управління сюжетом.

Реалізація хука базується на низці архітектурних рішень для забезпечення стабільності та продуктивності. Для збереження посилання на екземпляр рушія між циклами рендерингу використано useRef, що дозволяє уникнути повторного створення об’єкта класу. Процес підготовки системи до роботи виконується асинхронно всередині useEffect, завдяки чому ініціалізація не блокує відображення інтерфейсу, а стан готовності контролюється локальним індикатором isReady. З метою оптимізації ресурсів застосовано мемоізацію: хук useMemo гарантує, що ресурсомістка логіка формування сцени в методі getScene виконується лише при зміні ідентифікатора сцени, а useCallback забезпечує посилальну цілісність методів навігації для запобігання зайвим перемальовуванням дочірніх компонентів.

Реалізацію хука useEngine, що забезепечує інтеграцію рушія з React-інтерфейсом, послідовно наведено на рисунках 3.16 та 3.17 відповідно.

```
export default function useEngine(): {
  currentScene: Scene | null
  goToScene: (sceneId: string) => Promise<void>
  selectChoice: (choice: Choice) => Promise<void>
  isReady: boolean
} {
  const progressStore = useProgressStore()
  const currentSceneId = progressStore.currentSceneId

  const engineRef = useRef<Engine>(Engine.getInstance(scriptStart.branchId, progressStore))
  const engine = engineRef.current

  const [isReady, setIsReady] = useState(engine.isInitialized)

  const currentScene = useMemo(() => {
    return isReady ? engine.getScene(currentSceneId) : null
  }, [currentSceneId, isReady, engine])

  // Wraps engine.goToScene
```

Рисунок 3.16 – Реалізація хука useEngine (скорочено, частина 1)

```
export default function useEngine(): {
  currentScene: Scene | null
  goToScene: (sceneId: string) => Promise<void>
  selectChoice: (choice: Choice) => Promise<void>
  isReady: boolean
} {
  const progressStore = useProgressStore()
  const currentSceneId = progressStore.currentSceneId

  const engineRef = useRef<Engine>(Engine.getInstance(scriptStart.branchId, progressStore))
  const engine = engineRef.current

  const [isReady, setIsReady] = useState(engine.isInitialized)

  const currentScene = useMemo(() => {
    return isReady ? engine.getScene(currentSceneId) : null
  }, [currentSceneId, isReady, engine])

  // Wraps engine.goToScene
```

Рисунок 3.17 – Реалізація хука useEngine (скорочено, частина 2)

Такий підхід забезпечує чітке розділення логіки: компоненти UI займаються виключно візуалізацією, отримуючи від хука вже підготовлені дані.

3.4 Реалізація клієнтської частини додатку

3.4.1 Структура та функціональні елементи інтерфейсу користувача

Menu Page – реалізує головне меню додатку, яке дозволяє користувачу розпочати нову гру, відтворити попередній прогрес, завантажити збереження, відкрити налаштування або вийти з додатку.

Взаємодія з кнопками меню здійснюється через обробники подій, що використовують сховища для збережень (useProgressStore) та управління модальними вікнами (useModalStore). Дизайн сторінки показано на рисунку 3.18.

Main View Page – реалізує основний ігровий інтерфейс, відображає поточну сцену та дозволяє користувачу взаємодіяти з її елементами.

Сховище прогресу (useProgressStore) забезпечує швидке збереження та завантаження прогресу. Управління подіями клавіатури (Enter, Escape, F5, F9) реалізовано через кастомний хук useKeyboard. Дизайн сторінки показано на рисунку 3.19.



Рисунок 3.18 – Головне меню додатку



Рисунок 3.19 – Основний ігровий інтерфейс

LoadingScreen – компонент, що відображає екран завантаження з індикацією процесу. Використовується під час початкового завантаження ресурсів у LoadingProvider та всередині Main View Page, поки сцена ще не готова до відображення. Дизайн компонента показано на рисунку 3.20.

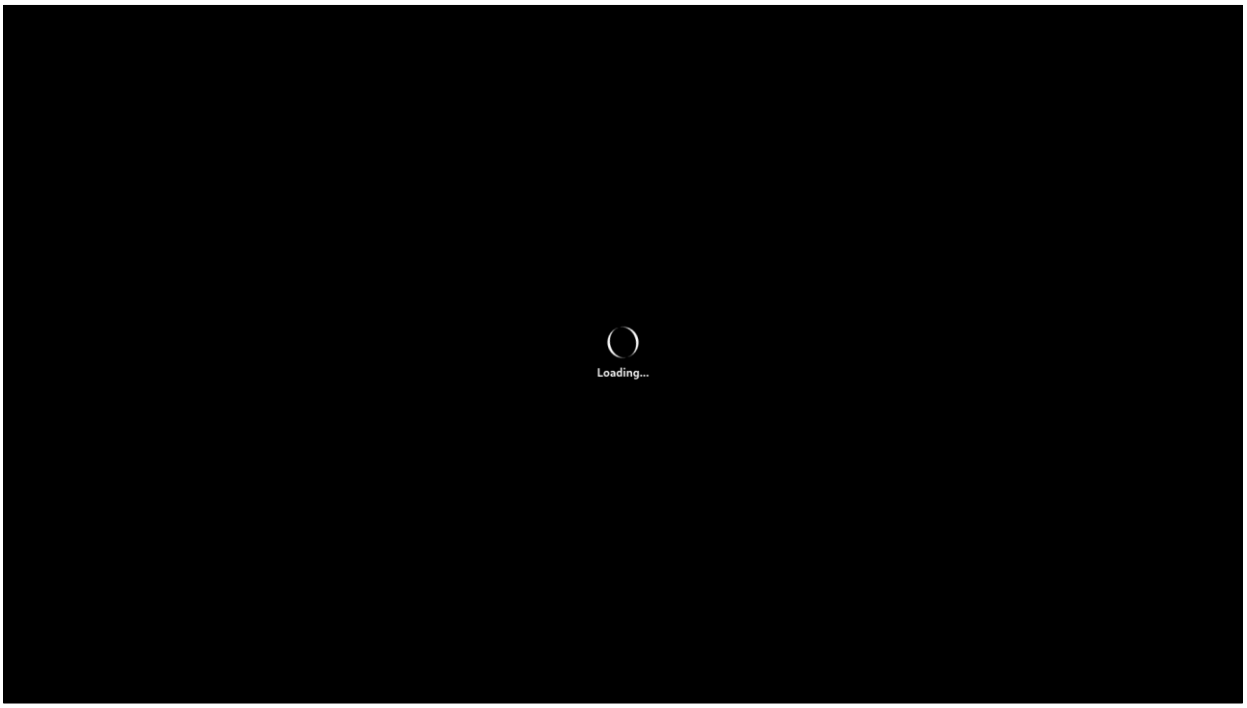


Рисунок 3.20 – Екран завантаження

Menu Modal – модальне меню, яке надає користувачу доступ до основних дій під час гри: почати нову гру, продовжити поточний прогрес, зберегти або завантажити прогрес, відкрити налаштування, повернутися до головного меню або вийти з додатку. Взаємодія з кнопками реалізована через сховища прогресу (useProgressStore) та управління модальними вікнами (useModalStore). Дизайн компонента показано на рисунку 3.21.

Save Progress Modal – модальне вікно для збереження прогресу гри у певний слот. Користувач вводить назву слота, після чого прогрес гри (поточна сцена, прапорці та змінні) зберігається у зовнішнє сховище через IPC API (window.api.progress.save). Компонент перевіряє правильність введеної назви та відображає повідомлення про помилки при некоректному форматі. Дизайн модального вікна показано на рисунку 3.22.



Рисунок 3.21 – Модальне меню

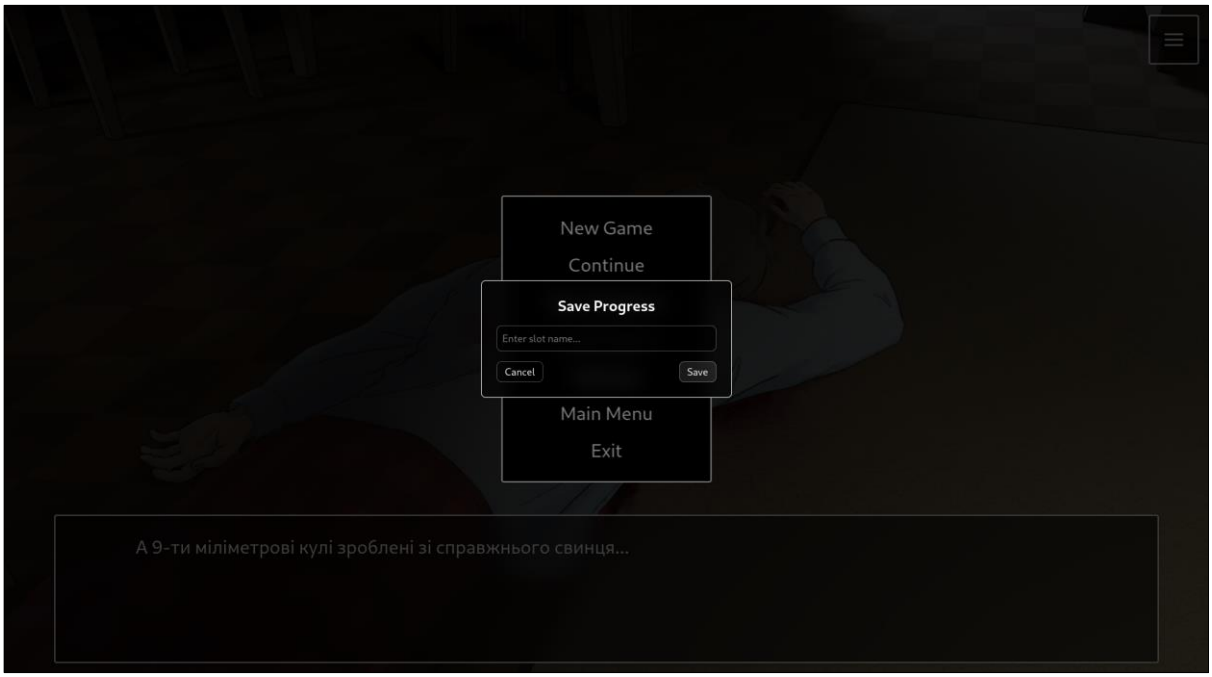


Рисунок 3.22 – Модальне вікно збереження прогресу

Load Progress Modal – модальне вікно для завантаження збереженого прогресу гри. Користувач обирає слот із наявних збережень, після чого прогрес гри (поточна сцена, прапорці та змінні) завантажується через IPC API та відображається у додатку. Компонент також дозволяє видаляти непотрібні збереження. Дизайн модального вікна показано на рисунку 3.23.

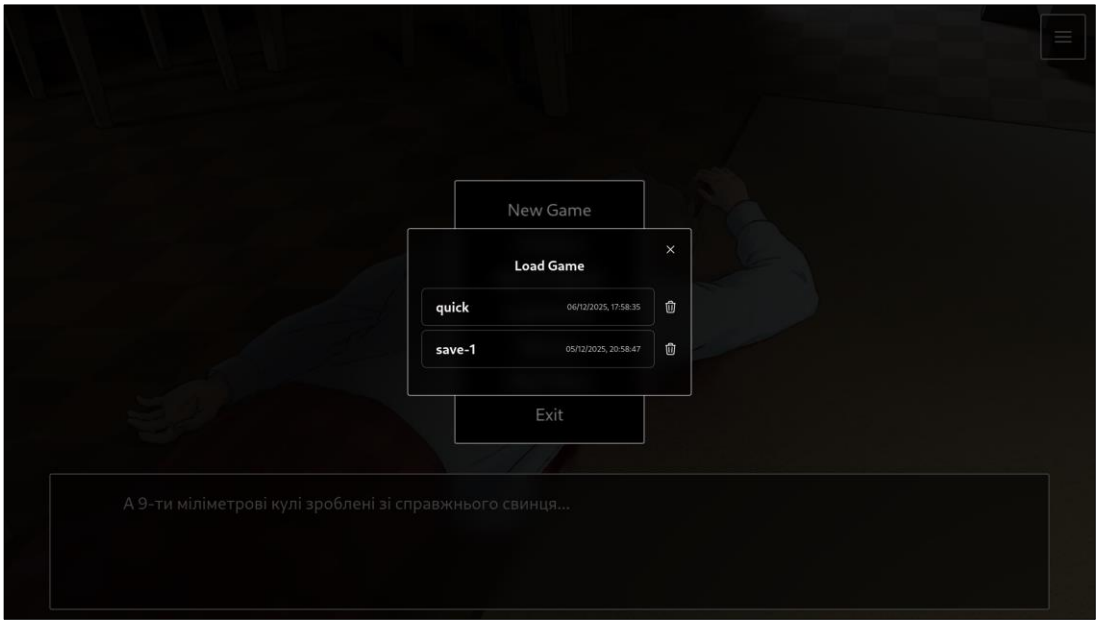


Рисунок 3.23 – Модальне вікно завантаження прогресу

Settings Modal – модальне вікно налаштувань гри. Користувач може регулювати гучність фонового звуку (BGM) та звукових ефектів (SFX) через сховище аудіо (useAudioStore), а також налаштовувати швидкість тексту через сховище загальних налаштувань (useGeneralSettingsStore). Зміни застосовуються в реальному часі, а для швидкості тексту передбачено попередній перегляд ефекту через хук useTypewriter. Дизайн модального вікна показано на рисунку 3.24.

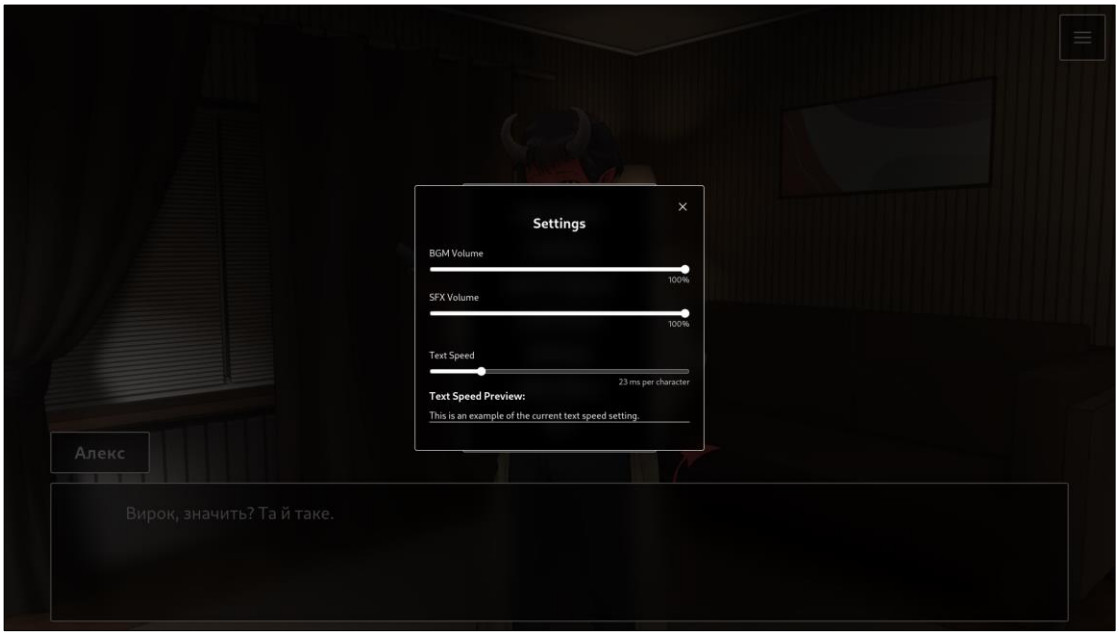


Рисунок 3.24 – Модальне вікно налаштувань

3.4.2 Реалізація системи композитного рендерингу спрайтів

Реалізація системи композитного рендерингу передбачає побудову зображення персонажа шляхом накладання окремих графічних шарів на HTML-canvas. Компонент отримує структуру CharacterLayers, що визначає актуальні шари, після чого ініціалізує контекст полотна та очищує попередній кадр.

На рисунку 3.25 наведено фрагмент коду компонента Character, що реалізує послідовне накладання шарів та рендеринг композиції на <canvas>.

```
export const Character: FC<CharacterProps> = ({ preset, width = 400, height = 600, style = {} }) => {
  const canvasRef = useRef<HTMLCanvasElement>(null)
  const loadedSprites = useSpriteStore((state) => state.loadedSprites)

  useEffect(() => {
    const canvas = canvasRef.current
    if (!canvas) return
    const ctx = canvas.getContext('2d')
    if (!ctx) return

    // Clear canvas
    ctx.clearRect(0, 0, canvas.width, canvas.height)

    // Draw layers in order
    const layerOrder: (keyof CharacterLayers)[] = ['pose', 'eyes', 'eyebrows', 'mouth']

    layerOrder.forEach((key) => {
      const sprite = preset[key] ? loadedSprites[preset[key]] : undefined
      if (sprite) {
        ctx.drawImage(sprite, 0, 0, width, height)
      }
    })
  }, [preset, loadedSprites, width, height])

  return (
    <canvas
      ref={canvasRef}
      width={width}
      height={height}
      style={{
        position: 'absolute',
        ...style
      }}
    />
  )
}
```

Рисунок 3.25 – Реалізація компонента Character, що виконує композитний рендеринг шарів спрайтів на Canvas

Графічні елементи відтворюються у строго визначеній послідовності: базова поза, очі, брови та рот. Для кожного шару здійснюється вибір

відповідного спрайта зі сховища useSpriteStore, після чого зображення відображається на полотні у встановлених розмірах. Такий підхід дає змогу формувати велику кількість комбінацій без необхідності створювати окремі цілісні спрайти.

Композитний рендеринг забезпечує модульність, гнучкість і ефективність обробки графічних ресурсів, використовуючи можливості Canvas API та попередньо завантажені растрові елементи. Така архітектура оптимізує процес відтворення персонажів і спрощує подальше розширення набору шарів.

3.5 Автоматизація процесів розробки та розгортання (CI/CD)

Для підвищення ефективності розробки та забезпечення стабільності збірок у проєкті впроваджено автоматизовані процеси CI/CD із використанням GitHub Actions [23]. Процеси охоплюють перевірку якості коду та збірку застосунку для різних операційних систем.

Для гілки dev налаштовано автоматичну перевірку при створенні pull request. На цьому етапі відбувається отримання актуальної версії репозиторію, налаштування Node.js (версія 20) із кешуванням пакетів npm, встановлення залежностей, виконання статичного аналізу коду. Такий підхід дозволяє виявляти синтаксичні помилки, порушення стилю та проблеми з типами на ранньому етапі, зменшуючи ризик помилок при інтеграції.

Для основної гілки main організовано автоматизовану збірку застосунку та створення інсталяційних пакетів для Linux і Windows. Перед збіркою виконується перевірка ресурсів та очищення кешу electron-builder для запобігання конфліктів із попередніми збірками. Далі відбувається налаштування Node.js, встановлення залежностей, виконання статичного аналізу коду. Після цього формується збірка застосунку, яка для Linux пакується за допомогою скрипту build:linux, а для Windows формується

інсталятор через build:win. Результатом процесу є артефакти збірки, що зберігаються на GitHub для подальшого тестування та розповсюдження.

Діаграму процесів CI/CD для гілок dev та main зображено на рисунку 3.26.

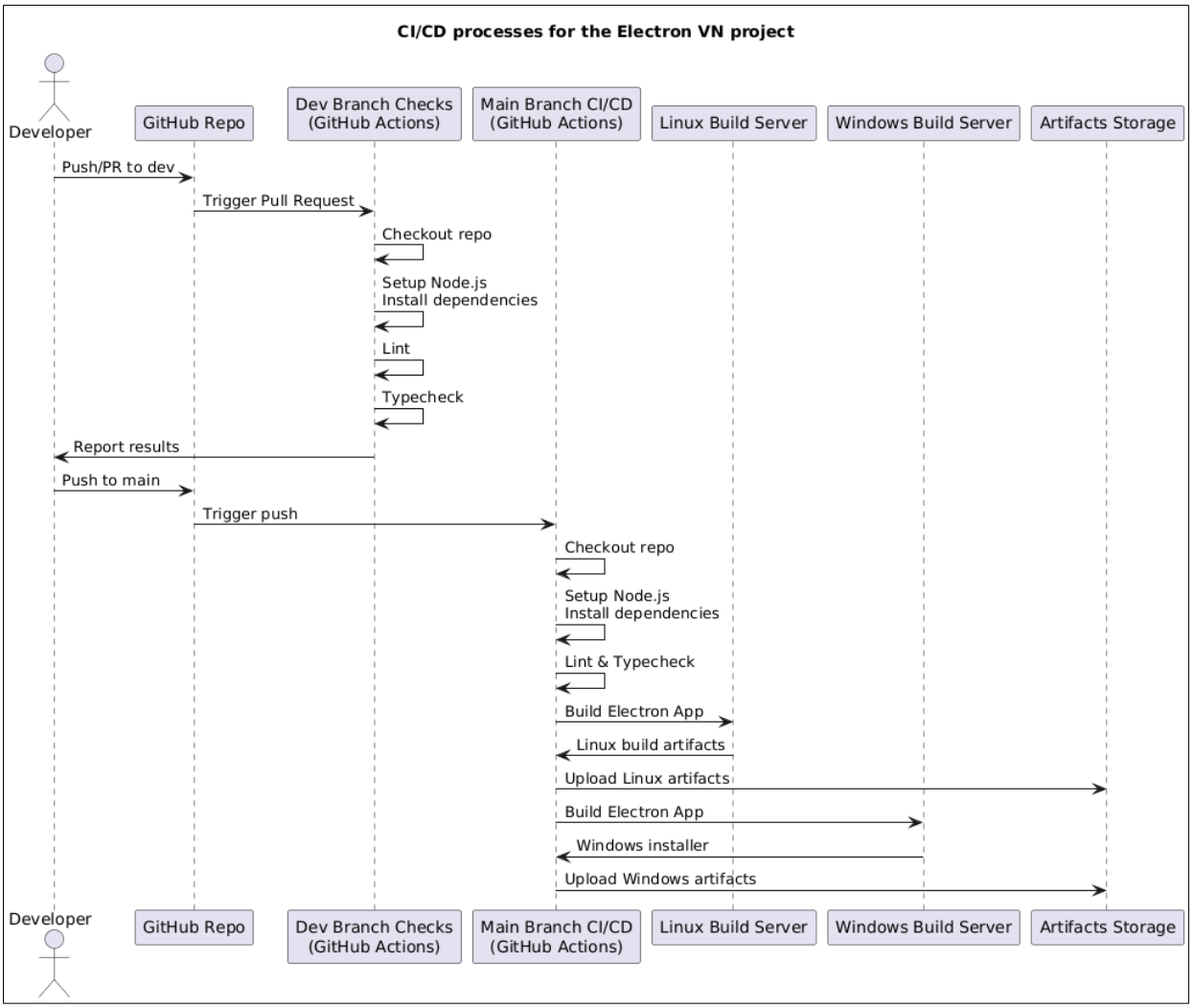


Рисунок 3.26 – Діаграма послідовності процесів CI/CD для гілок dev та main

Впровадження CI/CD дозволяє автоматично контролювати якість коду, забезпечувати стабільність збірок на різних платформах та прискорювати інтеграцію нових функцій. Таким чином, ці процеси є ключовим елементом автоматизації розробки та підтримки проєкту, підвищуючи продуктивність команди та надійність кінцевого продукту.

3.6 Висновки до розділу 3

Результатом етапу програмної реалізації стала розробка повнофункціональної кросплатформної візуальної новели на базі стеку Node.js, Electron та React. Обрана архітектура забезпечила високу продуктивність застосунку, модульність коду та можливість його подальшого масштабування.

Інфраструктуру проєкту розгорнуто з використанням інструментарію electron-vite, що дозволило налаштувати автоматизований процес збірки для Windows і Linux та підтримку HMR для швидкої розробки. Архітектурно застосунок розділено на два ізольовані контексти: головний процес (Main Process), який керує системними ресурсами, та рендер-процес (Renderer Process), відповідальний за інтерфейс. Безпечна взаємодія між ними реалізована через захищені IPC-канали, що унеможливлює прямий доступ до внутрішніх API із зовнішнього середовища.

Для роботи зі сценарієм впроваджено гібридну модель даних. Сюжетні гілки, описані у зручному JSON-форматі, під час ініціалізації трансформуються у реляційну базу даних SQLite. Такий підхід поєднав зручність редагування контенту з високою швидкістю виконання запитів (Runtime), гарантуючи миттєву реакцію системи на дії гравця. Також реалізовано функціонал збереження та відновлення ігрового прогресу.

Ядром системи виступає ігровий рушій (клас Engine), реалізований за патерном Singleton. Він централізовано керує станом гри, навігацією та обробкою нелінійного сюжету. З метою оптимізації ресурсів застосовано алгоритм «лінивого» завантаження (Lazy Loading), який підвантажує сцени динамічно, уникаючи переповнення оперативної пам’яті. Логічний блок рушія забезпечує коректну обробку умовних переходів та варіативність історії.

Клієнтську частину реалізовано на базі бібліотеки React із застосуванням оптимізованого рендерингу компонентів. Замість збереження

великих статичних зображень використано механізм композитного відмальовування персонажів через Canvas API, що дозволяє динамічно формувати образи персонажів з окремих шарів. Цей підхід суттєво зменшує об’єм графічних ресурсів. Користувацький досвід покращено завдяки попередньому завантаженню медіа-ресурсів та використанню екранів завантаження.

Процеси розробки та розгортання автоматизовано засобами CI/CD на платформі GitHub Actions. Налаштовані конвеєри виконують статичний аналіз коду, перевірку типів та автоматичну збірку інсталяційних файлів, що мінімізує ризик помилок та гарантує стабільність роботи на цільових платформах.

Таким чином, розроблений програмний продукт повністю відповідає технічним вимогам, демонструючи ефективність поєднання вебтехнологій із можливостями десктопних систем.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було проведено комплексне дослідження процесу розробки інтерактивних візуальних новел із використанням сучасних вебтехнологій та створено кросплатформений десктопний застосунок, що демонструє практичну придатність обраного технологічного стеку. На основі аналізу предметної області встановлено, що традиційні інструменти для створення візуальних новел, такі як Ren’Py або Unity, хоча й забезпечують широкий спектр можливостей, не завжди дозволяють реалізувати повністю кастомізовану логіку, сучасний UI та гнучку інтеграцію із зовнішніми системами. Саме тому у роботі було обґрунтовано застосування зв’язки React та Electron, яка поєднує переваги вебтехнологій з можливістю створення повноцінних десктопних додатків.

У процесі проєктування було розроблено архітектуру програмного забезпечення, що ґрунтується на чіткому розмежуванні відповідальностей між Main, Preload та Renderer процесами. Це забезпечило високий рівень безпеки доступу до файлової системи, модульність та передбачуваність роботи програми. Значну увагу приділено побудові власної предметно-орієнтованої мови (DSL) для опису сцен, ефектів, логічних умов, макросів та варіантів вибору. Запропонований DSL дав змогу формалізувати сценарну частину новели, зберігати її у базі даних у структурованому вигляді та відокремити логіку сценарію від реалізації інтерфейсу. Розроблена структура бази даних дозволила ефективно виконувати вибірку сцен та забезпечувати гнучке розгалуження сюжету.

У рамках реалізації було створено модуль рушія (Engine), який відповідає за виконання сценарію, обробку користувацьких виборів, застосування ефектів та навігацію між сценами. Рушій інтегровано з глобальним станом додатку, реалізованим через бібліотеку Zustand, що забезпечило передбачуваний потік даних і простоту керування прогресом.

Окремі підсистеми – менеджер спрайтів, система збережень, модуль доступу до сценарію – були реалізовані як незалежні модулі з чітко визначеними контрактами, що полегшує подальше розширення проєкту. Значну увагу приділено оптимізації продуктивності: зокрема, реалізовано lazy loading гілок сценарію, попереднє завантаження ресурсів та використання кешування. Для забезпечення якості коду та автоматизації збірки дистрибутивів під різні ОС впроваджено практики CI/CD.

Кінцевий результат – кросплатформений застосунок візуальної новели – відповідає сформованим функціональним і нефункціональним вимогам. Система забезпечує коректну роботу на основних операційних системах, дозволяє відтворювати інтерактивний сюжет із розгалуженнями, підтримує збереження прогресу, динамічне відображення персонажів, адаптивну логіку переходів та зручний користувацький інтерфейс. Отримані результати демонструють, що поєднання React та Electron є ефективним рішенням для створення ігрових застосунків, особливо у випадках, коли необхідна гнучкість сценарної частини та сучасний UI.

Практичне значення роботи полягає у створенні працездатного прототипу та архітектурної моделі, які можуть бути використані як основа для подальшого розширення, інтеграції додаткових механік або розробки повноцінних комерційних проєктів. Запропонована DSL та архітектура рушія можуть слугувати фундаментом для побудови редактора сценаріїв, інструментів авторингу або мультиплатформених інтерактивних систем. Отже, робота досягла поставленої мети та підтвердила доцільність використання вебтехнологій у сфері створення інтерактивних візуальних новел.

ПЕРЕЛІК ПОСИЛАНЬ

1. Visual Novels: 5 Characteristics of Visual Novel Video Games. URL: https://www.masterclass.com/articles/visual-novels-explained-plus-notable-visual-novel-video-games (дата звернення: 25.10.2025).

2. What is a visual novel, anyway? URL: https://tmginfo.net/2019/07/shho-take-vizualni-noveli-i-z-chogo-pochat/ (дата звернення: 25.10.2025).

3. What Are Visual Novels: Guide to Interactive Fiction | Outlook Respawn. URL: https://respawn.outlookindia.com/gaming/gaming-guides/what-are-visual-novels-a-complete-guide-to-interactive-fiction (дата звернення: 28.10.2025).

4. The Ren'Py Visual Novel Engine. URL: https://www.renpy.org/ (дата звернення: 28.10.2025).

5. Unity – офіційний сайт рушія. URL: https://unity.com/ (дата звернення: 30.10.2025).

6. Unity : Unity vs Renpy – compare differences & reviews? URL: https://www.saashub.com/compare-unity-vs-ren-py (дата звернення: 30.10.2025).

7. React – the library for web and native interfaces. URL: https://reactjs.org/ (дата звернення: 01.11.2025).

8. Electron – офіційний сайт фреймворка. URL: https://www.electronjs.org/ (дата звернення: 01.11.2025).

9. TypeScript: JavaScript With Syntax For Types. URL: https://www.typescriptlang.org/ (дата звернення: 04.11.2025).

10. Tailwind CSS – офіційний сайт фреймворку. URL: https://tailwindcss.com/ (дата звернення: 04.11.2025).

11. Introduction – Zustand. URL: https://zustand.docs.pmnd.rs/getting-started/introduction (дата звернення: 04.11.2025).

12. Process Model – Electron. URL: https://www.electronjs.org/docs/latest/tutorial/process-model (дата звернення: 05.11.2025).

13. IPC – Electron Documentation. URL: https://www.electronjs.org/docs/latest/tutorial/ipc (дата звернення: 05.11.2025).

14. DSL – Glossary. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Glossary/DSL/Domain_specific_language (дата звернення: 08.11.2025).

15. Nanoid – A tiny, secure, URL-friendly unique ID generator. URL: https://github.com/ai/nanoid (дата звернення: 08.11.2025).

16. SPA – Single-page application. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA (дата звернення: 10.11.2025).

17. Node.js – Official Website. URL: https://nodejs.org/en (дата звернення: 10.11.2025).

18. Electron-Vite – Official Website. URL: https://electron-vite.org (дата звернення: 12.11.2025).

19. Electron Toolkit – Official Repository. URL: https://github.com/alex8088/electron-toolkit (дата звернення: 12.11.2025).

20. Electron Builder – Official Repository. URL: https://github.com/electron-userland/electron-builder (дата звернення: 13.11.2025).

21. Vite. URL: https://vitejs.dev/ (дата звернення: 13.11.2025).

22. SQLite – Official Website. URL: https://www.sqlite.org (дата звернення: 14.11.2025).

23. GitHub Actions documentation. URL: https://docs.github.com/en/actions (дата звернення: 14.11.2025).