ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка відеогри в жанрі roguelike deck-building на рушії godot engine»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Андрій ЯСІНСЬКИЙ
(підпис)

Виконав: здобувач вищої освіти групи ПД-43
Андрій ЯСІНСЬКИЙ

Керівник: Олександр ЯРОШЕВСЬКИЙ
асистент кафедри ІПЗ

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Ясінському Андрію Борисовичу

1. Тема кваліфікаційної роботи: «Розробка відеогри в жанрі roguelike deck-building на рушії godot engine»
керівник кваліфікаційної роботи асистент кафедри ІПЗ Олександр ЯРОШЕВСЬКИЙ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про рушій Godot Engine, опис розробки відеогри в жанрі roguelike deck-building, документація Godot Engine.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд застосунку для розробки відеогри в жанрі roguelike deck-building та аналіз існуючих аналогів, визначення вимог до програмного забезпечення.
2. Огляд та аналіз засобів реалізації застосунку для розробки відеогри в жанрі roguelike deck-building.

3. Проєктування архітектури застосунку для розробки відеогри в жа
roguelike deck-building.
4. Програмна реалізація та тестування застосунку для розробки відеогри в
жанрі roguelike deck-building.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Діаграма класів.
6. Діаграма послідовності.
7. Архітектура проєкту.
8. Екранні форми.
9. Демонстрація гри.
10. Апробація результатів дослідження.

6. Дата видачі завдання«25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Огляд застосунку для розробки відеогри в жанрі roguelike deck-building та аналіз існуючих аналогів, визначення вимог до програмного забезпечення.	13.03-15.03.2024	
4	Огляд та аналіз засобів реалізації застосунку для розробки відеогри в жанрі roguelike deck-building.	16.03-21.03.2024	
5	Проєктування архітектури застосунку для розробки відеогри в жанрі roguelike deck-building.	22.03-03.04.2024	
6	Програмна реалізація та тестування застосунку для розробки відеогри в жанрі roguelike deck-building.	04.04-28.04.2024	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Андрій ЯСІНСЬКИЙ
(підпис)

Керівник
кваліфікаційної роботи _____ Олександр ЯРОШЕВСЬКИЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 81 стор., 1 табл., 18 рис., 20 джерел.

Мета роботи – створення відеогри в жанрі roguelike deck-building на рушії Godot Engine із впровадженням ключових механік жанру та демонстрацією можливостей розробки ігор для навчальних цілей.

Об’єкт дослідження – процес розробки відеогри в жанрі roguelike deck-building.

Предмет дослідження програмне забезпечення для створення відеогри в жанрі roguelike deck-building на базі Godot Engine.

Короткий зміст роботи: В роботі проаналізовано особливості жанру roguelike deck-building на прикладі його аналогів. Проведено огляд інструментів для розробки ігор, зокрема Godot Engine, GDScript та GUT для тестування. Розроблено архітектуру гри, а також програмно реалізовані архітектуру гри, а також програмно реалізовані основні механіки: система карт, бойова система, випадкова генерація вмісту, збереження прогресу та інтерфейс користувача. Проведено модульне, інтеграційне та системне тестування гри для забезпечення її стабільності. В роботі використано рушій Godot Engine, мову програмування GDScript, систему сигналів Godot для взаємодії компонентів та фреймворк GUT для тестування.

Сферою використання застосунку є навчання розробці ігор на Godot Engine та створення прототипів у жанрі roguelike deck-building.

КЛЮЧОВІ СЛОВА: ROGUELIKE, DECK-BUILDING, GODOT ENGINE, GAMESCRIPT, ВИПАДКОВА ГЕНЕРАЦІЯ, СИСТЕМА КАРТ

ЗМІСТ

ВСТУП

Актуальність: Розробка відеогри в жанрі roguelike deck-building на рушії Godot Engine є надзвичайно актуальною темою в контексті сучасної індустрії геймдеву. Цей жанр, який поєднує в собі елементи стратегічного планування, випадковості та високої реграбельності, здобув широку популярність серед гравців завдяки таким відомим проєктам, як Slay the Spire, Monster Train, Griftlands і Inscryption. Ці ігри демонструють, як унікальні механіки, засновані на управлінні колодою карт і процедурній генерації вмісту, можуть створювати глибокий і захопливий ігровий досвід, що спонукає гравців повертатися до гри неодноразово. У той же час, Godot Engine, як відкритий і гнучкий рушій із потужною підтримкою 2D-графіки, стає дедалі популярнішим серед розробників завдяки своїй доступності, легкості освоєння та відсутності необхідності у значних фінансових вкладеннях. Цей інструмент дозволяє створювати інноваційні проєкти, адаптовані до потреб сучасного ринку інтерактивних розваг.

Реалізація подібного проєкту має не лише практичне значення, але й сприяє вдосконаленню професійних навичок розробника у таких ключових сферах, як програмування, дизайн ігрових систем, оптимізація продуктивності та управління проєктами. Індустрія геймдеву сьогодні є однією з найбільш динамічних і прибуткових галузей, а зростання інтересу до ігор із процедурною генерацією та стратегічними елементами лише підкреслює важливість таких розробок. Доступність інструментів, таких як Godot Engine, відкриває нові можливості для незалежних розробників і невеликих команд, дозволяючи їм конкурувати з великими студіями. Таким чином, створення гри в жанрі roguelike deck-building на Godot Engine не лише відповідає сучасним тенденціям, але й має потенціал для внеску в розвиток індустрії інтерактивних розваг.

Об’єкт і предмет дослідження є процес розробки відеогри в жанрі roguelike deck-building, що охоплює планування, програмування, тестування та оптимізацію.

Предметом дослідження є створення гри на рушії Godot Engine, включаючи реалізацію механік (випадкова генерація, колода карт, бойові дії), використання GDScript і системи збереження гри. Особлива увага приділяється адаптації Godot для складних ігрових систем.

Метою роботи є створення функціональної гри в жанрі roguelike deck-building на Godot Engine з ключовими механіками: випадковою генерацією вмісту, системою карт і стратегічним плануванням. Проект демонструє навички автора у використанні Godot, управлінні проектом і впровадженні рішень, таких як генерація на основі насіння для повторюваності сесій. Робота поєднує практичну розробку з аналізом процесу створення гри.

Методи дослідження: першим кроком було проаналізовано представників жанру для визначення ключових механік і вимог до проекту. Далі було проведено розробка архітектури гри з основними компонентами: бій, колода карт, інтерфейс. Було також реалізовано механіки за допомогою GDScript, включаючи перетягування карт і логіку магазину. А також було проведене модульне інтеграційне тестування для забезпечення якості гри.

Наукова новизна роботи визначається у впровадженні системи збереження гри з використанням ресурсів Godot і генератора випадкових чисел на основі насіння для повторюваності сесій. Оптимізація гри для низької роздільної здатності (256x144 пікселів) із масштабуванням демонструє нові підходи до використання Godot Engine, що може бути корисним для подальших розробок.

Практична значущість результатів полягає в створенню гри з використан навчальних матеріалів для розробників, які вивчають Godot Engine і жанр roguelike deck-building. Проект описує процес розробки, від налаштування до реалізації механік, і може слугувати основою для розширення, наприклад, додавання нових персонажів чи рівнів. Робота сприяє популяризації відкритих інструментів і розвитку індустрії геймдеву.

1 АНАЛІЗ ЗАСТОСУНКУ ДЛЯ РОЗРОБКИ ВІДЕОГРИ В ЖАНРІ ROGUELIKE DECK-BUILDING

1.1 Визначення жанру roguelike deck-building

Жанр roguelike deck-building є інноваційним гібридом двох самобутніх ігрових стилів — roguelike та deck-building, які разом створюють унікальний ігровий досвід, що поєднує випадковість, стратегічне планування та високу реграбельність. Цей жанр приваблює гравців своєю здатністю пропонувати свіжі виклики у кожній сесії, одночасно дозволяючи розвивати глибокі тактичні підходи через управління колодою карт.

— Roguelike: Цей жанр бере свій початок від гри Rogue (1980), яка запровадила ключові принципи: випадкову генерацію вмісту (рівнів, ворогів, предметів) та механіку "перманентної смерті" (permadeath). У разі смерті персонажа гравець втрачає весь прогрес і починає гру заново. Такий підхід забезпечує високу реграбельність і робить кожну сесію непередбачуваною та унікальною.

— Deck-building: Цей жанр став популярним завдяки настільній грі Dominion (2008). Він базується на створенні та вдосконаленні колоди карт протягом гри. Гравець починає з базового набору карт і поступово додає нові — з ефектами атаки, захисту чи спеціальних здібностей, — щоб досягти перемоги.

Поєднання цих жанрів створює унікальний ігровий досвід: гравець проходить випадково згенеровані рівні, використовуючи карти зі своєї колоди для боротьби з ворогами, подолання перешкод і розвитку стратегії. Колода розширюється та вдосконалюється в процесі гри, додаючи глибини й варіативності. Приклади, такі як Slay the Spire чи Monster Train, показують, як випадковість roguelike гармонійно доповнює стратегічне планування deck-building, роблячи кожну сесію свіжою та захоплюючою.

1.2 Особливості розробки на рушії Godot Engine

Для створення гри в жанрі roguelike deck-building було обрано Godot Engine — потужний і відкритий рушій для розробки ігор, який підтримує як 2D, так і 3D проекти. Його особливості роблять його ідеальним вибором для цього типу гри:

— Система сцен і вузлів: Godot використовує модульну архітектуру, де гра складається зі сцен, а сцени — з вузлів (спрайти, кнопки, скрипти тощо). Це дозволяє легко створювати та комбінувати ігрові елементи, такі як карти, вороги чи елементи інтерфейсу, що є критично важливим для ігор із великою кількістю об’єктів.

— GDScript: Мова програмування Godot, схожа на Python, інтуїтивна й зручна для швидкої реалізації логіки гри — від бойової системи до генерації рівнів.

— Підтримка 2D: Оскільки проєкт орієнтований на 2D із піксельною графікою, Godot пропонує інструменти для роботи зі спрайтами, анімаціями та рендерингом у низькій роздільній здатності, що забезпечує чіткий ретро-стиль.

— Гнучкість ресурсів: Рушій дозволяє легко зберігати та завантажувати дані (наприклад, колоди карт, характеристики ворогів), що спрощує створення системи прогресії.

— Інструменти для інтерфейсу: Вбудована система UI полегшує розробку меню, відображення колоди та статистики гравця.Анімації: Godot надає потужні інструменти для створення динамічних ефектів, таких як анімації боїв чи ефекти карт.

1.3 Дослідження існуючих аналогів

Аналіз існуючих ігор у жанрі roguelike deck-building дозволяє виявити ключові механіки, сильні сторони та потенційні недоліки, які можна врахувати при розробці власного проекту.

1.3.1 Slay the Spire

Slay the Spire — це еталонна гра в жанрі roguelike deck-building, де гравець обирає персонажа і проходить вежу, борючись із ворогами за допомогою карт.

— Механіки: Випадково згенеровані рівні, широкий вибір карт із різними ефектами, система реліквій для додаткових бонусів.

— Сильні сторони: Висока реграбельність, глибока стратегічна складова, простий і зрозумілий інтерфейс.

— Слабкі сторони: Висока складність для новачків, обмежена кількість персонажів може зменшувати різноманітність.

— Унікальні особливості: Реліквії суттєво впливають на стратегію, додаючи бонуси чи змінюючи механіки.

Рис. 1.1 Приклад відеогри Slay the Spire

1.3.2 Monster Train

Monster Train додає унікальний поворот до жанру, де гравець захищає потяг від ворогів, розміщуючи юнітів і заклинання на кількох поверхах.

— Механіки: Управління поверхами потягу, різні клани з унікальними картами, випадкові події.

— Сильні сторони: Інноваційна механіка з поверхами, велика різноманітність стратегій.

— Слабкі сторони: Складність може відлякувати новачків, потреба в балансуванні великої кількості механік.

— Унікальні особливості: Клани пропонують різні стилі гри, а багатоповерховість додає тактичну глибину.

Рис. 1.2 Приклад відеогри Monster Train

1.3.3 Griftlands

Griftlands вирізняється поєднанням roguelike deck-building із RPG елементами, де карти використовуються як для бою, так і для переговорів.

— Механіки: Дві колоди (бойова та переговорна), розгалужений сюжет із виборами.

— Сильні сторони: Унікальний підхід до переговорів, сильна сюжетна складова.

— Слабкі сторони: Менший акцент на класичному deck-building, сюжет може знижувати реграбельність.

— Унікальні особливості: Переговори дозволяють уникати конфліктів або отримувати переваги, додаючи рольову варіативність.

Рис. 1.3 Приклад відеогри Griftlands

1.3.4 Inscryption

Inscryption додає до жанру елементи жахів і головоломок, створюючи атмосферний і незвичний досвід.

— Механіки: Колода на основі жертвування істот, унікальна система ресурсів (кров і кістки), випадкові події та головоломки.

— Сильні сторони: Атмосферний дизайн, інноваційна механіка жертвування, захоплюючий сюжет.

— Слабкі сторони: Менший акцент на стратегії колоди через фокус на сюжеті та головоломках.

— Унікальні особливості: Жертвування як основа механіки додає напруження й тактичного вибору.

Рис. 1.4 Приклад відеогри Inscryption

Таблиця 1.1

Зведена таблиця аналізу існуючих відеоігор у жанрі roguelike deck-building

Параметр	**Slay the Spire**	**Monster Train**	**Griftlands**	**Inscryption**
Основні механіки	Випадкові рівні, система реліквій, бойові карти	Управління поверхами, клани, випадкові події	Дві колоди (бойова і переговорна), розгалужений сюжет	Жертвування істот, головоломки, випадкові події
Сильні сторони	Висока реграбельність, глибока стратегія, простий інтерфейс	Інноваційна механіка поверхів, різноманітність стратегій	Унікальний підхід до переговорів, сильна сюжетна складова	Атмосфера, інноваційна механіка, сюжет

Продовження таблиці 1.1

Зведена таблиця аналізу існуючих відеоігор у жанрі roguelike deck-building

Параметр	**Slay the Spire**	**Monster Train**	**Griftlands**	**Inscryption**
Слабкі сторони	Висока складність для новачків, обмежена кількість персонажів	Складність, потреба в балансуванні механік	Менший акцент на deck-building, сюжет знижує реграбельність	Менший акцент на стратегії колоди
Унікальні особливості	Реліквії для бонусів, стратегічне планування колоди	Багатоповерхова оборона, клани з унікальними картами	Переговори як альтернатива бою, RPG елементи	Жертвування, система ресурсів
Графічний стиль	2D, мальована графіка	2D, фентезійний стиль	2D, комікcний стиль	2D, 3D, стиль жахів
Складність	Висока	Середня до високої	Середня	Середня
Рівень реграбельності	Високий	Високий	Середній	Середній

1.4 Визначення вимог до гри

На основі дослідження жанру, аналогів і можливостей Godot Engine сформульовано такі вимоги до гри:

— Випадкова генерація: Система створення унікальних рівнів, ворогів, подій і карт у кожній сесії. Генеруватимуться лабіринти, вороги з унікальними здібностями та випадкові події (скарби, пастки), що впливатимуть на реграбельність.

— Система колоди карт: Колода з картами різних типів (атака, захист, підтримка). Гравець отримуватиме нові карти як нагороди за перемоги чи події, зможе покращувати їх або видаляти для оптимізації стратегії.

— Бойова система: Покроковий бій на основі карт, де гравець використовує колоду проти ворогів. Перемога визначатиметься знищенням ворогів, поразка — смертю гравця.

— Прогресія: Можливість додавати нові карти, покращувати існуючі (збільшення сили, зменшення вартості) та видаляти слабкі. Додаватимуться реліквії для пасивних ефектів.

— Інтерфейс: Зручний UI для управління колодою, перегляду статистики (здоров’я, мана), інформації про ворогів і стан бою.

— Збереження прогресу: Система збереження/завантаження стану гри для комфортного проходження.

— Графіка та звук: Піксельна графіка в ретро-стилі, звукові ефекти для боїв, карт і фонової музики.

— Технічні аспекти: Роздільна здатність 256x144 пікселів, оптимізація для піксельної графіки, масштабування для великих екранів, підтримка ПК, мобільних платформ, різних видів контролерів керування, підтримка грального консолів і стабільна робота.

1.5 Цільова аудиторія

Гра орієнтована на таких гравців:

— Шанувальники roguelike: Ті, хто любить випадкову генерацію та перманентну смерть.

— Шанувальники deck-building: Гравці, які насолоджуються створенням і оптимізацією колод.

— Любителі викликів: Ті, хто шукає складні ігри з необхідністю адаптації та стратегічного мислення.

— Прихильники ретро-стилю: Гравці, які цінують піксельну графіку та ретро-естетику.

1.6 Технічні вимоги до розробки

— Godot Engine 4.0: Остання версія для підтримки всіх 2D-функцій.

— Плагіни: Можливе використання плагінів для генерації рівнів чи анімацій.

— Інструменти для графіки: Aseprite або Piskel для створення піксельних спрайтів.

— Інструменти для звуку: Audacity або FL Studio для звукових ефектів і музики.

Аналіз аналогів і вимог до відеогри в жанрі roguelike deck-building свідчить, що розроблений проєкт вирізнятиметься завдяки поєднанню глибокої стратегічної складової, високої реграбельності, інтуїтивно зрозумілого користувацького інтерфейсу та унікальних дизайнерських рішень, які забезпечать його конкурентоспроможність на ринку інтерактивних розваг. Ключовими особливостями гри стануть інноваційні ігрові механіки, такі як адаптивна система процедурної генерації вмісту на основі насіння для створення повторюваних, але різноманітних ігрових сесій.

2 ЗАСОБИ РЕАЛІЗАЦІЇ

2.1 Рушій гри та середовище розробки

У цьому розділі детально описано засоби, які використовувалися для створення відеогри в жанрі roguelike deck-building на базі рушія Godot Engine. До них належать рушій гри та середовище розробки, мова програмування, система контролю версій, а також інструменти для створення графічних і звукових ресурсів. Ці засоби разом забезпечили ефективну реалізацію проекту, дозволивши оптимізувати робочий процес, зосередитися на творчих аспектах і досягти поставлених цілей.

Godot Engine — це відкритий рушій із широкими можливостями для розробки 2D- та 3D-ігор, який пропонує повний набір інструментів для створення ігрових проєктів. Він включає візуальний редактор, систему скриптів, вбудовані вузли для реалізації типових ігрових механік, а також підтримку анімацій, фізики, аудіо та інших ключових аспектів розробки. Завдяки своїй легкості, гнучкості та адаптивності Godot Engine ідеально підійшов для створення гри в жанрі roguelike deck-building, де потрібна швидка ітерація та модульна структура.

2.1.1 Система вузлів і сцен

Основою архітектури Godot Engine є система вузлів (node system), яка дозволяє створювати всі ігрові об’єкти як комбінацію базових будівельних блоків. Кожен вузол має свій тип і набір властивостей, що визначають його функціональність. Наприклад, у цьому проекті для реалізації карт використовувалися:

— Вузли типу Sprite2D — для відображення графічних зображень карт.

— Вузли типу Area2D — для обробки взаємодії з картами через мишу або сенсорний екран.

Вузли можна об’єднувати в ієрархічні структури, що спрощує створення складних об’єктів. Наприклад, ігровий персонаж складався з:

— Вузла KinematicBody2D — для управління рухом.

— Вузла AnimatedSprite2D — для відтворення анімацій.

— Вузла CollisionShape2D — для обробки зіткнень.

Такий підхід не лише полегшує організацію об’єктів, але й сприяє повторному використанню коду.

Ще однією ключовою особливістю є **система сцен (scene system)**, яка дозволяє групувати вузли в модульні одиниці — сцени. Сцени зберігаються як окремі файли й можуть повторно використовуватися в різних частинах гри. У цьому проекті:

— Кожна карта була реалізована як окрема сцена, що дало змогу легко додавати нові карти та редагувати їх без впливу на інші елементи.

— Рівні гри створювалися як сцени з вузлами для фону, ворогів і подій, що спростило процедурну генерацію.

2.1.2 Редактор Godot

Редактор Godot виступає основним інтерфейсом для роботи з рушієм. Він об'єднує в собі редактор сцен, редактор скриптів, дебагер та інші інструменти, необхідні для створення ігор. Редактор дозволяє візуально компонувати сцени з вузлів, налаштовувати їхні властивості та зв’язки, а також писати й відлагоджувати скрипти безпосередньо в інтегрованому середовищі. Завдяки цьому розробка відбувалася в єдиному просторі без потреби в додаткових зовнішніх програмах.

У проекті редактор Godot був використаний для:

Створення та налаштування сцен: наприклад, сцени для карт, персонажів, інтерфейсу користувача та рівнів.

Налаштування анімацій: за допомогою вбудованого редактора анімацій було створено анімації для переміщення карт, бойових ефектів і переходів між сценами.

Управління ресурсами: редактор дозволяв імпортувати та організовувати графічні та аудіоресурси, а також налаштовувати їхні параметри для оптимального використання в грі.

Налагодження: вбудований дебагер дозволяв відстежувати помилки та перевіряти значення змінних у реальному часі, що значно прискорило процес розробки.

2.1.3 Переваги Godot Engine для проекту

Вибір Godot Engine для розробки гри в жанрі roguelike deck-building був обґрунтований рядом переваг:

Відкритий код і безкоштовність: Godot є повністю безкоштовним і відкритим, що дозволило уникнути ліцензійних витрат і дало можливість адаптувати рушій під потреби проекту, якщо це було б необхідно.

Підтримка 2D-ігор: Godot має потужні інструменти для розробки 2D-ігор, що ідеально підходить для створення карткової гри з елементами roguelike.

Гнучкість і модульність: система вузлів і сцен дозволяє легко розширювати гру, додавати нові механіки та змінювати існуючі без значних переробок коду.

Активна спільнота: велика спільнота розробників Godot надає численні навчальні матеріали, плагіни та приклади проектів, що допомогло швидко вирішувати технічні питання під час розробки.

2.2 Мови програмування

Для написання скриптів у Godot Engine використовувалася GDScript — високорівнева, динамічно типізована мова програмування, спеціально розроблена для цього рушія. GDScript має синтаксис, подібний до Python, що робить його легким для вивчення та використання навіть для розробників із середнім рівнем досвіду. Ця мова тісно інтегрована з Godot Engine, що дозволяє швидко й ефективно взаємодіяти з вузлами, сценами та іншими об’єктами рушія,

забезпечуючи високу продуктивність і зручність у реалізації складних ігрових механік.

2.2.1 Використання GDScript у проекті

У проекті всі скрипти були написані на GDScript, що дозволило реалізувати ключові ігрові механіки, характерні для жанру roguelike deck-building. До таких механік належать:

Перетягування карт: Реалізація інтерактивного інтерфейсу для вибору та переміщення карт гравцем.

Управління боєм: Логіка бойової системи, що включає обробку атак, захистів і спеціальних ефектів карт.

Генерація випадкових подій: Процедурна генерація подій і рівнів для забезпечення унікальності кожного проходження.

Обробка інтерфейсу користувача: Управління елементами UI, такими як відображення колоди, руки гравця та статусу ворогів.

Завдяки тісній інтеграції GDScript із Godot Engine вдалося досягти високої продуктивності та мінімальних зусиль на адаптацію коду. Наприклад, GDScript дозволяє безпосередньо звертатися до вузлів і їхніх властивостей, що значно спрощує написання коду для динамічних ігрових об’єктів.

Приклад реалізації механіки перетягування карт:

```
extends Area2D
var is_dragging = false
var original_position = Vector2()
func _ready():
    original_position = global_position
func _input(event):
    if event is InputEventMouseButton and event.button_index == BUTTON_LEFT:
```

```
            if event.pressed and
get_global_mouse_position().distance_to(global_position) < 50:
                is_dragging = true
            else:
                is_dragging = false
                global_position = original_position
        elif event is InputEventMouseMotion and is_dragging:
            global_position = get_global_mouse_position()
```

Цей скрипт прикріплюється до вузла Area2D, який представляє карту в грі. Він дозволяє гравцеві перетягувати карту за допомогою миші, повертаючи її на початкову позицію, якщо перетягування скасовано. Використання вбудованих функцій Godot, таких як get_global_mouse_position(), забезпечує простоту реалізації таких механік.

Приклад реалізації бойової системи:

```
extends Node
var player_health = 100
var enemy_health = 50
func apply_card_effect(card_type, value):
    if card_type == "attack":
        enemy_health -= value
        print("Enemy takes ", value, " damage. Enemy health: ", enemy_health)
    elif card_type == "defend":
        player_health += value
        print("Player gains ", value, " health. Player health: ", player_health)
func _process(delta):
    if enemy_health <= 0:
        print("Enemy defeated!")
        queue_free()
```

Цей скрипт демонструє базову логіку бойової системи, де карта типу "attack" завдає шкоди ворогу, а карта типу "defend" додає здоров'я гравцеві. Використання функції _process дозволяє перевіряти стан ворога в реальному часі, що є важливим для динамічних ігор.

Приклад процедурної генерації подій:

```
extends Node
var events = ["encounter_enemy", "find_treasure", "rest_area", "boss_fight"]
func generate_event():
    var random_index = randi() % events.size()
    var selected_event = events[random_index]
    print("Generated event: ", selected_event)
    return selected_event
```

Цей скрипт генерує випадкову подію з масиву можливих подій, що є основою для створення унікальних проходжень у жанрі roguelike. Використання функції randi() забезпечує швидке створення випадкових чисел, що є критично важливим для процедурної генерації.

2.2.2 Переваги GDScript

Вибір GDScript як основної мови програмування для проекту "Розробка відеогри в жанрі roguelike deck-building на рушії Godot Engine" був обґрунтований низкою ключових переваг, які зробили його ідеальним інструментом для реалізації задуманого.

Простота синтаксису: GDScript має синтаксис, подібний до Python, що робить його інтуїтивно зрозумілим і легким для освоєння навіть для розробників із обмеженим досвідом. Ця простота дозволяє швидко писати, читати та редагувати код, що особливо важливо на ранніх етапах розробки, коли необхідно часто змінювати та тестувати механіки гри. Наприклад, логіка перетягування карт у руці гравця або активація їхніх ефектів може бути реалізована лише кількома рядками

коду, що зменшує ймовірність помилок і полегшує підтримку проєкту. Завдяки цьому розробник може зосередитися на творчих аспектах, а не на боротьбі зі складними конструкціями.

Тісна інтеграція з Godot Engine: GDScript створений спеціально для роботи з вузлами та сценами Godot, що забезпечує безшовну взаємодію з об'єктами рушія. Це значно зменшує кількість допоміжного коду, який зазвичай потрібен для доступу до ігрових об'єктів у інших мовах програмування. Наприклад, метод get_node() дозволяє легко отримувати доступ до будь-якого вузла в сцені, що спрощує маніпуляції з ігровими елементами, такими як карти, вороги чи елементи інтерфейсу. Крім того, GDScript підтримує використання сигналів (signals), що є основою подієво-орієнтованого програмування в Godot. Наприклад, сигнал може бути використаний для обробки натискання на карту або завершення анімації атаки ворога, що робить код більш структурованим і легким для розуміння.

Швидкість розробки: Завдяки динамічній типізації та відсутності необхідності компіляції, GDScript дозволяє миттєво тестувати зміни в коді. Це критично важливо на етапі прототипування, коли розробник часто експериментує з різними варіантами механік. Наприклад, під час балансування бойової системи можна швидко змінити значення шкоди чи захисту карт і одразу перевірити результат у грі. Функція "гарячого перезавантаження" (hot reload) у Godot додатково прискорює цей процес, дозволяючи вносити зміни в скрипти без перезапуску гри, що економить час і підвищує продуктивність.

Оптимізація для 2D-ігор: GDScript забезпечує достатню продуктивність для більшості 2D-ігор, таких як roguelike deck-building, де обчислювальні навантаження зазвичай не є критичними. У цьому жанрі основний акцент робиться на логіку гри — наприклад, процедурну генерацію колоди чи обробку взаємодій між картами, — а не на складні 3D-обчислення чи рендеринг. GDScript, оптимізований для роботи з Godot Engine, забезпечує швидке виконання коду та мінімальні затримки навіть у сценах із великою кількістю об'єктів, таких як десятки карт на екрані чи динамічні ефекти бою.

Ці переваги зробили GDScript незамінним інструментом для проєкту, дозволивши зосередитися на реалізації ігрових механік і творчих ідей, а не на технічних складнощах..

2.2.3 Порівняння з іншими мовами програмування

Godot Engine підтримує кілька мов програмування, зокрема C# і VisualScript, які розглядалися як потенційні альтернативи GDScript. Однак після аналізу їхніх особливостей вибір зупинився на GDScript через його оптимальне поєднання простоти, швидкості розробки та інтеграції з рушієм.

C#: Ця мова є потужною альтернативою для проєктів, що вимагають високої продуктивності або інтеграції з платформами .NET. Завдяки статичній типізації C# може допомагати виявляти помилки ще на етапі компіляції, а також підтримує складні об'єктно-орієнтовані конструкції. Проте для цього проєкту C# не був обраний з кількох причин:

— Складність налаштування: Використання C# у Godot вимагає встановлення Mono SDK і додаткових інструментів, що ускладнює початкове налаштування робочого середовища.

— Громіздкий синтаксис: Порівняно з GDScript, C# потребує більше коду для виконання базових операцій, що може уповільнити розробку, особливо для невеликих проєктів або розробників із меншим досвідом.

— Продуктивність: Хоча C# може бути швидшим у сценаріях із високими обчислювальними навантаженнями, для 2D-гри типу roguelike deck-building продуктивність GDScript є більш ніж достатньою, тому переваги C# не виправдовують додаткових зусиль.

VisualScript: Ця мова дозволяє створювати логіку гри за допомогою візуальних вузлів, що може бути корисним для розробників без досвіду програмування або для швидкого створення простих прототипів. Однак для цього проєкту VisualScript виявився менш практичним:

— Обмежена гнучкість: Візуальне програмування ускладнює реалізацію складних алгоритмів, таких як процедурна генерація колоди карт чи управління бойовою системою з великою кількістю змінних і умов.

— Складність масштабування: Із збільшенням обсягу коду візуальні схеми стають громіздкими, що ускладнює їхнє редагування та підтримку.

— Продуктивність: VisualScript може бути менш ефективним у плані швидкості виконання порівняно з текстовими мовами, що є важливим для ігор із динамічними механіками.

GDScript був обраний завдяки своїй простоті, швидкості розробки та глибокій інтеграції з Godot Engine. Це дозволило уникнути технічних ускладнень і зосередитися на створенні ігрових механік, таких як управління колодою чи бойова система.

2.2.4 Налагодження та тестування

GDScript підтримує ефективне налагодження завдяки вбудованим інструментам Godot Engine, що значно полегшує процес розробки та тестування гри. Редактор Godot включає повнофункціональний дебагер із такими можливостями:

— Точки зупинки (breakpoints): Розробник може зупиняти виконання коду в певних місцях, щоб перевірити стан змінних або логіку програми.

— Перегляд значень у реальному часі: Дебагер відображає поточні значення всіх змінних у сцені, що дозволяє відстежувати зміни під час гри.

— Покрокове виконання: Можливість виконувати код рядок за рядком допомагає детально аналізувати поведінку програми та виявляти помилки.

Наприклад, під час тестування бойової системи дебагер використовувався для перевірки коректності обчислень шкоди та здоров’я. Розробник міг установити точку зупинки в методі, що застосовує ефект карти, і переглянути, як змінюються значення player_health і enemy_health після кожної дії. Це допомогло швидко

виявити помилки, наприклад, неправильне застосування модифікаторів чи некоректне округлення значень.

Функція "гарячого перезавантаження" (hot reload) у Godot дозволяла вносити зміни в скрипти GDScript і бачити їхні результати без перезапуску гри. Це прискорило ітеративний процес розробки, дозволяючи експериментувати з різними варіантами механік. Наприклад, під час балансування ефектів карт розробник міг змінити значення шкоди чи захисту прямо під час гри й одразу оцінити вплив на бій.

Також Godot надає інструменти для профілювання, які допомагають виявляти вузькі місця в продуктивності. Це було корисно для оптимізації коду в сценах із великою кількістю карт чи ефектів, дозволяючи переконатися, що гра працює плавно.

2.2.5 Оптимізація коду

Для забезпечення високої продуктивності гри в GDScript було використано кілька підходів до оптимізації коду, що було особливо важливим для сцен із великою кількістю об’єктів, таких як карти чи бойові ефекти.

Мінімізація звернень до вузлів: Замість повторного виклику get_node() у кожному кадрі, що є витратною операцією, посилання на часто використовувані вузли зберігалися в змінних під час ініціалізації сцени. Наприклад, у скрипті для управління колодою карт посилання на вузол руки гравця зберігалося в змінній у методі _ready(), що уникало повторних пошуків.

Використання пулів об’єктів: Для часто створюваних і видаляємих об’єктів, таких як карти чи візуальні ефекти, використовувалися пули об’єктів. Це дозволяло повторно використовувати інстанси замість створення нових, зменшуючи навантаження на збірник сміття. Наприклад, для карт у руці гравця був створений пул із фіксованою кількістю інстансів, які активувалися та деактивувалися за потреби.

Обмеження оновлень: Логіка, яка не потребувала виконання в кожному кадрі, переносилася з функції _process до подієво-орієнтованих методів. Наприклад,

оновлення інтерфейсу здоров'я відбувалося лише при зміні значення, а не щокадру, що зменшувало кількість обчислень.

Оптимізація рендерингу: Використовувалися техніки, такі як batching спрайтів і atlas текстур, для зменшення навантаження на графічний процесор. Це дозволяло рендерити кілька спрайтів за один виклик, що було важливим для сцен із великою кількістю карт.

2.3 Інструменти для створення ресурсів

Для створення ресурсів гри, таких як графіка, анімації, звукові ефекти та музика, використовувалися спеціалізовані інструменти, які забезпечили високу якість контенту та ефективну інтеграцію з Godot Engine. Основними інструментами стали Aseprite для створення графічних ресурсів і Audacity для роботи зі звуком. Додатково застосовувалися Tiled для створення карт рівнів і GIMP для обробки зображень. Ці інструменти обрано через їхню доступність, простоту використання та сумісність із Godot Engine, що дозволило зосередитися на творчому процесі, уникаючи технічних ускладнень.

2.3.1 Aseprite для створення графічних ресурсів

Aseprite — це спеціалізований інструмент для створення піксель-арту, популярний серед розробників 2D-ігор завдяки зручному інтерфейсу для малювання, анімації та експорту спрайтів. У грі в жанрі roguelike deck-building піксель-арт було обрано через його естетичну привабливість, ретро-стиль і ефективність для відображення великої кількості дрібних елементів, таких як карти, іконки та персонажі.

У проекті Aseprite використовувався для створення таких графічних елементів:

Спрайти для карт: Кожна карта мала унікальний дизайн із іконкою (наприклад, меч для атаки, щит для захисту), рамкою та текстовим полем для опису

ефекту. Aseprite забезпечував точність до пікселя, що критично для піксель-арту, де кожен піксель впливає на візуальний стиль.

Персонажі та вороги: Створювалися спрайт-листи для анімацій, таких як рухи, атаки чи смерть. Наприклад, анімація атаки персонажа включала 4 кадри: підготовка, удар, відкат і повернення в нейтральну позицію, що забезпечило плавність рухів у грі.

Оточення: Фонові елементи, такі як текстури підлоги, стін, дверей чи об'єктів (скрині, пастки), додавали атмосфери та унікальності рівням. Aseprite дозволив створювати тайлові набори (tilesets), які імпортувалися в Godot для генерації рівнів.

Aseprite спрощував створення анімацій завдяки вбудованому редактору, який розбиває спрайт-листи на кадри та налаштовує швидкість анімації. Наприклад, анімація вибуху для бойового ефекту складалася з 8 кадрів і імпортувалася в Godot як AnimatedSprite2D, що полегшило інтеграцію. Інструмент підтримує експорт у формати PNG і JSON, які Godot автоматично розпізнає, заощаджуючи час розробника.

Переваги використання Aseprite:

— Інтуїтивний інтерфейс: Інструменти малювання, палітри кольорів і шари дозволяють швидко створювати деталізовані спрайти.

— Підтримка анімацій: Вбудований редактор спрощує створення плавних переходів і ефектів, важливих для динамічних ігор.

— Експорт у зручні формати: PNG із прозорим фоном і JSON ідеально сумісні з Godot Engine.

— Гнучкість: Робота з шарами полегшує створення складних композицій і редагування.

2.3.2 Audacity для створення звукових ресурсів

Audacity — це безкоштовний аудіоредактор із відкритим кодом, який використовувався для створення звукових ефектів і музики. Він пропонує інструменти для запису, редагування, накладання ефектів і експорту аудіо, що робить його ідеальним для звукового супроводу гри.

У проекті Audacity застосовувався для:

— Звукових ефектів: Створення коротких аудіофайлів для дій, таких як переміщення карт, атака, отримання шкоди чи активація ефектів.

— Фонової музики: Розробка треків для головного меню, боїв і подій. Музика створювалася з використанням безкоштовних бібліотек і налаштовувалася для відповідності атмосфері гри: динамічний ритм для боїв, спокійна мелодія для подій.

Audacity дозволив налаштувати гучність, темп, тон і додати ефекти (exo, реверберація), створюючи унікальні звуки. Експорт у формати OGG і WAV забезпечив оптимальну якість і сумісність із Godot.

Переваги використання Audacity:

— Безкоштовність: Відкритий код робить інструмент доступним для всіх.

— Широкий функціонал: Від запису до накладання ефектів — усе для професійного звуку.

— Сумісність із Godot: Формати OGG і WAV оптимізують використання пам’яті.

2.3.3 Tiled для створення карт рівнів

Tiled — це відкритий редактор карт для 2D-ігор, який допомагав створювати шаблони рівнів і подій. Незважаючи на процедурну генерацію в грі, Tiled використовувався для розробки базових шаблонів кімнат і коридорів.

У проекті Tiled застосовувався для:

— Тайлових наборів: Дизайн підлоги, стін, дверей і об’єктів, які імпортувалися в Godot як TileMap.

— Шаблонів кімнат: Створення заготовок для бойових кімнат, скарбниць чи пасток, які комбінувалися для генерації рівнів.

Експорт у формат JSON дозволив легко інтегрувати карти в Godot, спрощуючи процедурну генерацію.

Переваги використання Tiled:

— Візуальний редактор: Швидке створення карт без кодування.

— Шари: Підтримка багатошарових карт для складних рівнів.

— Інтеграція з Godot: JSON забезпечує легке використання тайлових карт.

2.3.4 GIMP для обробки зображень

GIMP — це безкоштовний графічний редактор, який використовувався для обробки зображень, створених в Aseprite чи інших інструментах, надаючи додаткові можливості поза межами піксель-арту.

У проекті GIMP застосовувався для:

— Фонових зображень: Налаштування контрасту, яскравості та кольорів.

— UI-елементів: Дизайн кнопок, панелей і рамок.

— Оптимізації: Зменшення розміру файлів і налаштування прозорості.

Експорт у PNG забезпечив сумісність із Godot, а GIMP дозволив створювати складні композиції.

Переваги використання GIMP:

— Потужність: Фільтри, шари та маски для детального редагування.

— Безкоштовність: Доступний для всіх розробників.

— Гнучкість: Підходить для різних типів графіки.

2.3.5 Інтеграція ресурсів у Godot Engine

Ресурси інтегрувалися в Godot Engine завдяки підтримці стандартних форматів:

— Графіка: PNG імпортувалися як текстури для Sprite2D, AnimatedSprite2D або TileMap.

— Анімації: Спрайт-листи з Aseprite розпізнавалися автоматично.

— Звук: OGG-файли використовувалися через AudioStreamPlayer.

Godot спрощував імпорт і налаштування, дозволяючи зосередитися на розробці.

3 ПРОЄКТУВАННЯ

3.1 Проєктування архітектури застосунку

Архітектура застосунку базується на ігровому рушії Godot і використовує його систему вузлів та сцен для організації компонентів гри. Центральним елементом є вузол "Battle", який виконує роль кореня сцени та координує загальну логіку гри. Він відповідає за початок бою, керування ходами, а також визначення умов перемоги чи поразки. Цей вузол є основним координатором, який взаємодіє з іншими компонентами через сигнали Godot, забезпечуючи слабке зв'язування та модульний дизайн.

Підпорядковані вузлу "Battle" компоненти включають:

— Enemy Handler: Цей компонент відповідає за керування ворогами в грі. Він створює ворожі сутності, управляє їхніми ходами, штучним інтелектом та діями під час бою. Наприклад, він визначає, коли ворог атакує чи блокується, і координує їхню поведінку.

— Institute Handler: Компонент, який управляє механікою карт гравця. Він відповідає за витягування карт із колоди, скидання карт, обробку ефектів зіграних карт та керування ходом гравця. Цей вузол є ключовим для реалізації основної механіки "deck-building".

— Player: Представляє персонажа гравця в грі. Цей компонент зберігає та обробляє статистику гравця, таку як здоров'я, мана чи інші параметри, що змінюються під час бою внаслідок дій гравця чи ворогів.

— UI: Вузол користувацького інтерфейсу, який відображає стан гри. Він включає елементи для показу руки гравця (карт), статусу ворогів, здоров'я гравця, мани та іншої інформації. UI оновлюється на основі даних, отриманих від інших компонентів, і забезпечує візуальну взаємодію з гравцем.

Взаємодія між компонентами здійснюється через систему сигналів Godot. Наприклад, коли гравець зіграє карту, Institute Handler може видати сигнал, який

Battle обробляє для застосування ефектів до ворогів чи гравця, а UI отримує сигнал для оновлення відображення. Такий підхід дозволяє уникнути жорстких залежностей між компонентами, полегшуючи модифікацію та розширення гри.

Додатково, застосунок активно використовує систему ресурсів Godot для управління даними. Ресурси, такі як Картки, Колоди, Ефекти та Статистика, зберігаються у вигляді окремих файлів і динамічно завантажуються компонентами за потреби. Наприклад, Institute Handler використовує ресурси карт для формування колоди гравця, а Enemy Handler звертається до статистики ворогів для визначення їхніх характеристик. Ця відокремленість даних від логіки сприяє гнучкості системи: додавання нових карт чи ворогів не потребує зміни коду, лише створення нових ресурсів.

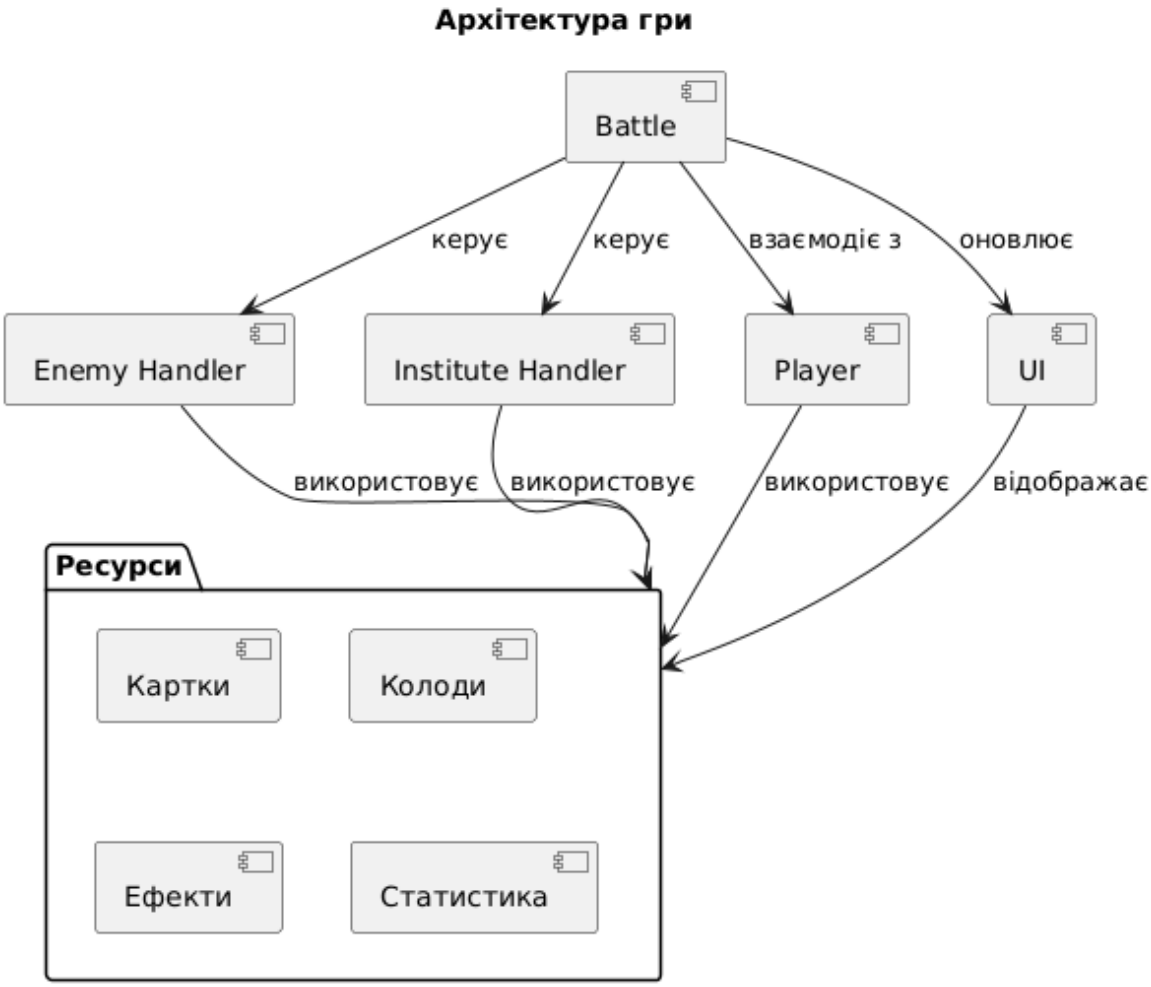


Рис. 3.1 Діаграма архітектури системи відеогри

Ця діаграма послідовності ілюструє процес гри картки гравцем у картковій грі, показуючи взаємодію між різними сутностями системи. Учасники діаграми представлені вертикальними лініями життя, а стрілки між ними позначають повідомлення (методи або події), що передаються в певній послідовності.

Учасники діаграми

— Player (актор): Людина, яка грає в гру і ініціює процес гри картки (зображена як фігурка людини).

— Player:Player: Об'єкт у системі, що представляє гравця.

— Deck:Deck: Об'єкт, що представляє колоду карт.

— Card:Card: Об'єкт, що представляє конкретну картку, яку грають.

— Battle:Battle: Об'єкт, що відповідає за поточний бій у грі.

— Events:Events: Об'єкт, що обробляє події гри та оновлює її стан.

Послідовність взаємодій

Початок процесу: Гравець (актор) надсилає повідомлення play_card() до об'єкта Player:Player, сигналізуючи про бажання зіграти картку.

Витягування картки: Об'єкт Player:Player надсилає повідомлення drawCard() до Deck:Deck, щоб витягти картку з колоди.

Повернення картки: Deck:Deck відповідає повідомленням return Card, повертаючи витягнуту картку до Player:Player.

Активація ефекту картки: Player:Player надсилає повідомлення playEffect() до Card:Card, щоб активувати ефект зіграної картки.

Застосування ефекту до бою: Card:Card надсилає повідомлення applyEffect() до Battle:Battle, застосовуючи ефект картки до поточного бою.

Оновлення подій гри:

— Battle:Battle надсилає повідомлення emit CARD_PLAYED(Card) до Events:Events, сповіщаючи систему про те, що картку зіграно.

— Battle:Battle також надсилає повідомлення update_game_state до Events:Events, щоб оновити загальний стан гри на основі ефекту картки та результатів бою.

Підтвердження дії: Player:Player надсилає повідомлення confirm_card_played назад до гравця (актора), підтверджуючи успішне виконання дії.

Завершення процесу: Гравець (актор) отримує повідомлення card_played_successfully, що знаменує завершення послідовності.

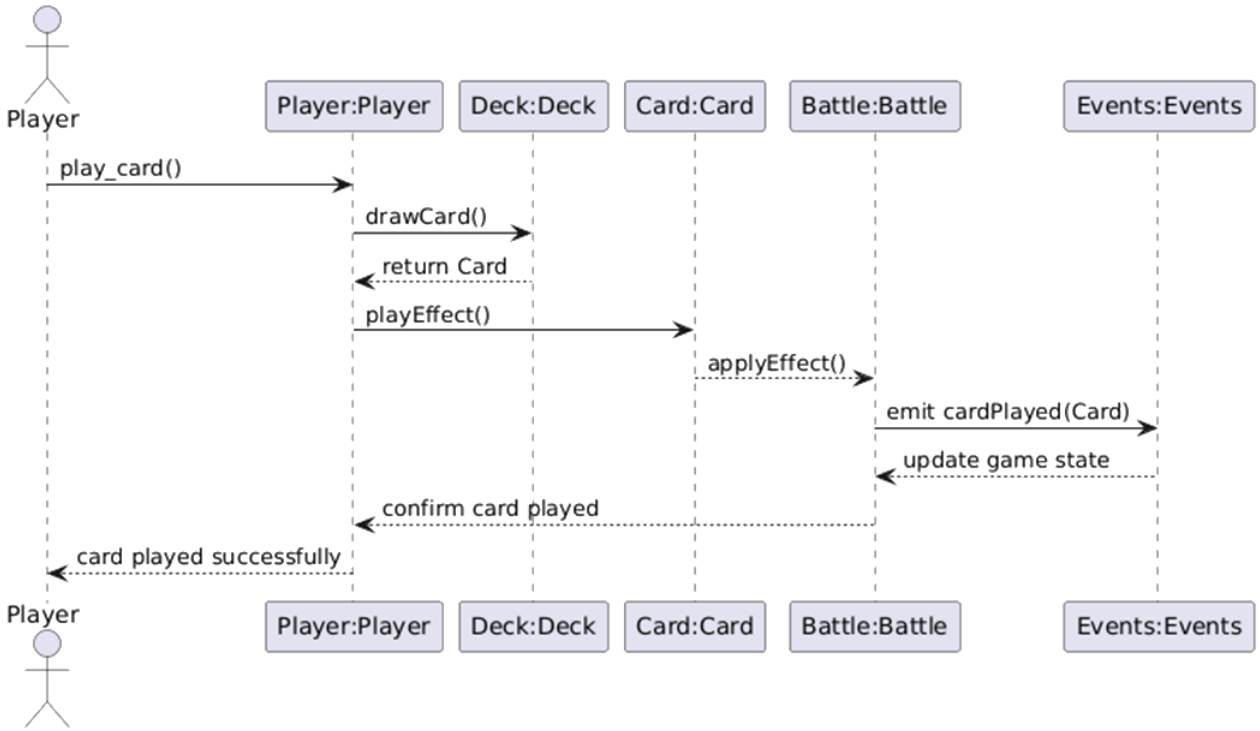


Рис. 3.2 Діаграма послідовності

Діаграма чітко демонструє, як система обробляє дію гри картки: від витягування картки з колоди до активації її ефекту в бою, сповіщення про подію та оновлення стану гри. Кожен крок логічно пов'язаний із наступним, забезпечуючи плавний перебіг ігрового процесу.

Ця діаграма класів пояснює, як влаштована структура гри. Вона описує класи, такі як Player і Card, разом із їхніми характеристиками, функціями та зв’язками між ними. Це допомагає зробити гру організованою, дозволяє повторно використовувати код і полегшує її підтримку в майбутньому.

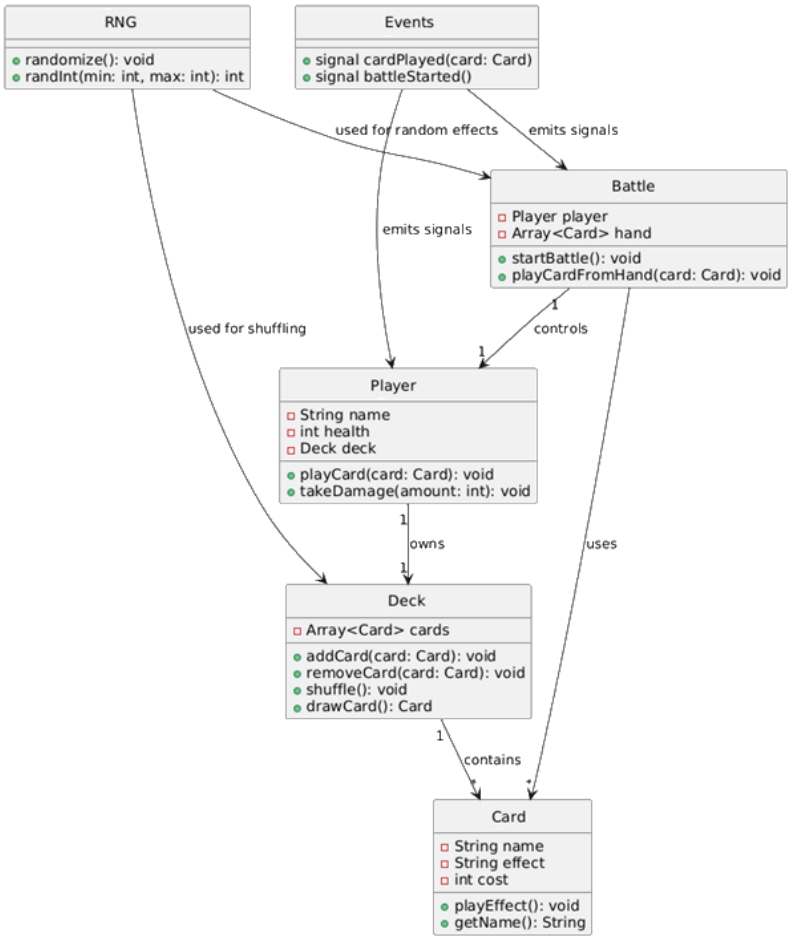


Рис. 3.3 Діаграма класів застосунку

Для зображення взаємодії користувача з грою створимо діаграму варіантів використання. На ній зліва зображується актор (в даному випадку – користувач) і система, яка відповідає на певні запроси користувача.

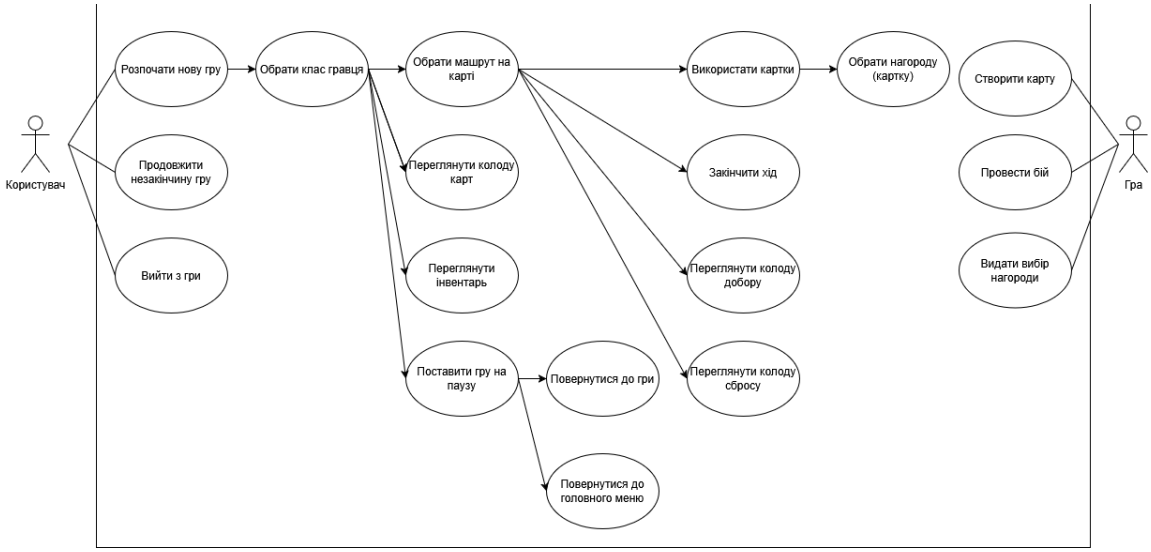


Рис. 3.4 Діаграма варіантів використання

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Дизайн інтерфесу

Інтерфейс гри, розробленої в Godot 4 як roguelike deckbuilder, поєднує мінімалістичний підхід із функціональністю, що дозволяє гравцеві легко орієнтуватися в грі, зберігаючи при цьому атмосферу підземелля, характерну для жанру. Дизайн виконаний у стилі піксель-арту з темною колірною палітрою, що підкреслює ретро-естетику та занурює гравця у світ гри.

4.1.1 Основний екран та навігація

— Головне меню: Гра починається з простого головного меню, де центральним елементом є кнопка "Почати нову гру". Цей екран, ймовірно, включає додаткові опції, такі як "Налаштування" або "Вихід", але основна увага зосереджена на початку гри. Дизайн чистий, з великим шрифтом у піксельному стилі для легкої читабельності.

— Екран вибору класу або карти: Після запуску нової гри гравець переходить до екрану вибору початкового класу чи карти. Інтерфейс представляє вибір у вигляді сітки або списку з іконками та короткими описами здібностей, що супроводжуються інтуїтивними кнопками для підтвердження вибору. Візуальна ієрархія чітка, з підсвічуванням вибраного елемента (наприклад, жовтим контуром).

4.1.2 Екран бою

— Фон і атмосфера: Екран бою занурює гравця в підземелля з темним коричневим фоном підлоги та цегляними стінами. Деталі, такі як заґратоване вікно або скелет на стіні, додають глибини атмосфері.

— Статус гравця: У верхній частині або біля персонажа відображається ім’я гравця та його здоров’я у вигляді червоної іконки серця з числовим значенням

(наприклад, "35/35"). Поруч можуть бути ресурси, такі як валюта (іконка монети з "70") або інші показники (наприклад, "10"), розміщені компактно для швидкого огляду.

— Рука гравця: У нижній частині екрану розташована горизонтальна панель із картами. Кожна карта має коричневу рамку, іконку дії (наприклад, молот для атаки) та значення вартості (наприклад, "2" у синьому ромбі). Вибрана карта підсвічується товстим жовтим контуром, а стрілка вказує на ціль, супроводжуючись банером із описом дії (наприклад, "Deal 6 damage" із червоним числом "6").

— Вороги: На правій стороні екрану зображені вороги (наприклад, кажаноподібні істоти з помаранчевими тілами) з їхніми панелями здоров'я (червоними серцями з "8") та індикаторами атаки (наприклад, "2x4"). Їхній піксельний дизайн гармонує із загальною естетикою.

— Кнопка завершення ходу: У нижньому правому куті розміщена кнопка "END TURN" у коричневій рамці з білим текстом і контуром, що виділяє її на темному фоні.

4.1.3 Екран управління колодою

— Перегляд колоди: Цей екран дозволяє гравцеві переглядати та редагувати свою колоду. Карти відображаються у вигляді списку або сітки з назвами, ефектами та іконками. Інтерфейс підтримує сортування (наприклад, за вартістю чи типом) та інтерактивні опції, такі як кнопки "Додати", "Видалити" або "Перемішати".

— Функціональність: Кнопки розташовані зручно, можливо, у верхній частині екрану, а при виборі карти з'являються додаткові деталі чи контекстне меню для її редагування.

4.1.4 Екран подій та нагород

— Вибір нагороди: Після бою чи події гравець переходить до екрану вибору нагороди, де 2–3 карти представлені поруч із назвами, ефектами та іконками.

Дизайн дозволяє легко порівнювати варіанти, з кнопками або жестами для підтвердження вибору.

— Карта подій: Інтерфейс може включати карту з вузлами (наприклад, "Бій" або "Викопати артефакт"), де гравець обирає свій шлях. Вузли позначені іконками або текстом у піксельному стилі.

4.1.5 Візуальний стиль

— Колірна палітра: Темні землисті тони (коричневий, сірий) домінують у дизайні, з червоним для здоров'я та шкоди, білим для тексту та жовтим для підсвічування активних елементів.

— Шрифти: Використовується піксельний шрифт, що відповідає ретро-естетиці. Важлива інформація (наприклад, шкода) виділена більшим розміром або кольором.

— Карти: Карти мають простий дизайн із рамкою, іконкою та текстом. Їхній вигляд нагадує фізичні карти, з чітким розмежуванням елементів для зрозумілості.

Рис. 4.1 Екранна форма грального процесу

4.2. Реалізація відеогри в жанрі roguelike deck-building

Цей розділ описує процес створення відеогри в жанрі roguelike deck-building з використанням Godot 4 на основі навчального проєкту, доступного за посиланням GitHub репозиторій, та документа "Опис виконання проекту.pdf". Проєкт покроково демонструє створення основних механік гри, таких як перетягування карт, управління колодою, бої з ворогами, система нагород і збереження прогресу. Нижче наведено послідовність етапів розробки.

4.2.1. Ініціалізація проєкту в Godot 4

Першим кроком є створення нового проєкту в Godot 4 через інтерфейс редактора. У вікні "Project Manager" обираємо "New Project", задаємо назву (наприклад, "Deck Builder Tutorial") та шлях для збереження. Після створення Godot генерує базову структуру каталогів, включаючи папки для сцен (scenes/), скриптів (scripts/) і ресурсів (res://). Для початку роботи завантажуємо стартові активи від Кенні (Kenney) з його сайту та імпортуємо їх до проєкту, перетягнувши файли у панель файлової системи.

4.2.2. Налаштування ресурсів та параметрів проєкту

На цьому етапі імпортуємо графічні активи (спрайти, шрифти) та налаштовуємо параметри проєкту:

— У "Project Settings" встановлюємо роздільну здатність гри на 256x144 пікселів (Display > Window > Size).

— Змінюємо режим розтягування на "Viewport" (Stretch > Mode), щоб гра адаптувалася до різних розмірів екрану.

— Для чіткості піксельного арту в розділі "Rendering > Textures" змінюємо "Default Texture Filter" з "Linear" на "Nearest".

4.2.3. Створення основної сцени бою

Сцена бою є центральною частиною гри:

— Створюємо нову сцену з кореневим вузлом Node2D, називаємо її "Battle" і зберігаємо як battle.tscn.

— Додаємо дочірній вузол Sprite2D для фону, призначаємо текстуру (наприклад, background.png), вимикаємо властивість "Centered" і фіксуємо позицію через іконку замка в редакторі.

— Налаштовуємо затемнення фону через властивість "Modulate" для кращої видимості елементів.

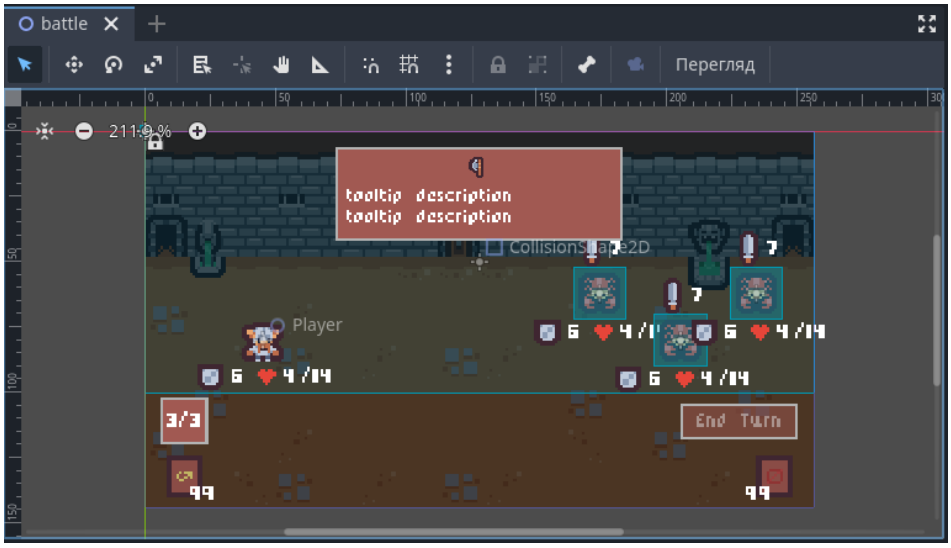


Рис 4.2 Сцена battle.tscn

4.2.4. Реалізація механіки перетягування карт

Механіка перетягування карт — ключовий елемент гри:

— Створюємо сцену CardUI.tscn з кореневим вузлом Control.

— Додаємо дочірні вузли ColorRect (для фону карти) і Label (для відображення стану).

— Налаштовуємо розмір карти (наприклад, 25x30 пікселів) і додаємо тему зі шрифтом (наприклад, "Pixel RPG Font").

— Додаємо Area2D (DropPointDetector) для виявлення зони скидання з колізійним шаром "CardTargetSelector".

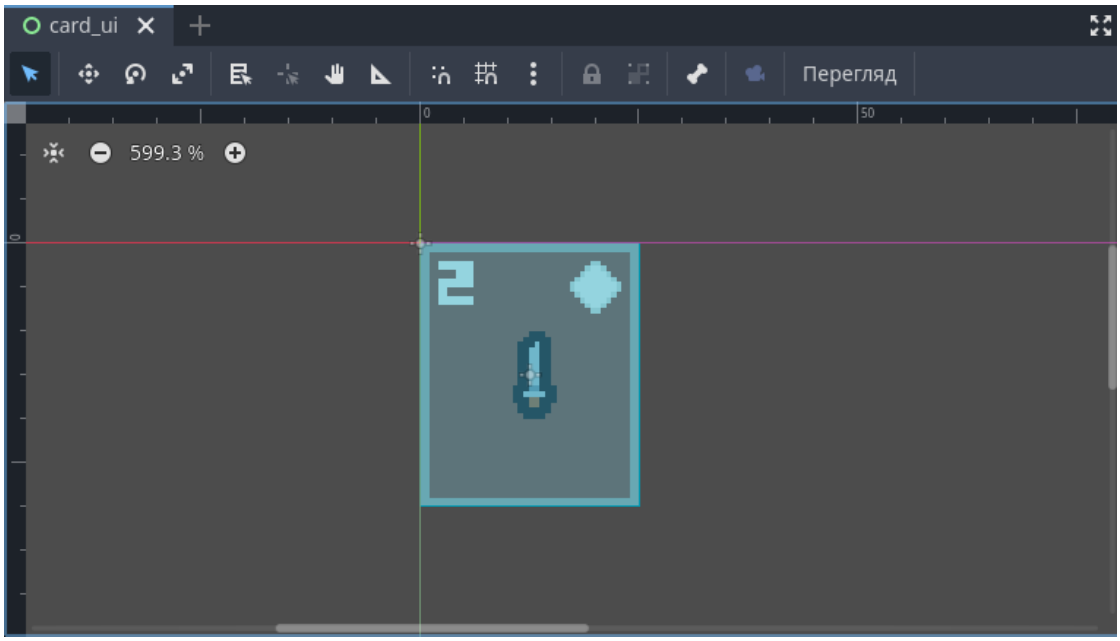


Рис. 4.3 Сцена CardUI.tscn

4.2.5. Налаштування зони скидання карт

Зона скидання дозволяє зіграти карти:

— У сцені battle.tscn додаємо вузол Area2D під назвою "CardDropArea".

— Налаштовуємо колізійний шар "CardDropArea" і маску для взаємодії з "CardTargetSelector".

— Додаємо CollisionShape2D з формою RectangleShape2D розміром 256x100 пікселів, розташованим у верхній частині екрану.

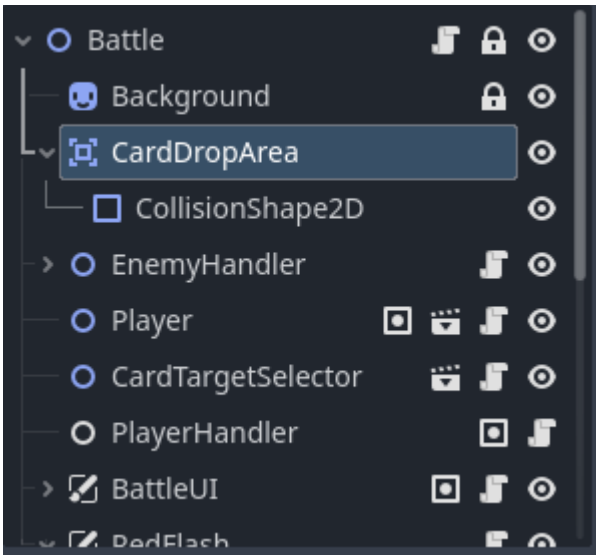


Рис 4.4 В сцені battle.tscn додаємо вузол CardDropArea

4.2.6. Додавання інтерфейсу користувача

Інтерфейс відображає руку гравця:

— Додаємо CanvasLayer до сцени бою під назвою "BattleUI".

— Створюємо HBoxContainer ("Hand") для карт у руці, додаємо три екземпляри CardUI.tscn як дочірні вузли.

— Налаштовуємо якір контейнера на "Center Bottom" і задаємо мінімальний розмір.

Рис. 4.5 В сцені battle.tscn додаємо CanvasLayer під назвою "BattleUI"

4.2.7. Реалізація кінцевої машини станів для карт

Для управління станами карт створюємо кінцеву машину станів:

— Створюємо базовий скрипт CardState.gd з переліком станів (Base, Clicked, Dragging, Released).

— Реалізуємо скрипти для кожного стану (наприклад, CardBaseState.gd, CardDraggingState.gd) з методами enter(), exit(), _on_input().

— У сцені CardUI.tscn додаємо вузол CardStateMachine зі скриптом CardStateMachine.gd для переходів між станами.

Реалізація методу в card_state.gd:

```
class_name CardState
extends Node
enum State {BASE, CLICKED, DRAGGING, AIMING, RELEASED}
signal transition_requested(from: CardState, to: State)
@export var state: State
var card_ui: CardUI

func enter() -> void:
	pass

func exit() -> void:
	pass

func post_enter() -> void:
	pass

func on_input(_event: InputEvent) -> void:
	pass

func on_gui_input(_event: InputEvent) -> void:
	pass

func on_mouse_entered() -> void:
	pass

func on_mouse_exited() -> void:
	pass
```

4.2.8. Налаштування обробки подій вводу

Обробка подій миші:

— У скрипті CardUI.gd реалізуємо методи _input() і _on_gui_input() для перехоплення кліків і перетягувань.

— Передаємо події до поточного стану через машину станів.

Реалізований метод _input() в card_ui.gd:

```
func _input(event: InputEvent) -> void:
        card_state_machine.on_input(event)
```

Реалізований метод _on_gui_input() в card_ui.gd:

```
func _on_gui_input(event: InputEvent) -> void:
        card_state_machine.on_gui_input(event)
```

4.2.9. Додавання ворогів та їхньої поведінки

Вороги додають бойову динаміку:

— Створюємо сцену ворога (enemy.tscn) з вузлами Sprite2D і Area2D.

— Реалізуємо скрипт для ворога з логікою атак і блокування.

— Додаємо екземпляр ворога до сцени battle.tscn.

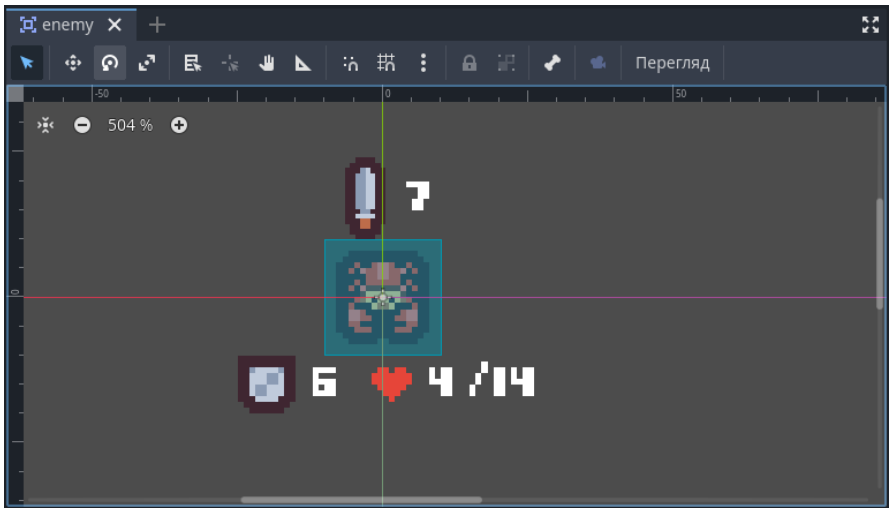


Рис. 4.6 Сцена enemy.tscn

4.2.10. Реалізація системи колод та управління картами

Система колод:

— Створюємо ресурс CardPile.gd для представлення колоди.

— Додаємо методи для перемішування, витягування та додавання карт.

— Інтегруємо колоду в сцену бою для витягування карт у руку.

Методи для перемішування, витягування та додавання карт в cardpile.gd:

```
func draw_card() -> Card:
	var card = cards.pop_front()
	card_pile_size_changed.emit(cards.size())
	return card

func add_card(card: Card) -> void:
	cards.append(card)
	card_pile_size_changed.emit(cards.size())

func shuffle() -> void:
	RNG.array_shuffle(cards)

func clear() -> void:
	cards.clear()
	card_pile_size_changed.emit(cards.size())
func duplicate_cards() -> Array[Card]:
	var new_array: Array[Card] = []
	for card: Card in cards:
		new_array.append(card.duplicate())
	return new_array

func custom_duplicate() -> CardPile:
	var new_card_pile := CardPile.new()
```

```
        new_card_pile.cards = duplicate_cards()

        return new_card_pile

func _to_string() -> String:
        var _card_strings: PackedStringArray = []
        for i in range(cards.size()):
                _card_strings.append("%s: %s" % [i+1, cards[i].id])
        return "\n".join(_card_strings)
```

4.2.11. Додавання системи нагород та прогресії

Нагороди після бою:

— Створюємо сцену battle_reward.tscn з інтерфейсом вибору карт або реліквій.

— Реалізуємо логіку випадкового вибору нагород через RNG.array_pick_random().

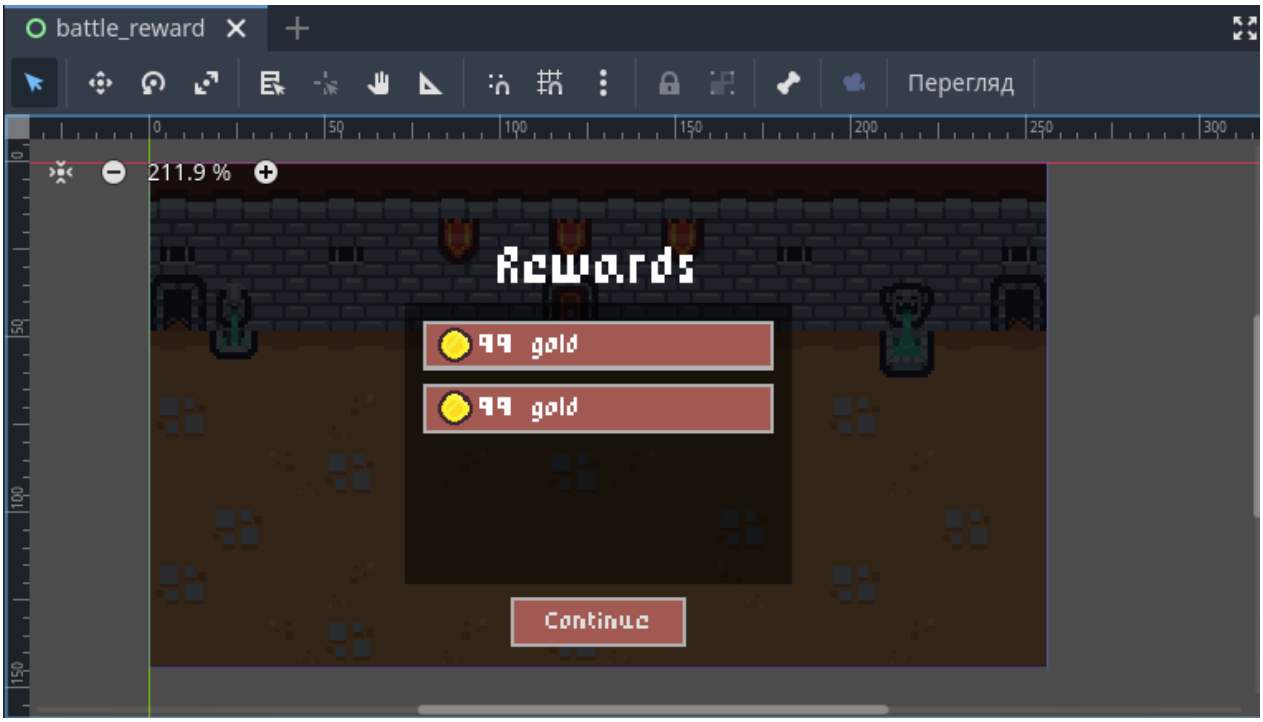


Рис. 4.7 Створена сцена battle_reward.tscn

4.2.12. Налаштування системи збереження та завантаження

Збереження прогресу:

— Створюємо ресурс SaveGame.gd з експортованими змінними (колода, здоров'я, реліквії тощо).

— Реалізуємо методи save_data() і load_data() з використанням ResourceSaver і ResourceLoader.

— Додаємо автозавантажуваний скрипт rng.gd для фіксованої генерації чисел на основі насіння.

Реалізовані методи save_data(), load_data() і deleta_data в SaveGame.gd:

```
func save_data() -> void:
	var err := ResourceSaver.save(self, SAVE_PATH)
	assert(err == OK, "Couldn't save the game!")

static func load_data() -> SaveGame:
	if FileAccess.file_exists(SAVE_PATH):
		return ResourceLoader.load(SAVE_PATH) as SaveGame

	return null

static func delete_data() -> void:
	if FileAccess.file_exists(SAVE_PATH):
		DirAccess.remove_absolute(SAVE_PATH)
```

4.2.13. Додавання головного меню та екрану перемоги

Завершення ігрового циклу:

— Створюємо сцени main_menu.tscn і win_screen.tscn з вузлами Control.

— Реалізуємо кнопки для початку гри, продовження та виходу.

— Налаштовуємо перехід до win_screen.tscn після перемоги над босом.

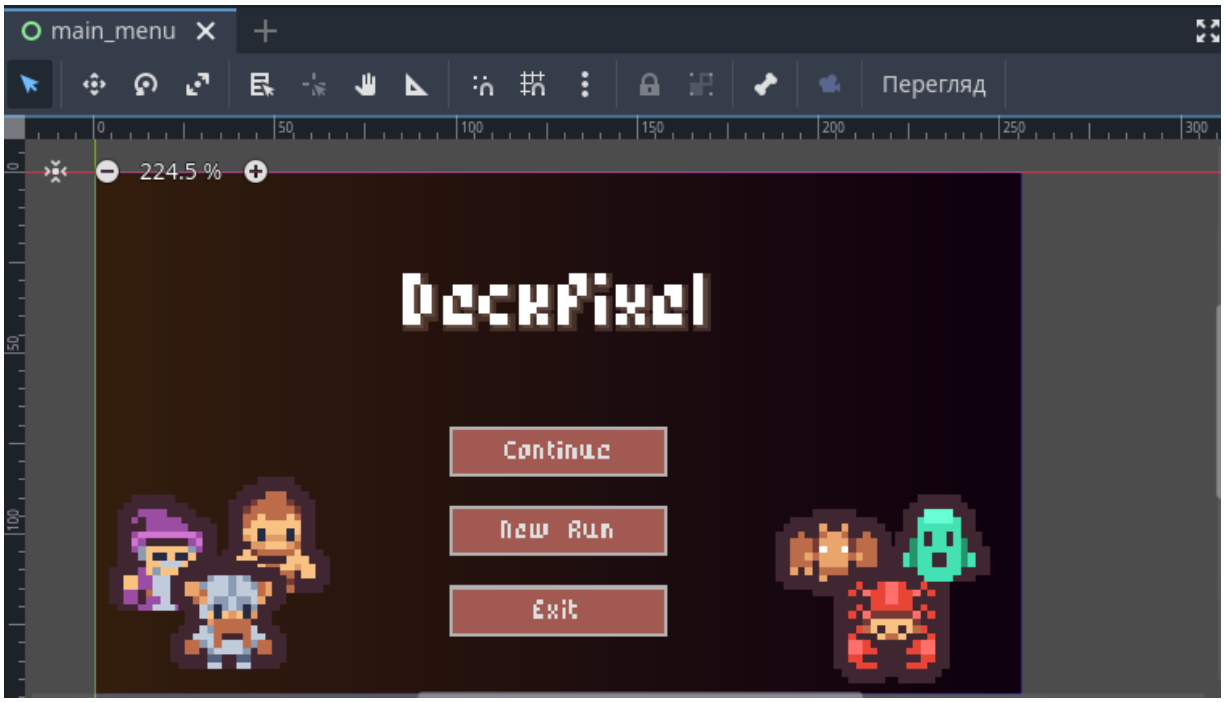


Рис. 4.8 Сцена main_menu.tscn

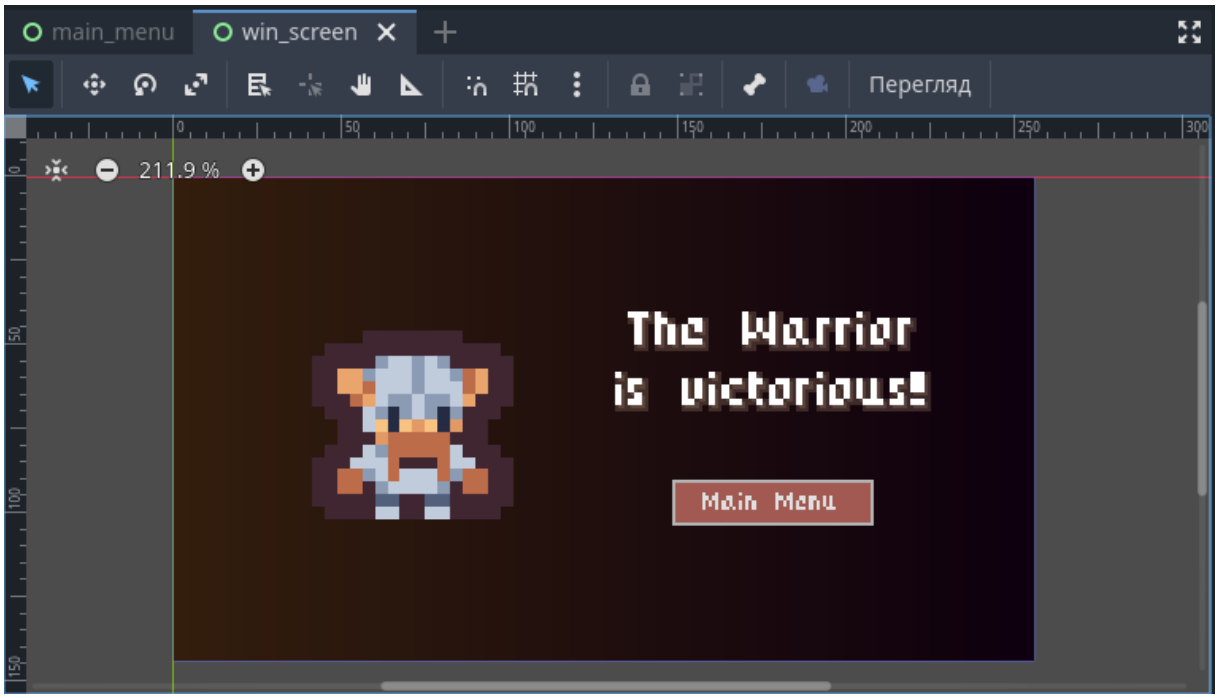


Рис. 4.9 Сцена win_screen.tscn

4.2.14. Тестування та відлагодження

Перевірка працездатності:

— Тестуємо механіки (перетягування карт, бої, збереження) через вбудовану консоль Godot.

— Виправляємо баги, наприклад, очищення цілей при скасуванні прицілювання або коректне повернення карт у руку.

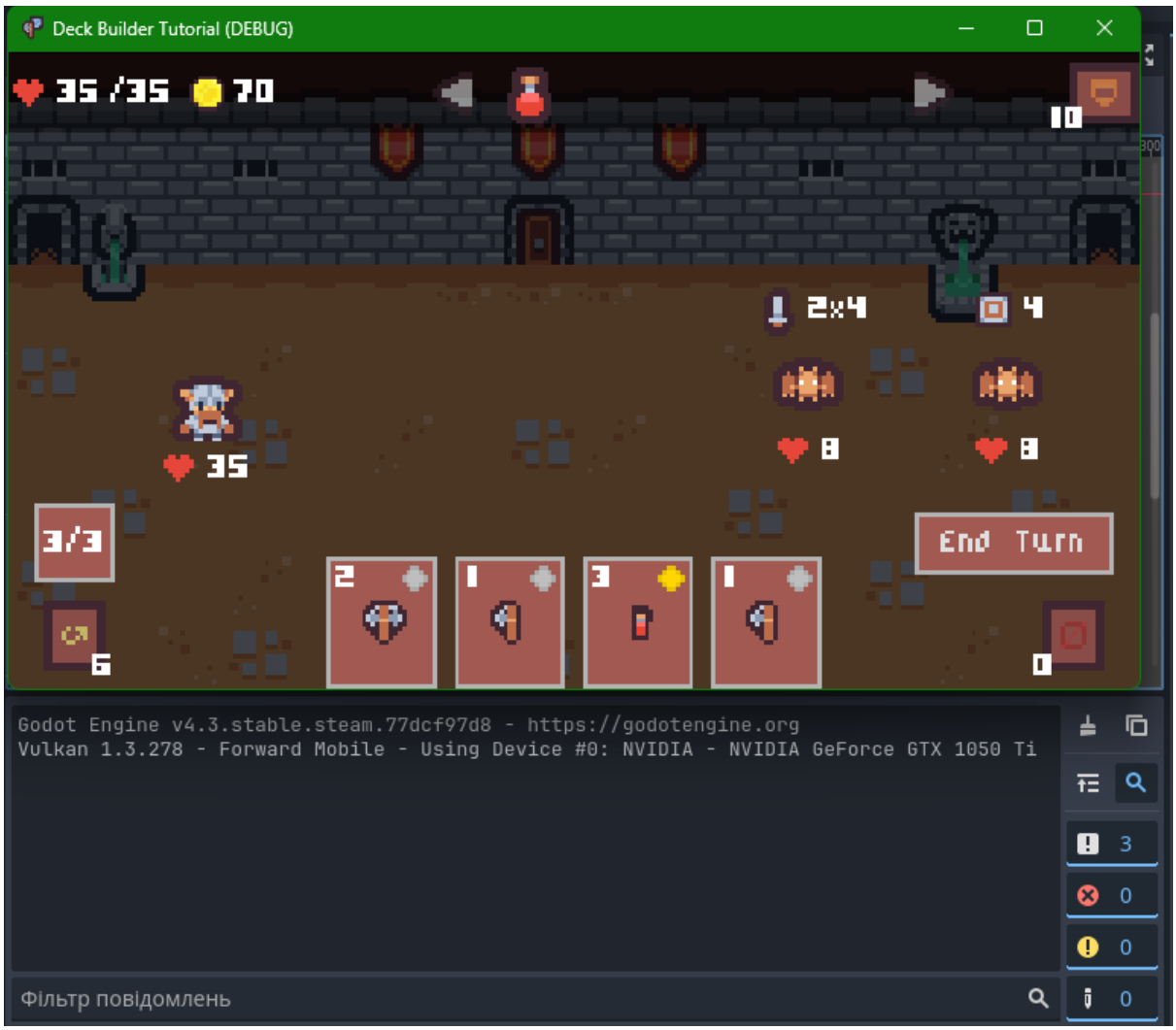


Рис. 4.10 Тестування застосунку

4.3 Тестування застосунку

Тестування є невід'ємною частиною розробки відеогри в жанрі roguelike deck-building на рушії Godot Engine, що дозволяє забезпечити її стабільність, правильність роботи та високу якість користувацького досвіду. Тестування охоплює як окремі компоненти гри, так і їх взаємодію, а також загальну поведінку системи.

4.3.1 Інструменти тестування

Для тестування застосунку використано вбудовані можливості Godot Engine, зокрема фреймворк **GUT (Godot Unit Test)**, який дозволяє писати та запускати тести безпосередньо в рушії. GUT підтримує модульне та інтеграційне тестування, що робить його зручним для перевірки як окремих скриптів GDScript, так і їхньої взаємодії. Крім того, для оцінки поведінки гри використовувалися ручні тести та дебаг-інструменти Godot, такі як консоль виведення та інспектор.

4.3.2 Модульне тестування

Модульне тестування зосереджено на перевірці окремих компонентів гри в ізоляції. Це дозволяє швидко виявляти помилки в конкретних функціях чи класах. У цьому проєкті модульні тести застосовувалися до ключових елементів, описаних у документі "Опис виконання проекту.pdf", зокрема:

— Створення карт: Перевірка правильності ініціалізації об’єктів карт із заданими властивостями, такими як назва, вартість і ефект.

— Керування колодою: Перевірка функцій додавання карт, перемішування колоди та витягування карт.

— Штучний інтелект ворогів: Тестування логіки вибору дій ворогів, наприклад, чи Токсичний Привид правильно підсилює себе на першому ході.

— Статистика гравця: Перевірка коректності оновлення здоров’я, мани та інших параметрів після дій.

4.3.3 Інтеграційне тестування

Інтеграційне тестування перевіряє взаємодію між різними компонентами гри. У цьому проєкті особлива увага приділялася таким аспектам:

— Перетягування карт: Перевірка, чи карта коректно переходить із руки в область скидання, оновлюючи стан колоди та руки.

— Бойова система: Тестування взаємодії між гравцем і ворогами, наприклад, чи атака Токсичного Привида завдає правильну кількість шкоди та додає карту "Токсин" до колоди гравця.

— Інтерфейс користувача: Перевірка, чи UI (наприклад, рука гравця чи статистика) оновлюється відповідно до дій у грі.

4.3.4 Системне тестування

Системне тестування оцінює гру як цілісну систему, перевіряючи повний ігровий цикл. У цьому проєкті системні тести охоплювали:

— Прогрес гри: Перевірка переходів між картою, боями, крамницею та скарбницею.

— Збереження та завантаження: Тестування коректності збереження стану гри та його завантаження через кнопку "Продовжити".

— Умова перемоги: Перевірка, чи після перемоги над босом (Токсичний Привид) відображається екран перемоги та чи працює перехід до головного меню.

4.3.5 Ручне тестування та виправлення помилок

Окрім автоматизованих тестів, які забезпечують базову перевірку функціональності гри, проводилося ручне тестування. Воно необхідне для виявлення проблем, які важко або неможливо автоматизувати, таких як баги інтерфейсу, некоректна поведінка при швидких діях гравця, візуальні дефекти чи інші аспекти, що потребують людської оцінки. Ручне тестування відіграє ключову роль у доповненні автоматизованих методів, забезпечуючи всебічну перевірку якості та стабільності гри.

Ручне тестування проводилося з чітко визначеними цілями, щоб охопити аспекти, які не піддаються автоматичній перевірці:

— Виявлення UI/UX проблем: Перевірка зручності та коректності відображення елементів інтерфейсу, таких як карти, кнопки, текстові описи, а також їх відповідність дизайну гри.

— Тестування на крайніх випадках: Аналіз поведінки гри в нестандартних ситуаціях, наприклад, при переповненні колоди карт, нульовому здоров’ї персонажа чи максимальному завантаженні системи.

— Перевірка візуальних ефектів: Оцінка анімацій, спрайтів, графічних переходів та інших візуальних компонентів для виявлення дефектів, таких як зсув текстур чи затримки.

— Тестування на різних пристроях: Перевірка сумісності гри з різними роздільними здатностями екранів, операційними системами та апаратними конфігураціями.

Для досягнення поставлених цілей застосовувалися різні методи ручного тестування, які дозволяли охопити широкий спектр потенційних проблем:

— Експлораторне тестування: Вільне дослідження гри без заздалегідь визначених сценаріїв для виявлення неочевидних помилок, що виникають у процесі взаємодії з системою.

— Сценарії користувача: Імітація типових дій гравця, таких як створення колоди, проходження рівнів, використання карт чи взаємодія з інтерфейсом, для перевірки логіки гри.

— Стрес-тестування: Виконання швидких і повторюваних дій (наприклад, багаторазове натискання кнопок або перетягування елементів) для оцінки стабільності та реакції гри на високе навантаження.

У процесі ручного тестування було виявлено низку проблем, які вплинули на якість гри. Нижче наведено приклади таких проблем та способи їх усунення:

— Проблема: Карта "зависала" на екрані при швидкому перетягуванні з однієї зони в іншу.

Виправлення: Додано механізм очищення масиву цілей при скасуванні дії гравця, що усунуло конфлікт у логіці переміщення.

— Проблема: Текст на картах з довгими описами обрізався або відображався некоректно.

Виправлення: У UI-елементах налаштовано автоматичне перенесення тексту та додано адаптивне масштабування шрифту залежно від розміру картки.

ВИСНОВКИ

Результатом виконаної кваліфікаційної роботи стала розробка повноцінної відеогри в жанрі roguelike deck-building на рушії Godot Engine. Ця гра поєднує ключові механіки жанру, такі як процедурна генерація вмісту, стратегічне управління колодою карт, тактична бойова система, система прогресії та інтуїтивно зрозумілий користувацький інтерфейс.

У процесі виконання роботи було успішно вирішено низку поставлених завдань, кожне з яких відіграло важливу роль у досягненні кінцевої мети:

1. Аналіз предметної області. Проведено ґрунтовне дослідження жанру roguelike deck-building, що включало аналіз популярних ігор, таких як Slay the Spire, Monster Train, Griftlands і Inscryption. Ці ігри стали основою для визначення ключових механік, таких як випадкова генерація подій і рівнів, управління колодою карт, балансування складності та створення захопливого ігрового досвіду. На основі аналізу сформульовано технічні та функціональні вимоги до проекту, зокрема необхідність реалізації процедурної генерації вмісту, системи карт, бойової системи з тактичними рішеннями, системи нагород і прогресії, а також зручного інтерфейсу для взаємодії гравця з грою. Цей етап дозволив чітко окреслити цілі розробки та уникнути надмірної складності в реалізації.

2. Обрання засобів реалізації. Для розробки гри обрано рушій Godot Engine, який вирізняється своєю гнучкістю, підтримкою 2D-ігор і відкритим вихідним кодом. Godot Engine забезпечив ефективне створення прототипів і впровадження складних ігрових систем завдяки вбудованим інструментам, таким як система вузлів, сигнали та ресурси. Для програмування використано мову GDScript, яка є інтуїтивною та оптимізованою для роботи з Godot. Додатково використано систему ресурсів Godot для управління даними карт, ворогів і подій, що спростило організацію даних гри. Для створення інтерфейсу застосовано вбудовані інструменти Godot, такі як Control nodes, що дозволили реалізувати адаптивний і

зручний дизайн. Тестування проводилося з використанням фреймворку GUT (Godot Unit Test), який автоматизував перевірку модулів і підвищив якість коду.

3. Проєктування архітектури. Розроблено модульну архітектуру гри, що складається з кількох ключових компонентів, які забезпечують чітке розділення функціоналу та легке масштабування системи. Основними вузлами є:

— Battle: відповідає за координацію бойових сцен, обробку черговості ходів і взаємодію між гравцем і ворогами.

— Enemy Handler: управляє поведінкою ворогів, їхньою логікою та характеристиками, що дозволяє створювати різноманітних противників.

— Institute Handler: обробляє логіку карт гравця, включаючи їхнє використання, комбінації та ефекти.

— UI: забезпечує відображення ігрового інтерфейсу, включаючи колоду карт, характеристики гравця, інформацію про ворогів і систему нагород. Використано систему сигналів Godot для реалізації слабкого зв'язування між компонентами, що зменшило залежності та полегшило тестування й подальшу модифікацію коду. Такий підхід до архітектури сприяв створенню гнучкої та надійної системи, готової до розширення.

4. Реалізація системи. На етапі розробки впроваджено основні ігрові механіки, що відповідають жанру roguelike deck-building. Зокрема:

— Перетягування карт (drag-and-drop): реалізовано інтерактивну систему, що дозволяє гравцям легко використовувати карти через інтуїтивний інтерфейс.

— Управління колодою: створено систему, яка підтримує додавання, видалення та модифікацію карт у колоді, а також комбінування їх для створення унікальних стратегій.

— Бойова система: розроблено тактичну систему, де гравець приймає рішення на основі доступних карт, характеристик ворогів і власних ресурсів, таких як енергія чи здоров'я.

— Система нагород: після кожного бою гравець отримує нові карти, ресурси або покращення, що сприяє прогресії та мотивує до подальшої гри.

— Система збереження та завантаження: використано ресурси Godot для збереження прогресу гравця, включаючи колоду, статистику та стан гри. Для процедурної генерації вмісту реалізовано генератор випадкових чисел на основі насіння, що забезпечує повторюваність ігрових сесій і дозволяє гравцям повертатися до попередніх проходжень із тими самими умовами. Особливу увагу приділено оптимізації продуктивності, зокрема зменшенню кількості обчислень у реальному часі та ефективному управлінню пам'яттю, що забезпечило плавну роботу гри навіть на пристроях із обмеженими ресурсами.

5. Тестування застосунку. Для забезпечення якості гри проведено комплексне тестування на кількох рівнях:

— Модульне тестування: перевірено коректність роботи окремих компонентів, таких як логіка карт, поведінка ворогів і генерація подій, за допомогою фреймворку GUT. Автоматизовані тести дозволили швидко виявляти помилки в коді та підвищили надійність системи.

— Інтеграційне тестування: протестовано взаємодію між компонентами, наприклад, правильність передачі даних між вузлами Battle, Enemy Handler і UI.

— Системне тестування: проведено перевірку гри в цілому, включаючи сценарії взаємодії гравця з інтерфейсом, стабільність бойової системи та коректність збереження прогресу.

— Ручне тестування: виконано для оцінки користувацького досвіду, зокрема зручності інтерфейсу, балансу гри та загального враження від ігрового процесу. Виявлені помилки, такі як некоректне відображення карт або нестабільність у рідкісних сценаріях, були оперативно виправлені. Цей підхід до тестування гарантував стабільність гри, її відповідність вимогам і приємний ігровий досвід для користувачів.

Розроблена гра є не лише функціональним продуктом, але й цінним навчальним ресурсом. Вона демонструє практичне застосування Godot Engine для створення складних ігрових систем, від базового налаштування сцени до реалізації процедурної генерації та модульної архітектури. Проект супроводжується докладною документацією та коментованим кодом, що робить його доступним для

вивчення розробниками-початківцями. Структура гри дозволяє легко розширювати її функціонал, наприклад, додаванням нових персонажів, типів карт, ворогів, рівнів або навіть мережевих функцій, таких як таблиця лідерів.

Перспективи подальшого розвитку. У майбутньому проект може бути вдосконалений шляхом:

— Додавання нових механік, таких як кооперативний режим або спеціальні події в грі.

— Розширення контенту, включаючи нові колоди карт, унікальних ворогів і сюжетні елементи.

— Оптимізації для різних платформ, зокрема мобільних пристроїв і веб-браузерів, завдяки кросплатформним можливостям Godot Engine.

— Впровадження аналітики ігрового процесу для збору даних про поведінку гравців і подальшого балансування гри.

— Інтеграції з онлайн-сервісами, такими як хмарне збереження прогресу чи соціальні функції.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Ясінський А.Б., Ярошевський О.В. Розробка відеогри в жанрі roguelike deck-building на рушії Godot Engine. Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025, С. 245-246.

2. Ясінський А.Б., Ярошевський О.В. Аналіз інструментів рушія godot для розробки ROGUELIKE DECK-BUILDING гри. Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025, С. 247-248.

ПЕРЕЛІК ПОСИЛАНЬ

1. Adams E. Fundamentals of Game Design. 3rd ed. New Riders, 2014. 560 с. Книга описує основи дизайну ігор, включаючи принципи створення механік для жанрів, таких як roguelike, та стратегії балансування ігрового процесу.

2. Hendrikx M., Meijer S., Van Der Velden J., Iosup A. Procedural Content Generation for Games: A Survey. ACM Transactions on Multimedia Computing, Communications, and Applications. 2013. Vol. 9, no. 1. P. 1–22. DOI: 10.1145/2422956.2422957. Наукова стаття аналізує методи процедурної генерації контенту в іграх, включаючи алгоритми для створення рівнів і подій, ключові для жанру roguelike.

3. Godot Engine Documentation. Godot Engine Official Documentation. 2025. URL: https://docs.godotengine.org/en/stable/ (дата звернення: 09.06.2025). Офіційна документація Godot Engine із детальними інструкціями щодо використання рушія для розробки 2D-ігор, включаючи GDScript і процедурну генерацію.

4. Juul J. The Art of Failure: An Essay on the Pain of Playing Video Games. MIT Press, 2013. 168 с. Наукове дослідження аналізує ігровий досвід, зокрема аспекти складності та реіграбельності, важливі для жанру roguelike deck-building.

5. Karavolos D., Liapis A., Yannakakis G. N. Using a Surrogate Model of Gameplay for Automated Level Design in a Roguelike Game. Procedia Computer Science. 2018. Vol. 136. P. 405–414. DOI: 10.1016/j.procs.2018.08.272. Стаття досліджує автоматизацію дизайну рівнів у roguelike-іграх за допомогою процедурної генерації з акцентом на балансування складності.

6. Togelius J., Yannakakis G. N., Stanley K. O., Browne C. Search-Based Procedural Content Generation: A Taxonomy and Survey. IEEE Transactions on Computational Intelligence and AI in Games. 2011. Vol. 3, no. 3. P. 172–186. DOI: 10.1109/TCIAIG.2011.2161576. Наукова стаття систематизує методи процедурної генерації в іграх, включаючи алгоритми, які можна застосувати в Godot Engine.

7. GDNative Documentation. Godot Engine GDNative. 2025. URL: https://docs.godotengine.org/en/stable/tutorials/scripting/gdnative/index.html (дата звернення: 09.06.2025). Документація описує використання GDNative для розширення можливостей Godot Engine, зокрема для процедурної генерації та оптимізації.

8. Schell J. The Art of Game Design: A Book of Lenses. 3rd ed. CRC Press, 2020. 610 с. Авторитетний посібник із дизайну ігор, що охоплює принципи створення ігрових механік, наративу та досвіду гравця, важливих для жанру roguelike deck-building.

9. Koster R. A Theory of Fun for Game Design. 2nd ed. O'Reilly Media, 2013. 300 с. Книга досліджує психологічні аспекти цікавості ігор, допомагаючи зрозуміти, як створити механіки для залучення гравців у жанрі roguelike deck-building.

10. Smith G., Whitehead J., Treanor M. PCG-based Game Design: Creating Endless Web. Proceedings of the Fifth International Conference on the Foundations of Digital Games. 2010. P. 188–195. Стаття присвячена використанню процедурної генерації контенту в дизайні ігор із прикладами технік для різноманітних рівнів у roguelike-іграх.

11. Dormans J. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. Proceedings of the 2010 Workshop on Procedural Content Generation in Games. 2010. P. 1–8. Дослідження зосереджене на генерації рівнів і місій, методи якого можна адаптувати для процедурно згенерованих рівнів у roguelike-іграх.

12. Godot Engine Community Tutorials. Godot Engine Tutorials. 2025. URL: https://godotengine.org/tutorials (дата звернення: 09.06.2025). Колекція навчальних посібників від спільноти Godot, що охоплюють механіки 2D-ігор і процедурну генерацію.

13. Brackeen D. Developing Games with Godot Engine. Manning Publications, 2023. 350 с. Книга пропонує практичний підхід до вивчення Godot Engine із прикладами, що можуть бути основою для розробки гри в жанрі roguelike deck-building.

14. Liapis A., Yannakakis G. N., Togelius J. Enhancements to Constrained Novelty Search: Two-Population Novelty Search for Generating Game Content. Proceedings of the 2013 Genetic and Evolutionary Computation Conference. 2013. P. 1443–1450. Стаття досліджує вдосконалені техніки процедурної генерації контенту з використанням еволюційних алгоритмів для унікальних ігрових елементів.

15. Godot Engine Forum. Godot Engine Community Forum. 2025. URL: https://forum.godotengine.org (дата звернення: 09.06.2025). Онлайн-форум для розробників Godot Engine, де можна знайти рішення для типових проблем розробки.

16. Togelius J., Preuss M., Beume N. Controllable Procedural Map Generation via Multiobjective Evolution. Genetic Programming and Evolvable Machines. 2010. Vol. 11, no. 2. P. 245–277. Дослідження представляє методи генерації ігрових карт за допомогою еволюційних алгоритмів, адаптованих для roguelike-ігор.

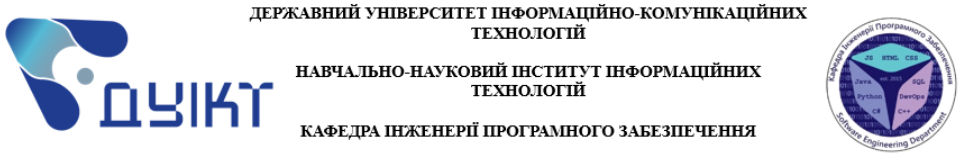
17. Godot Engine GitHub Repository. Godot Engine Source Code. 2025. URL: https://github.com/godotengine/godot (дата звернення: 09.06.2025). Офіційний репозиторій Godot Engine на GitHub із доступом до вихідного коду та трекера проблем для глибшого розуміння рушія.

18. Shaker N., Togelius J., Nelson M. J. Procedural Content Generation in Games. Springer, 2016. 237 с. Всебічна книга про теорію, техніки та приклади процедурної генерації контенту в іграх, важлива для складних систем.

19. Godot Engine Asset Library. Godot Asset Library. 2025. URL: https://godotengine.org/asset-library (дата звернення: 09.06.2025). Репозиторій безкоштовних активів, скриптів і плагінів для Godot Engine, що прискорює розробку.

20. Yannakakis G. N., Togelius J. Artificial Intelligence and Games. Springer, 2018. 337 с. Книга досліджує штучний інтелект у розробці ігор, включаючи процедурну генерацію та балансування, актуальні для жанру roguelike.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



Розробка відеогри в жанрі roguelike deck-building на рушії godot engine.

Виконав студент 4 курсу
Групи ПД-43
Ясінський Андрій Борисович
Керівник роботи
асистент кафедри ІПЗ Ярошевський Олександр Вікторович
Київ – 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи**: спрощення та оптимізація процесу розробки ігри жанру roguelike deck-building за рахунок застосування модульної архітектури та патернів у Godot Engine.
- **Об'єкт дослідження**: процес розробки відеоігор у жанрі roguelike та deck-building з використанням Godot Engine.
- **Предмет дослідження**: засоби й методи реалізації процедурної генерації рівнів, управління колодою карт і бойової логіки в Godot Engine.

2

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Проаналізувати та оглянути архітектуру Godot Engine та вивчення його можливостей для реалізації проекта.
2. Провести аналіз існуючих механік жанру.
3. Проєктування й документування загальної архітектури гри.
4. Реалізація ключових модулів: генератор рівнів, система колоди, логіку бою та взаємодії карт.
5. Розробка користувацького інтерфейсу.
6. Проведення тестування та перевірки застосунку на відповідність до вимог.

3

АНАЛІЗ АНАЛОГІВ

Функція	Slay the Spire	Monster Train	Griftlands	Dream Quest	DeckPixel
Процедурна генерація	+	+	+	+	+
Механіка побудови колоди	+	+	+	+	+
Перманентна смерть	+	-	–	+	+
Різноманітність карт	-	+	+	+	+
Різноманітність ворогів	-	+	+	+	+
Графіка/Стиль	+	-	+	–	+
Звук/Музика	+	+	+	–	+
Інтерфейс (UI/UX)	+	-	+	–	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Генератор рівнів: випадкова побудова лабіринту з контрольованою складністю.
2. Система колоди: створення, збереження, тасування та оновлення колод карт.
3. Бойовий-менеджер: поетапні ходи гравця та ворогів із застосуванням карт.
4. Користувацький інтерфейс: інтуїтивні меню, відображення стану здоров'я/енергії, візуалізація карт.
5. Збереження прогресу: локальні зберігання з можливістю відновлення.

Нефункціональні вимоги:

1. Продуктивність: мінімальна частота кадрів — 60 FPS на середньому ПК.
2. Модульність: чітке розділення компонентів із можливістю розширення новими механіками.
3. Використовність: простий і зрозумілий інтерфейс для першого знайомства користувача.
4. Надійність: коректна обробка помилок, стійкість до некоректних вхідних даних.
5. Мінімальні системні вимоги: ОС: Windows XP, Vista, 7, 8/8.1, 10;

Процесор: 2.0 Ghz;

Оперативна пам'ять: 2 GB ОП;

Відеокарта: 1 Gb Video Memory, capable of OpenGL 3.0+ support;

Місце на диску: 300 MB доступного місця.

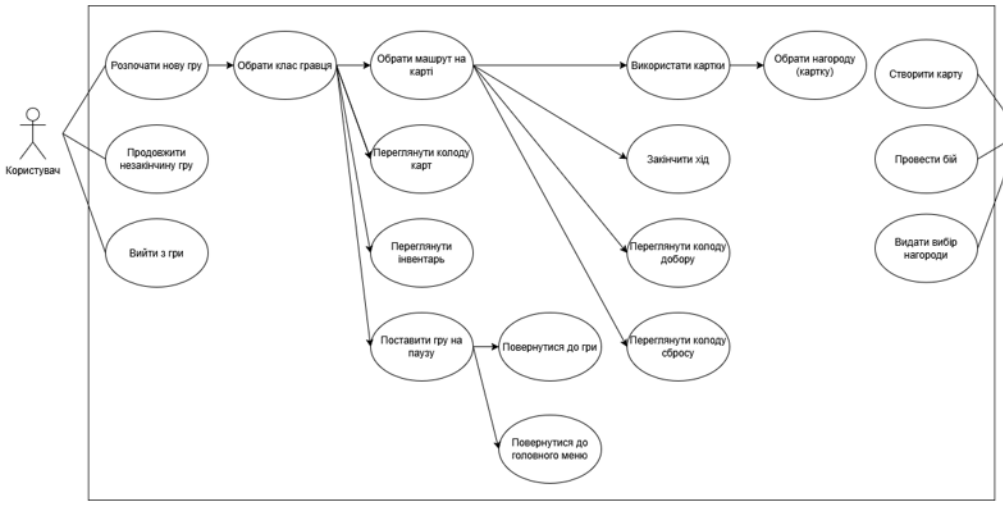
5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Безкоштовні ігрові ресурси

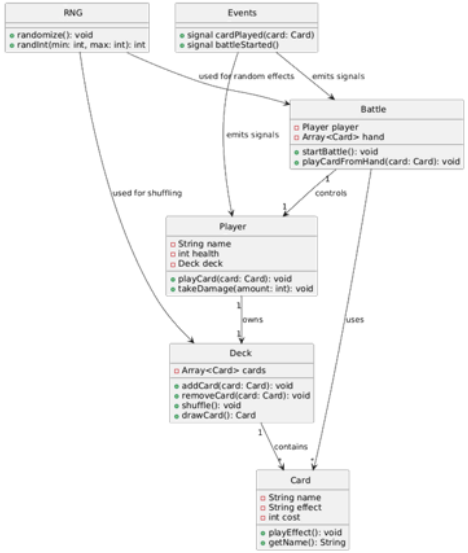
6

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



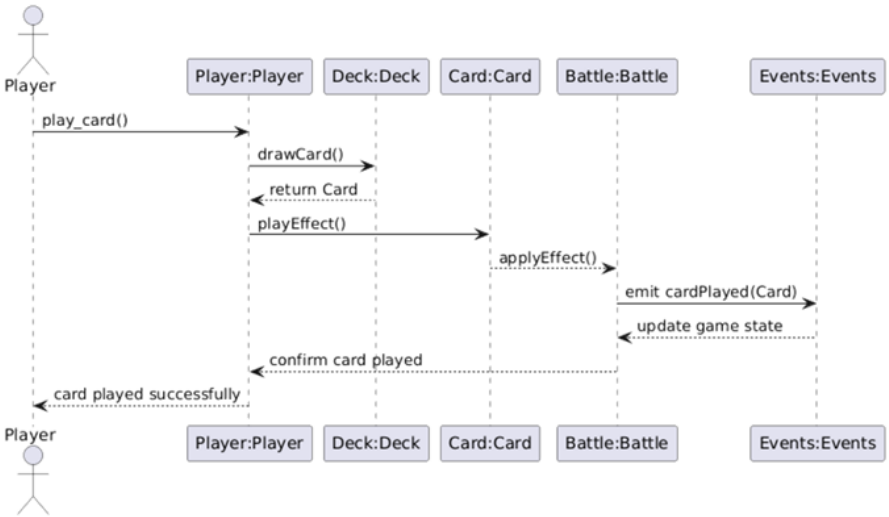
7

ДІАГРАМА КЛАСІВ



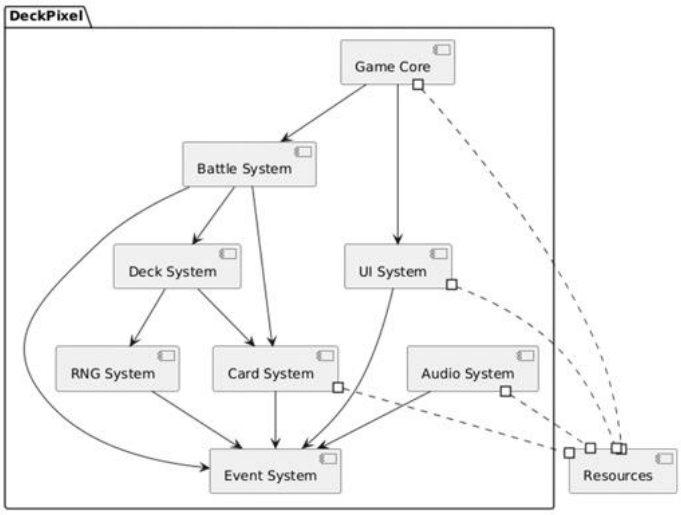
8

ДІАГРАМА ПОСЛІДОВНОСТІ



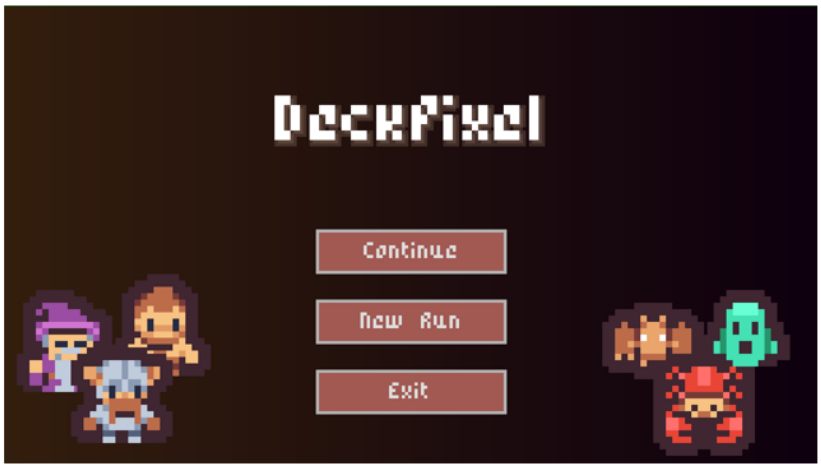
9

АРХІТЕКТУРА ПРОЄКТУ



10

ЕКРАННІ ФОРМИ



Головне меню гри

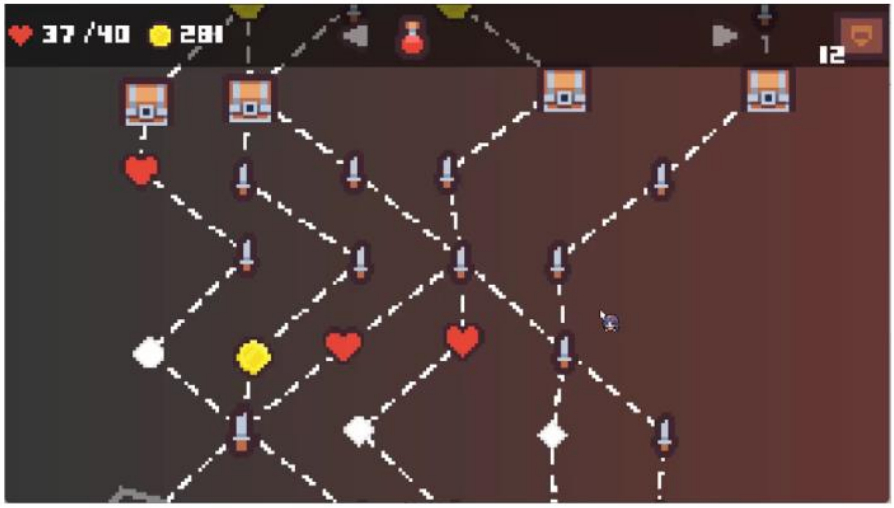
11

ЕКРАННІ ФОРМИ

Екран бою

12

ДЕМОНСТРАЦІЯ ГРИ



13

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Ясінський А.Б., Ярошевський О.В. Розробка відеогри в жанрі roguelike deck-building на рушії Godot Engine. Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2024 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2024, С. 245 - 246
2. Ясінський А.Б., Ярошевський О.В. Аналіз інструментів рушія godot для розробки ROGUELIKE DECK-BUILDING гри. Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24 квітня 2024 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2024, С. 247 - 248

14

ВИСНОВКИ

1. Аналіз Godot Engine показав його придатність для проєкту завдяки модульності, підтримці GDScript і гнучкості для реалізації генератора рівнів, системи колоди та бойової логіки.
2. Аналіз механік жанру допоміг інтегрувати динамічну генерацію контенту та стратегічну взаємодію, що відповідає очікуванням аудиторії.
3. Архітектура гри спроєктована модульно, з чітким розділенням компонентів, що полегшує розширення та підтримку.
4. Ключові модулі — генератор рівнів, система колоди, бойовий менеджер — реалізовані відповідно до вимог.
5. Користувацький інтерфейс інтуїтивний, з відображенням стану гри та підтримкою збереження прогресу.
6. Тестування підтвердило відповідність вимогам, надійність і можливість розширення, забезпечуючи якісний ігровий досвід.

15

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Вихідний код файлу battle_stats.gd

```
class_name BattleStats
extends Resource
@export_range(0, 2) var battle_tier: int
@export_range(0.0, 10.0) var weight: float
@export var gold_reward_min: int
@export var gold_reward_max: int
@export var enemies: PackedScene
var accumulated_weight: float = 0.0
func roll_gold_reward() -> int:
	return RNG.instance.randi_range(gold_reward_min,
gold_reward_max)
```

Вихідний код файлу battle_stats_pool.gd

```
class_name BattleStatsPool
extends Resource
@export var pool: Array[BattleStats]
var total_weights_by_tier := [0.0, 0.0, 0.0]
func _get_all_battles_for_tier(tier: int) -> Array[BattleStats]:
	return pool.filter(
		func(battle: BattleStats):
			return battle.battle_tier == tier
	)
func _setup_weight_for_tier(tier: int) -> void:
	var battles := _get_all_battles_for_tier(tier)
	total_weights_by_tier[tier] = 0.0
	for battle: BattleStats in battles:
		total_weights_by_tier[tier] += battle.weight
		battle.accumulated_weight =
total_weights_by_tier[tier]
func get_random_battle_for_tier(tier: int) -> BattleStats:
	var roll := randf_range(0.0,
total_weights_by_tier[tier])
	var battles := _get_all_battles_for_tier(tier)
	for battle: BattleStats in battles:
		if battle.accumulated_weight > roll:
			return battle
	return null
func setup() -> void:
	for i in 3:
		_setup_weight_for_tier(i)
```

Вихідний код файлу card.gd

```
class_name Card
extends Resource
enum Type {ATTACK, SKILL, POWER}
enum Rarity {COMMON, UNCOMMON, RARE}
enum Target {SELF, SINGLE_ENEMY, ALL_ENEMIES,
EVERYONE}
const RARITY_COLORS := {
	Card.Rarity.COMMON: Color.GRAY,
	Card.Rarity.UNCOMMON:
Color.CORNFLOWER_BLUE,
	Card.Rarity.RARE: Color.GOLD,
}
@export_group("Card Attributes")
@export var id: String
@export var type: Type
@export var rarity: Rarity
@export var target: Target
@export var cost: int
@export var exhausts: bool = false
@export_group("Card Visuals")
@export var icon: Texture
@export_multiline var tooltip_text: String
@export var sound: AudioStream
func is_single_targeted() -> bool:
	return target == Target.SINGLE_ENEMY
func _get_targets(targets: Array[Node]) -> Array[Node]:
	if not targets:
		return []
	var tree := targets[0].get_tree()
	match target:
		Target.SELF:
			return
tree.get_nodes_in_group("player")
		Target.ALL_ENEMIES:
			return
tree.get_nodes_in_group("enemies")
		Target.EVERYONE:
			return
tree.get_nodes_in_group("player") +
tree.get_nodes_in_group("enemies")_:
			return []
func play(targets: Array[Node], char_stats: CharacterStats,
modifiers: ModifierHandler) -> void:
	Events.card_played.emit(self)
	char_stats.mana -= cost

	if is_single_targeted():
		apply_effects(targets, modifiers)
	else:
		apply_effects(_get_targets(targets),
modifiers)
func apply_effects(_targets: Array[Node], _modifiers:
ModifierHandler) -> void:
	pass
func get_default_tooltip() -> String:
	return tooltip_text
func get_updated_tooltip(_player_modifiers: ModifierHandler,
_enemy_modifiers: ModifierHandler) -> String:
	return tooltip_text
```

Вихідний код файлу card_pile.gd

```
class_name CardPile
extends Resource
signal card_pile_size_changed(cards_amount)
@export var cards: Array[Card] = []
func empty() -> bool:
	card_pile_size_changed.emit(cards.size())
func duplicate_cards() -> Array[Card]:
	var new_array: Array[Card] = []
	for card: Card in cards:
		new_array.append(card.duplicate())
```

```
        return cards.is_empty()
func draw_card() -> Card:
        var card = cards.pop_front()
        card_pile_size_changed.emit(cards.size())
        return card
func add_card(card: Card) -> void:
        cards.append(card)
        card_pile_size_changed.emit(cards.size())
func shuffle() -> void:
        RNG.array_shuffle(cards)
func clear() -> void:
        cards.clear()
        return new_array
func custom_duplicate() -> CardPile:
        var new_card_pile := CardPile.new()
        new_card_pile.cards = duplicate_cards()
        return new_card_pile
func _to_string() -> String:
        var _card_strings: PackedStringArray = []
        for i in range(cards.size()):
                _card_strings.append("%s: %s" % [i+1,
cards[i].id])
        return "\n".join(_card_strings)
```

Вихідний код файлу character_stats.gd

```
class_name CharacterStats
extends Stats
@export_group("Visuals")
@export var character_name: String
@export_multiline var description: String
@export var portrait: Texture
@export_group("Gameplay Data")
@export var starting_deck: CardPile
@export var draftable_cards: CardPile
@export var cards_per_turn: int
@export var max_mana: int
@export var starting_relic: Relic
var mana: int : set = set_mana
var deck: CardPile
var discard: CardPile
var draw_pile: CardPile
func set_mana(value: int) -> void:
        mana = value
        stats_changed.emit()
func reset_mana() -> void:
        mana = max_mana
func take_damage(damage: int) -> void:
        var initial_health := health
        super.take_damage(damage)
        if initial_health > health:
                Events.player_hit.emit()
func can_play_card(card: Card) -> bool:
        return mana >= card.cost
func create_instance() -> Resource:
        var instance: CharacterStats = self.duplicate()
        instance.health = max_health
        instance.block = 0
        instance.reset_mana()
        instance.deck = instance.starting_deck.duplicate()
        instance.draw_pile = CardPile.new()
        instance.discard = CardPile.new()
        return instance
```

Вихідний код файлу effect.gd

```
class_name Effect
extends RefCounted
var sound: AudioStream
func execute(_targets: Array[Node]) -> void:
        pass
```

Вихідний код файлу enemy_stats.gd

```
class_name EnemyStats
extends Stats
@export var ai: PackedScene
```

Вихідний код файлу event_room_pool.gd

```
class_name EventRoomPool
extends Resource
@export var event_rooms: Array[PackedScene]
func get_random() -> PackedScene:
        return event_rooms.pick_random(
```

Вихідний код файлу intent.gd

```
class_name Intent
extends Resource
@export var base_text: String
@export var icon: Texture
var current_text: String
```

Вихідний код файлу relic.gd

```
class_name Relic
extends Resource
enum Type {START_OF_TURN, START_OF_COMBAT,
END_OF_TURN, END_OF_COMBAT, EVENT_BASED}
enum CharacterType {ALL, ASSASSIN, WARRIOR,
WIZARD}
@export var relic_name: String
@export var id: String
@export var type: Type
@export var character_type: CharacterType
@export var starter_relic: bool = false
@export var icon: Texture
@export_multiline var tooltip: String
func initialize_relic(_owner: RelicUI) -> void:
        pass
func deactivate_relic(_owner: RelicUI) -> void:
        pass
func get_tooltip() -> String:
        return tooltip
func can_appear_as_reward(character: CharacterStats) -> bool:
        if starter_relic:
                return false
        if character_type == CharacterType.ALL:
                return true
        var relic_char_name: String =
CharacterType.keys()[character_type].to_lower()
        var char_name :=
character.character_name.to_lower()
        return relic_char_name == char_name
```

```
        pass
```

Вихідний код файлу room.gd

```
class_name Room
extends Resource
enum Type {NOT_ASSIGNED, MONSTER, TREASURE,
CAMPFIRE, SHOP, BOSS, EVENT}
@export var type: Type
@export var row: int
@export var column: int
@export var position: Vector2
@export var next_rooms: Array[Room]
@export var selected := false
@export var battle_stats: BattleStats
@export var event_scene: PackedScene
func _to_string() -> String:
        return "%s (%s)" % [column, Type.keys()[type][0]]
```

Вихідний код файлу run_stats.gd

```
class_name RunStats
extends Resource
signal gold_changed
const STARTING_GOLD := 70
const BASE_CARD_REWARDS := 3
const BASE_COMMON_WEIGHT := 6.0
const BASE_UNCOMMON_WEIGHT := 3.7
const BASE_RARE_WEIGHT := 0.3
@export var gold := STARTING_GOLD : set = set_gold
@export var card_rewards := BASE_CARD_REWARDS
@export_range(0.0, 10.0) var common_weight :=
BASE_COMMON_WEIGHT
@export_range(0.0, 10.0) var uncommon_weight :=
BASE_UNCOMMON_WEIGHT
@export_range(0.0, 10.0) var rare_weight :=
BASE_RARE_WEIGHT
func set_gold(new_amount: int) -> void:
        gold = new_amount
        gold_changed.emit()
func reset_weights() -> void:
        common_weight = BASE_COMMON_WEIGHT
        uncommon_weight =
BASE_UNCOMMON_WEIGHT
        rare_weight = BASE_RARE_WEIGHT
```

Вихідний код файлу save_game.gd

```
class_name SaveGame
extends Resource
const SAVE_PATH := "user://savegame.tres"
@export var rng_seed: int
@export var rng_state: int
@export var run_stats: RunStats
@export var char_stats: CharacterStats
@export var current_deck: CardPile
@export var current_health: int
@export var relics: Array[Relic]
@export var map_data: Array[Array]
@export var last_room: Room
@export var floors_climbed: int
@export var was_on_map: bool
func save_data() -> void:
        var err := ResourceSaver.save(self, SAVE_PATH)
        assert(err == OK, "Couldn't save the game!")
static func load_data() -> SaveGame:
        if FileAccess.file_exists(SAVE_PATH):
                return ResourceLoader.load(SAVE_PATH)
as SaveGame
        return null
static func delete_data() -> void:
        if FileAccess.file_exists(SAVE_PATH):
        DirAccess.remove_absolute(SAVE_PATH)
```

Вихідний код файлу stats.gd

```
class_name Stats
extends Resource
signal stats_changed
@export var max_health := 1 : set = set_max_health
@export var art: Texture
var health: int : set = set_health
var block: int : set = set_block
func set_health(value : int) -> void:
        health = clampi(value, 0, max_health)
        stats_changed.emit()
func set_max_health(value : int) -> void:
        var diff := value - max_health
        max_health = value
        if diff > 0:
                health += diff
        elif health > max_health:
                health = max_health
        stats_changed.emit()
func set_block(value : int) -> void:
        block = clampi(value, 0, 999)
        stats_changed.emit()
func take_damage(damage : int) -> void:
        if damage <= 0:
                return
        var initial_damage = damage
        damage = clampi(damage - block, 0, damage)
        block = clampi(block - initial_damage, 0, block)
        health -= damage
func heal(amount : int) -> void:
        health += amount
func create_instance() -> Resource:
        var instance: Stats = self.duplicate()
        instance.health = max_health
        instance.block = 0
        return instance
```

Вихідний код файлу events.gd

```
extends Node
signal card_drag_started(card_ui: CardUI)
signal card_drag_ended(card_ui: CardUI)
signal card_aim_started(card_ui: CardUI)
signal card_aim_ended(card_ui: CardUI)
signal card_played(card: Card)
signal enemy_died(enemy: Enemy)
signal battle_over_screen_requested(text: String, type:
BattleOverPanel.Type)
signal battle_won
signal status_tooltip_requested(statuses: Array[Status])
signal map_exited(room: Room)
```

```
signal card_tooltip_requested(card: Card)
signal tooltip_hide_requested
signal player_hand_drawn
signal player_hand_discarded
signal player_turn_ended
signal player_hit
signal player_died
signal enemy_action_completed(enemy: Enemy)
signal enemy_turn_ended
signal shop_entered(shop: Shop)
signal shop_relic_bought(relic: Relic, gold_cost: int)
signal shop_card_bought(card: Card, gold_cost: int)
signal shop_exited
signal campfire_exited
signal battle_reward_exited
signal treasure_room_exited(found_relic: Relic)
signal relic_tooltip_requested(relic: Relic)
signal event_room_exited
```

Вихідний код файлу battle.gd

```
class_name Battle
extends Node2D
@export var battle_stats: BattleStats
@export var char_stats: CharacterStats
@export var music: AudioStream
@export var relics: RelicHandler
@onready var battle_ui: BattleUI = $BattleUI
@onready var player_handler: PlayerHandler = $PlayerHandler
@onready var enemy_handler: EnemyHandler = $EnemyHandler
@onready var player: Player = $Player
func _ready() -> void:
	enemy_handler.child_order_changed.connect(_on_enemies_child_order_changed)
	Events.enemy_turn_ended.connect(_on_enemy_turn_ended)
	Events.player_turn_ended.connect(player_handler.end_turn)
	Events.player_hand_discarded.connect(enemy_handler.start_turn)
	Events.player_died.connect(_on_player_died)
func start_battle() -> void:
	get_tree().paused = false
	MusicPlayer.play(music, true)
	battle_ui.char_stats = char_stats
	player.stats = char_stats
	player_handler.relics = relics
	enemy_handler.setup_enemies(battle_stats)
	enemy_handler.reset_enemy_actions()
	relics.relics_activated.connect(_on_relics_activated)
	relics.activate_relics_by_type(Relic.Type.START_OF_COMBAT)
func _on_enemies_child_order_changed() -> void:
	if enemy_handler.get_child_count() == 0 and is_instance_valid(relics):
		relics.activate_relics_by_type(Relic.Type.END_OF_COMBAT)
func _on_enemy_turn_ended() -> void:
	player_handler.start_turn()
	enemy_handler.reset_enemy_actions()
func _on_player_died() -> void:
	Events.battle_over_screen_requested.emit("Game Over!", BattleOverPanel.Type.LOSE)
	SaveGame.delete_data()
func _on_relics_activated(type: Relic.Type) -> void:
	match type:
	Relic.Type.START_OF_COMBAT:
	player_handler.start_battle(char_stats)
		battle_ui.initialize_card_pile_ui()
		Relic.Type.END_OF_COMBAT:
	Events.battle_over_screen_requested.emit("Victorious!", BattleOverPanel.Type.WIN)
```

Вихідний код файлу card_base_state.gd

```
extends CardState
var mouse_over_card := false
func enter() -> void:
	if not card_ui.is_node_ready():
		await card_ui.ready
	if card_ui.tween and card_ui.tween.is_running():
		card_ui.tween.kill()
	card_ui.card_visuals.panel.set("theme_override_styles/panel", card_ui.BASE_STYLEBOX)
	card_ui.reparent_requested.emit(card_ui)
	card_ui.pivot_offset = Vector2.ZERO
	Events.tooltip_hide_requested.emit()
func on_gui_input(event: InputEvent) -> void:
	if not card_ui.playable or card_ui.disabled:
		return
	if mouse_over_card and event.is_action_pressed("left_mouse"):
		card_ui.pivot_offset = card_ui.get_global_mouse_position() - card_ui.global_position
		transition_requested.emit(self, CardState.State.CLICKED)
func on_mouse_entered() -> void:
	mouse_over_card = true
	if not card_ui.playable or card_ui.disabled:
		return
	card_ui.card_visuals.panel.set("theme_override_styles/panel", card_ui.HOVER_STYLEBOX)
	card_ui.request_tooltip()
func on_mouse_exited() -> void:
	mouse_over_card = false
	if not card_ui.playable or card_ui.disabled:
		return
	card_ui.card_visuals.panel.set("theme_override_styles/panel", card_ui.BASE_STYLEBOX)
	Events.tooltip_hide_requested.emit()
```

Вихідний код файлу card_ui.gd

```
class_name CardUI
extends Control
signal reparent_requested(which_card_ui: CardUI)
const BASE_STYLEBOX := preload("res://scenes/card_ui/card_base_stylebox.tres")
const DRAG_STYLEBOX := preload("res://scenes/card_ui/card_drag_stylebox.tres")
const HOVER_STYLEBOX := preload("res://scenes/card_ui/card_hover_stylebox.tres")
@export var player_modifiers: ModifierHandler
	rect.has_point(get_local_mouse_position())
func request_tooltip() -> void:
	var enemy_modifiers := get_active_enemy_modifiers()
	var updated_tooltip := card.get_updated_tooltip(player_modifiers, enemy_modifiers)
	Events.card_tooltip_requested.emit(card.icon, updated_tooltip)
func _on_gui_input(event: InputEvent) -> void:
	card_state_machine.on_gui_input(event)
```

```
@export var card: Card : set = _set_card
@export var char_stats: CharacterStats : set = _set_char_stats
@onready var card_visuals: CardVisuals = $CardVisuals
@onready var drop_point_detector: Area2D = $DropPointDetector
@onready var card_state_machine: CardStateMachine =
$CardStateMachine
@onready var targets: Array[Node] = []
var original_index := 0
var parent: Control
var tween: Tween
var playable := true : set = _set_playable
var disabled := true
func _ready() -> void:
	Events.card_aim_started.connect(_on_card_drag_or_aimi
ng_started)
	Events.card_drag_started.connect(_on_card_drag_or_aimi
ng_started)
	Events.card_drag_ended.connect(_on_card_drag_or_aim_
ended)
	Events.card_aim_ended.connect(_on_card_drag_or_aim_e
nded)
	card_state_machine.init(self)
func _input(event: InputEvent) -> void:
	card_state_machine.on_input(event)
func animate_to_position(new_position: Vector2, duration: float) ->
void:
	tween =
create_tween().set_trans(Tween.TRANS_CIRC).set_ease(Tween.E
ASE_OUT)
	tween.tween_property(self, "global_position",
new_position, duration)
func play() -> void:
	if not card:
		return
	card.play(targets, char_stats, player_modifiers)
	queue_free()
func get_active_enemy_modifiers() -> ModifierHandler:
	if targets.is_empty() or targets.size() > 1 or not targets[0]
is Enemy:
		return null
	return targets[0].modifier_handler
func is_hovered() -> bool:
	var rect := Rect2(Vector2.ZERO, self.size)
	return
func _on_mouse_entered() -> void:
	card_state_machine.on_mouse_entered()
func _on_mouse_exited() -> void:
	card_state_machine.on_mouse_exited()
func _set_card(value: Card) -> void:
	if not is_node_ready():
		await ready
	card = value
	card_visuals.card = card
func _set_playable(value: bool) -> void:
	playable = value
	if not playable:
	card_visuals.cost.add_theme_color_override("font_c
olor", Color.RED)
		card_visuals.icon.modulate = Color(1, 1,
1, 0.5)
	else:
	card_visuals.cost.remove_theme_color_override("fo
nt_color")
		card_visuals.icon.modulate = Color(1, 1,
1, 1)
func _set_char_stats(value: CharacterStats) -> void:
	char_stats = value
	char_stats.stats_changed.connect(_on_char_stats_ch
anged)
	_on_char_stats_changed()
func _on_drop_point_detector_area_entered(area: Area2D) ->
void:
	if not targets.has(area):
		targets.append(area)
func _on_drop_point_detector_area_exited(area: Area2D) ->
void:
	targets.erase(area)
func _on_card_drag_or_aiming_started(used_card: CardUI) -
> void:
	if used_card == self:
		return

	disabled = true
func _on_card_drag_or_aim_ended(_card: CardUI) -> void:
	disabled = false
	playable = char_stats.can_play_card(card)
func _on_char_stats_changed() -> void:
	playable = char_stats.can_play_card(card)
```

Вихідний код файлу enemy.gd

```
class_name Enemy
extends Area2D
const ARROW_OFFSET := 5
const WHITE_SPRITE_MATERIAL :=
preload("res://art/white_sprite_material.tres")
@export var stats: EnemyStats : set = set_enemy_stats
@onready var sprite_2d: Sprite2D = $Sprite2D
@onready var arrow: Sprite2D = $Arrow
@onready var stats_ui: StatsUI = $StatsUI
@onready var intent_ui: IntentUI = $IntentUI
@onready var status_handler: StatusHandler =
$StatusHandler
@onready var modifier_handler: ModifierHandler =
$ModifierHandler
var enemy_action_picker: EnemyActionPicker
var current_action: EnemyAction : set =
set_current_action
func _ready() -> void:
	status_handler.status_owner = self
func set_current_action(value: EnemyAction) -> void:
	current_action = value
	update_intent()
func set_enemy_stats(value: EnemyStats) -> void:
	if not current_action:
		current_action = enemy_action_picker.get_action()
		return
	var new_conditional_action :=
enemy_action_picker.get_first_conditional_action()
	if new_conditional_action and current_action !=
new_conditional_action:
		current_action = new_conditional_action
func update_enemy() -> void:
	if not stats is Stats:
		return
	if not is_inside_tree():
		await ready
	sprite_2d.texture = stats.art
	arrow.position = Vector2.RIGHT * (sprite_2d.get_rect().size.x / 2
+ ARROW_OFFSET)
	setup_ai()
	update_stats()
func update_intent() -> void:
	if current_action:
		current_action.update_intent_text()
		intent_ui.update_intent(current_action.intent)
func do_turn() -> void:
```

```
		stats = value.create_instance()
		if not
stats.stats_changed.is_connected(update_stats):
		stats.stats_changed.connect(update_stats)
		stats.stats_changed.connect(update_action)
		update_enemy()
func setup_ai() -> void:
		if enemy_action_picker:
		enemy_action_picker.queue_free()
		var new_action_picker :=
stats.ai.instantiate() as EnemyActionPicker
		add_child(new_action_picker)
		enemy_action_picker = new_action_picker
		enemy_action_picker.enemy = self
func update_stats() -> void:
		stats_ui.update_stats(stats)
func update_action() -> void:
		if not enemy_action_picker:
			return
```

```
class_name Map
extends Node2D
const SCROLL_SPEED := 15
const MAP_ROOM =
preload("res://scenes/map/map_room.tscn")
const MAP_LINE =
preload("res://scenes/map/map_line.tscn")
@onready var map_generator: MapGenerator =
$MapGenerator
@onready var lines: Node2D = %Lines
@onready var rooms: Node2D = %Rooms
@onready var visuals: Node2D = $Visuals
@onready var camera_2d: Camera2D = $Camera2D
var map_data: Array[Array]
var floors_climbed: int
var last_room: Room
var camera_edge_y: float
func _ready() -> void:
		camera_edge_y = MapGenerator.Y_DIST *
(MapGenerator.FLOORS - 1)
func _unhandled_input(event: InputEvent) -> void:
		if not visible:
			return
		if event.is_action_pressed("scroll_up"):
			camera_2d.position.y -=
SCROLL_SPEED
		elif event.is_action_pressed("scroll_down"):
			camera_2d.position.y +=
SCROLL_SPEED
		camera_2d.position.y =
clamp(camera_2d.position.y, -camera_edge_y, 0)
func generate_new_map() -> void:
		floors_climbed = 0
		map_data = map_generator.generate_map()
		create_map()
func load_map(map: Array[Array], floors_completed: int,
last_room_climbed: Room) -> void:
		floors_climbed = floors_completed
		map_data = map
		last_room = last_room_climbed
		create_map()
		if floors_climbed > 0:
```

```
		stats.block = 0
		if not current_action:
			return
		current_action.perform_action()
func take_damage(damage: int, which_modifier: Modifier.Type) -> void:
		if stats.health <= 0:
			return
		sprite_2d.material = WHITE_SPRITE_MATERIAL
		var modified_damage :=
modifier_handler.get_modified_value(damage, which_modifier)
		var tween := create_tween()
		tween.tween_callback(Shaker.shake.bind(self, 16, 0.15))
		tween.tween_callback(stats.take_damage.bind(modified_damage))
		tween.tween_interval(0.17)
		tween.finished.connect(
			func():
				sprite_2d.material = null
				if stats.health <= 0:
					Events.enemy_died.emit(self)
					queue_free()
		)
func _on_area_entered(_area: Area2D) -> void:
		arrow.show()
func _on_area_exited(_area: Area2D) -> void:
		arrow.hide()
```

Вихідний код файлу map.gd

```
	map_data:
			for room: Room in current_floor:
				if room.next_rooms.size() > 0:
		_spawn_room(room)
		var middle :=
	floori(MapGenerator.MAP_WIDTH * 0.5)
		_spawn_room(map_data[MapGenerator.FLOORS-
	1][middle])
		var map_width_pixels := MapGenerator.X_DIST *
	(MapGenerator.MAP_WIDTH - 1)
		visuals.position.x = (get_viewport_rect().size.x -
	map_width_pixels) / 2
		visuals.position.y = get_viewport_rect().size.y / 2
	func unlock_floor(which_floor: int = floors_climbed) -> void:
		for map_room: MapRoom in rooms.get_children():
			if map_room.room.row == which_floor:
				map_room.available = true
	func unlock_next_rooms() -> void:
		for map_room: MapRoom in rooms.get_children():
			if last_room.next_rooms.has(map_room.room):
				map_room.available = true
	func show_map() -> void:
		show()
		camera_2d.enabled = true
	func hide_map() -> void:
		hide()
		camera_2d.enabled = false
	func _spawn_room(room: Room) -> void:
		var new_map_room := MAP_ROOM.instantiate() as
	MapRoom
		rooms.add_child(new_map_room)
		new_map_room.room = room
		new_map_room.clicked.connect(_on_map_room_clicked)
		new_map_room.selected.connect(_on_map_room_selected)
		_connect_lines(room)
		if room.selected and room.row < floors_climbed:
			new_map_room.show_selected()
	func _connect_lines(room: Room) -> void:
		if room.next_rooms.is_empty():
			return
		for next: Room in room.next_rooms:
```

```
                unlock_next_rooms()
        else:
                unlock_floor()
func create_map() -> void:
        for current_floor: Array in
                var new_map_line := MAP_LINE.instantiate() as
Line2D
                new_map_line.add_point(room.position)
                new_map_line.add_point(next.position)
                lines.add_child(new_map_line)
func _on_map_room_clicked(room: Room) -> void:
        for map_room: MapRoom in rooms.get_children():
                if map_room.room.row == room.row:
                        map_room.available = false
func _on_map_room_selected(room: Room) -> void:
        last_room = room
        floors_climbed += 1
        Events.map_exited.emit(room)
```

Вихідний код файлу map_generator.gd

```
class_name MapGenerator
extends Node
const X_DIST := 30
const Y_DIST := 25
const PLACEMENT_RANDOMNESS := 5
const FLOORS := 15
const MAP_WIDTH := 7
const PATHS := 6
const MONSTER_ROOM_WEIGHT := 12.0
const EVENT_ROOM_WEIGHT := 5.0
const SHOP_ROOM_WEIGHT := 2.5
const CAMPFIRE_ROOM_WEIGHT := 4.0
@export var battle_stats_pool: BattleStatsPool
@export var event_room_pool: EventRoomPool
var random_room_type_weights = {
        Room.Type.MONSTER: 0.0,
        Room.Type.CAMPFIRE: 0.0,
        Room.Type.SHOP: 0.0,
        Room.Type.EVENT: 0.0
}
var random_room_type_total_weight := 0
var map_data: Array[Array]
func generate_map() -> Array[Array]:
        map_data = _generate_initial_grid()
        var starting_points :=
_get_random_starting_points()
        for j in starting_points:
                var current_j := j
                for i in FLOORS - 1:
                        current_j =
_setup_connection(i, current_j)
        battle_stats_pool.setup()
        _setup_boss_room()
        _setup_random_room_weights()
        _setup_room_types()
        return map_data
func _generate_initial_grid() -> Array[Array]:
        var result: Array[Array] = []
        for i in FLOORS:
                var adjacent_rooms: Array[Room]= []
                for j in MAP_WIDTH:
                        var current_room :=
Room.new()
                        var offset := Vector2(randf(),
randf()) * PLACEMENT_RANDOMNESS
                        current_room.position =
Vector2(j * X_DIST, i * -Y_DIST) + offset
                        current_room.row = i
                        current_room.column = j
        current_room.next_rooms = []

                        if i == FLOORS - 1:
        current_room.position.y = (i + 1) * -Y_DIST
        adjacent_rooms.append(current_room)
                var middle := floori(MAP_WIDTH * 0.5)
                var boss_room := map_data[FLOORS - 1][middle] as Room
                for j in MAP_WIDTH:
                        var current_room = map_data[FLOORS - 2][j] as
Room
                        if current_room.next_rooms:
                                current_room.next_rooms = [] as
Array[Room]
                current_room.next_rooms.append(boss_room)
                boss_room.type = Room.Type.BOSS
                boss_room.battle_stats =
battle_stats_pool.get_random_battle_for_tier(2)
func _setup_random_room_weights() -> void:
        random_room_type_weights[Room.Type.MONSTER] =
MONSTER_ROOM_WEIGHT
        random_room_type_weights[Room.Type.CAMPFIRE] =
MONSTER_ROOM_WEIGHT + CAMPFIRE_ROOM_WEIGHT
        random_room_type_weights[Room.Type.SHOP] =
MONSTER_ROOM_WEIGHT + CAMPFIRE_ROOM_WEIGHT +
SHOP_ROOM_WEIGHT
        random_room_type_weights[Room.Type.EVENT] =
random_room_type_weights[Room.Type.SHOP] +
EVENT_ROOM_WEIGHT
        random_room_type_total_weight =
random_room_type_weights[Room.Type.EVENT]
func _setup_room_types() -> void:
        for room: Room in map_data[0]:
                if room.next_rooms.size() > 0:
                        room.type =
Room.Type.MONSTER
                        room.battle_stats =
battle_stats_pool.get_random_battle_for_tier(0)
        for room: Room in map_data[8]:
                if room.next_rooms.size() > 0:
                        room.type =
Room.Type.TREASURE
        for room: Room in map_data[13]:
                if room.next_rooms.size() > 0:
                        room.type =
Room.Type.CAMPFIRE
        for current_floor in map_data:
                for room: Room in current_floor:
                        for next_room: Room in
room.next_rooms:
                                if next_room.type ==
Room.Type.NOT_ASSIGNED:
        _set_room_randomly(next_room)
func _set_room_randomly(room_to_set: Room) -> void:
        var campfire_below_4 := true
        var consecutive_campfire := true
        var consecutive_shop := true
        var campfire_on_13 := true
        var type_candidate: Room.Type
```

```
                result.append(adjacent_rooms)
        return result
func _get_random_starting_points() -> Array[int]:
        var y_coordinates: Array[int]
        var unique_points: int = 0
        while unique_points < 2:
                unique_points = 0
                y_coordinates = []
                for i in PATHS:
                        var starting_point :=
randi_range(0, MAP_WIDTH - 1)
                        if not
y_coordinates.has(starting_point):
                                unique_points += 1
        y_coordinates.append(starting_point)

        return y_coordinates
func _setup_connection(i: int, j: int) -> int:
        var next_room: Room = null
        var current_room := map_data[i][j] as Room
        while not next_room or
_would_cross_existing_path(i, j, next_room):
                var random_j := clampi(randi_range(j -
1, j + 1), 0, MAP_WIDTH - 1)
                next_room = map_data[i + 1][random_j]
        current_room.next_rooms.append(next_room)
        return next_room.column
func _would_cross_existing_path(i: int, j: int, room: Room)
-> bool:
        var left_neighbour: Room
        var right_neighbour: Room
        if j > 0:
                left_neighbour = map_data[i][j - 1]
        if j < MAP_WIDTH - 1:
                right_neighbour = map_data[i][j + 1]
        if right_neighbour and room.column > j:
                for next_room: Room in
right_neighbour.next_rooms:
                        if next_room.column <
room.column:
                                return true
        if left_neighbour and room.column < j:
                for next_room: Room in
left_neighbour.next_rooms:
                        if next_room.column >
room.column:
                                return true
        return false
func _setup_boss_room() -> void:
        while campfire_below_4 or consecutive_campfire or
consecutive_shop or campfire_on_13:
                type_candidate =
_get_random_room_type_by_weight()
                var is_campfire := type_candidate ==
Room.Type.CAMPFIRE
                var has_campfire_parent :=
_room_has_parent_of_type(room_to_set, Room.Type.CAMPFIRE)
                var is_shop := type_candidate ==
Room.Type.SHOP
                var has_shop_parent :=
_room_has_parent_of_type(room_to_set, Room.Type.SHOP)
                campfire_below_4 = is_campfire and
room_to_set.row < 3
                consecutive_campfire = is_campfire and
has_campfire_parent
                consecutive_shop = is_shop and has_shop_parent
                campfire_on_13 = is_campfire and
room_to_set.row == 12
        room_to_set.type = type_candidate
        if type_candidate == Room.Type.MONSTER:
                var tier_for_monster_rooms := 0
                if room_to_set.row > 2:
                        tier_for_monster_rooms = 1
                room_to_set.battle_stats =
battle_stats_pool.get_random_battle_for_tier(tier_for_monster_rooms)
        if type_candidate == Room.Type.EVENT:
                room_to_set.event_scene =
event_room_pool.get_random()
func _room_has_parent_of_type(room: Room, type: Room.Type) ->
bool:
        var parents: Array[Room] = []
        if room.column > 0 and room.row > 0:
                var parent_candidate := map_data[room.row -
1][room.column - 1] as Room
                if parent_candidate.next_rooms.has(room):
                        parents.append(parent_candidate)
        if room.row > 0:
                var parent_candidate := map_data[room.row -
1][room.column] as Room
                if parent_candidate.next_rooms.has(room):
                        parents.append(parent_candidate)
        if room.column < MAP_WIDTH-1 and room.row > 0:
                var parent_candidate := map_data[room.row -
1][room.column + 1] as Room
                if parent_candidate.next_rooms.has(room):
                        parents.append(parent_candidate)
        for parent: Room in parents:
                if parent.type == type:
                        return true
        return false
func _get_random_room_type_by_weight() -> Room.Type:
        var roll := randf_range(0.0,
random_room_type_total_weight)
        for type: Room.Type in random_room_type_weights:
                if random_room_type_weights[type] > roll:
                        return type
        return Room.Type.MONSTER
```

Вихідний код файлу player.gd

```
class_name Player
extends Node2D
const WHITE_SPRITE_MATERIAL :=
preload("res://art/white_sprite_material.tres")
@export var stats: CharacterStats : set =
set_character_stats
@onready var sprite_2d: Sprite2D = $Sprite2D
@onready var stats_ui: StatsUI = $StatsUI
        if not is_inside_tree():
                await ready
        sprite_2d.texture = stats.art
        update_stats()
func update_stats() -> void:
        stats_ui.update_stats(stats)
func take_damage(damage: int, which_modifier: Modifier.Type) -> void:
        if stats.health <= 0:
                return
```

```
@onready var status_handler: StatusHandler =
$StatusHandler
@onready var modifier_handler: ModifierHandler =
$ModifierHandler
func _ready() -> void:
	status_handler.status_owner = self
func set_character_stats(value: CharacterStats) ->
void:
	stats = value
	if not
stats.stats_changed.is_connected(update_stats):
	stats.stats_changed.connect(update_stats)
	update_player()
func update_player() -> void:
	if not stats is CharacterStats:
		return
	sprite_2d.material = WHITE_SPRITE_MATERIAL
	var modified_damage :=
modifier_handler.get_modified_value(damage, which_modifier)
	var tween := create_tween()
	tween.tween_callback(Shaker.shake.bind(self, 16, 0.15))
	tween.tween_callback(stats.take_damage.bind(modified_damage))
	tween.tween_interval(0.17)
	tween.finished.connect(
		func():
			sprite_2d.material = null

			if stats.health <= 0:
				Events.player_died.emit()
				queue_free()
	)
```

Вихідний код файлу pause_menu.gd

```
class_name PauseMenu
extends CanvasLayer
signal save_and_quit
@onready var back_to_game_button: Button = %BackToGameButton
@onready var save_and_quit_button: Button = %SaveAndQuitButton
func _ready() -> void:
	back_to_game_button.pressed.connect(_unpause)
	save_and_quit_button.pressed.connect(_on_save_and_quit_button_presse
d)
func _input(event: InputEvent) -> void:
	if event.is_action_pressed("pause"):
		if visible:
			_unpause()
		else:
			_pause()
	get_viewport().set_input_as_handled
()
func _pause() -> void:
	show()
	get_tree().paused = true
func _unpause() -> void:
	hide()
	get_tree().paused = false
func _on_save_and_quit_button_pressed() ->
void:
	get_tree().paused = false
	save_and_quit.emit()
```

Вихідний код файлу run.gd

```
class_name Run
extends Node
const BATTLE_SCENE :=
preload("res://scenes/battle/battle.tscn")
const BATTLE_REWARD_SCENE :=
preload("res://scenes/battle_reward/battle_reward.tsc
n")
const CAMPFIRE_SCENE :=
preload("res://scenes/campfire/campfire.tscn")
const SHOP_SCENE :=
preload("res://scenes/shop/shop.tscn")
const TREASURE_SCENE =
preload("res://scenes/treasure/treasure.tscn")
const WIN_SCREEN_SCENE :=
preload("res://scenes/win_screen/win_screen.tscn")
const MAIN_MENU_PATH :=
"res://scenes/ui/main_menu.tscn"
@export var run_startup: RunStartup
@onready var map: Map = $Map
@onready var current_view: Node = $CurrentView
@onready var health_ui: HealthUI = %HealthUI
@onready var gold_ui: GoldUI = %GoldUI
@onready var relic_handler: RelicHandler =
%RelicHandler
@onready var relic_tooltip: RelicTooltip =
%RelicTooltip
@onready var deck_button: CardPileOpener =
%DeckButton
@onready var deck_view: CardPileView =
%DeckView
@onready var pause_menu: PauseMenu =
$PauseMenu
@onready var battle_button: Button = %BattleButton
@onready var campfire_button: Button =
%CampfireButton
save_data.was_on_map:
	_on_map_exited(save_data.last_room)
func _change_view(scene: PackedScene) -> Node:
	if current_view.get_child_count() > 0:
	current_view.get_child(0).queue_free()
	get_tree().paused = false
	var new_view := scene.instantiate()
	current_view.add_child(new_view)
	map.hide_map()
	return new_view
func _show_map() -> void:
	if current_view.get_child_count() > 0:
	current_view.get_child(0).queue_free()
	map.show_map()
	map.unlock_next_rooms()
	_save_run(true)
func _setup_event_connections() -> void:
	Events.battle_won.connect(_on_battle_won)
	Events.battle_reward_exited.connect(_show_map)
	Events.campfire_exited.connect(_show_map)
	Events.map_exited.connect(_on_map_exited)
	Events.shop_exited.connect(_show_map)
	Events.treasure_room_exited.connect(_on_treasure_room_exited)
	Events.event_room_exited.connect(_show_map)
	battle_button.pressed.connect(_change_view.bind(BATTLE_SCE
NE))
	campfire_button.pressed.connect(_change_view.bind(CAMPFIRE
_SCENE))
	map_button.pressed.connect(_show_map)
	rewards_button.pressed.connect(_change_view.bind(BATTLE_RE
WARD_SCENE))
	shop_button.pressed.connect(_change_view.bind(SHOP_SCENE))
	treasure_button.pressed.connect(_change_view.bind(TREASURE_
SCENE))
func _setup_top_bar():
```

```
@onready var map_button: Button = %MapButton
@onready var rewards_button: Button = %RewardsButton
@onready var shop_button: Button = %ShopButton
@onready var treasure_button: Button = %TreasureButton
var stats: RunStats
var character: CharacterStats
var save_data: SaveGame
func _ready() -> void:
	if not run_startup:
		return
	pause_menu.save_and_quit.connect(
		func():
	get_tree().change_scene_to_file(MAIN_MENU_PATH)
	)
	match run_startup.type:
	RunStartup.Type.NEW_RUN:
			character = run_startup.picked_character.create_instance()
			_start_run()
	RunStartup.Type.CONTINUED_RUN:
			_load_run()
func _start_run() -> void:
	stats = RunStats.new()
	_setup_event_connections()
	_setup_top_bar()
	map.generate_new_map()
	map.unlock_floor(0)
	save_data = SaveGame.new()
	_save_run(true)
func _save_run(was_on_map: bool) -> void:
	save_data.rng_seed = RNG.instance.seed
	save_data.rng_state = RNG.instance.state
	save_data.run_stats = stats
	save_data.char_stats = character
	save_data.current_deck = character.deck
	save_data.current_health = character.health
	save_data.relics = relic_handler.get_all_relics()
	save_data.last_room = map.last_room
	save_data.map_data = map.map_data.duplicate()
	save_data.floors_climbed = map.floors_climbed
	save_data.was_on_map = was_on_map
	save_data.save_data()
func _load_run() -> void:
	save_data = SaveGame.load_data()
	assert(save_data, "Couldn't load last save")
	RNG.set_from_save_data(save_data.rng_seed, save_data.rng_state)
	stats = save_data.run_stats
	character = save_data.char_stats
	character.deck = save_data.current_deck
	character.health = save_data.current_health
	relic_handler.add_relics(save_data.relics)
	_setup_top_bar()
	_setup_event_connections()
	map.load_map(save_data.map_data, save_data.floors_climbed, save_data.last_room)
	if save_data.last_room and not
		character.stats_changed.connect(health_ui.update_stats.bind(character))
		health_ui.update_stats(character)
		gold_ui.run_stats = stats
		relic_handler.add_relic(character.starting_relic)
		Events.relic_tooltip_requested.connect(relic_tooltip.show_tooltip)
		deck_button.card_pile = character.deck
		deck_view.card_pile = character.deck
		deck_button.pressed.connect(deck_view.show_current_view.bind("Deck"))
func _show_regular_battle_rewards() -> void:
	var reward_scene := _change_view(BATTLE_REWARD_SCENE) as BattleReward
	reward_scene.run_stats = stats
	reward_scene.character_stats = character
	reward_scene.add_gold_reward(map.last_room.battle_stats.roll_gold_reward())
	reward_scene.add_card_reward()
func _on_battle_room_entered(room: Room) -> void:
	var battle_scene: Battle = _change_view(BATTLE_SCENE) as Battle
	battle_scene.char_stats = character
	battle_scene.battle_stats = room.battle_stats
	battle_scene.relics = relic_handler
	battle_scene.start_battle()
func _on_treasure_room_entered() -> void:
	var treasure_scene := _change_view(TREASURE_SCENE) as Treasure
	treasure_scene.relic_handler = relic_handler
	treasure_scene.char_stats = character
	treasure_scene.generate_relic()
func _on_treasure_room_exited(relic: Relic) -> void:
	var reward_scene := _change_view(BATTLE_REWARD_SCENE) as BattleReward
	reward_scene.run_stats = stats
	reward_scene.character_stats = character
	reward_scene.relic_handler = relic_handler
	reward_scene.add_relic_reward(relic)
func _on_campfire_entered() -> void:
	var campfire := _change_view(CAMPFIRE_SCENE) as Campfire
	campfire.char_stats = character
func _on_shop_entered() -> void:
	var shop := _change_view(SHOP_SCENE) as Shop
	shop.char_stats = character
	shop.run_stats = stats
	shop.relic_handler = relic_handler
	Events.shop_entered.emit(shop)
	shop.populate_shop()
func _on_event_room_entered(room: Room) -> void:
	var event_room := _change_view(room.event_scene) as EventRoom
	event_room.character_stats = character
	event_room.run_stats = stats
	event_room.setup()
func _on_battle_won() -> void:
	if map.floors_climbed == MapGenerator.FLOORS:
		var win_screen := _change_view(WIN_SCREEN_SCENE) as WinScreen
		win_screen.character = character
		SaveGame.delete_data()
	else:
		_show_regular_battle_rewards()
func _on_map_exited(room: Room) -> void:
	_save_run(false)
	match room.type:
		Room.Type.MONSTER:
			_on_battle_room_entered(room)
		Room.Type.TREASURE:
			_on_treasure_room_entered()
```

```
Room.Type.CAMPFIRE:
	_on_campfire_entered()
Room.Type.SHOP:
	_on_shop_entered()
Room.Type.BOSS:
	_on_battle_room_entered(room)
Room.Type.EVENT:
	_on_event_room_entered(room)
```

Вихідний код файлу shop.gd

```
class_name Shop
extends Control
const SHOP_CARD = preload("res://scenes/shop/shop_card.tscn")
const SHOP_RELIC = preload("res://scenes/shop/shop_relic.tscn")
@export var shop_relics: Array[Relic]
@export var char_stats: CharacterStats
@export var run_stats: RunStats
@export var relic_handler: RelicHandler
@onready var cards: HBoxContainer = %Cards
@onready var relics: HBoxContainer = %Relics
@onready var shop_keeper_animation: AnimationPlayer = %ShopkeeperAnimation
@onready var blink_timer: Timer = %BlinkTimer
@onready var card_tooltip_popup: CardTooltipPopup = %CardTooltipPopup
@onready var modifier_handler: ModifierHandler = $ModifierHandler
func _ready() -> void:
	for shop_card: ShopCard in cards.get_children():
		shop_card.queue_free()
	for shop_relic: ShopRelic in relics.get_children():
		shop_relic.queue_free()
	Events.shop_card_bought.connect(_on_shop_card_bought)
	Events.shop_relic_bought.connect(_on_shop_relic_bought)
	_blink_timer_setup()
	blink_timer.timeout.connect(_on_blink_timer_timeout)
func _input(event: InputEvent) -> void:
	if event.is_action_pressed("ui_cancel") and card_tooltip_popup.visible:
		card_tooltip_popup.hide_tooltip()
func populate_shop() -> void:
	_generate_shop_cards()
	_generate_shop_relics()
func _blink_timer_setup() -> void:
	blink_timer.wait_time = randf_range(1.0, 5.0)
	blink_timer.start()
func _generate_shop_cards() -> void:
	var shop_card_array: Array[Card] = []
	var available_cards: Array[Card] =
char_stats.draftable_cards.duplicate_cards()
	RNG.array_shuffle(available_cards)
	shop_card_array = available_cards.slice(0, 3)
	for card: Card in shop_card_array:
		var new_shop_card := SHOP_CARD.instantiate() as ShopCard
		cards.add_child(new_shop_card)
		new_shop_card.card = card
	new_shop_card.current_card_ui.tooltip_requested.connect(card_tooltip_pop
up.show_tooltip)
		new_shop_card.gold_cost =
_get_updated_shop_cost(new_shop_card.gold_cost)
		new_shop_card.update(run_stats)
func _generate_shop_relics() -> void:
	var shop_relics_array: Array[Relic] = []
	var available_relics := shop_relics.filter(
		func(relic: Relic):
			var can_appear :=
relic.can_appear_as_reward(char_stats)
			var already_had_it := relic_handler.has_relic(relic.id)
			return can_appear and not already_had_it
	)
	RNG.array_shuffle(available_relics)
	shop_relics_array = available_relics.slice(0, 3)
	for relic: Relic in shop_relics_array:
		var new_shop_relic :=
SHOP_RELIC.instantiate() as ShopRelic
		relics.add_child(new_shop_relic)
		new_shop_relic.relic =
relic
		new_shop_relic.gold_cost =
_get_updated_shop_cost(new_shop_relic.g
old_cost)
		new_shop_relic.update(run_stats)
func _update_items() -> void:
	for shop_card: ShopCard in
cards.get_children():
		shop_card.update(run_stats)
	for shop_relic: ShopRelic in
relics.get_children():
		shop_relic.update(run_stats)
func _update_item_costs() -> void:
	for shop_card: ShopCard in
cards.get_children():
		shop_card.gold_cost =
_get_updated_shop_cost(shop_card.gold_c
ost)
		shop_card.update(run_stats)
	for shop_relic: ShopRelic in
relics.get_children():
		shop_relic.gold_cost =
_get_updated_shop_cost(shop_relic.gold_c
ost)
		shop_relic.update(run_stats)
func _get_updated_shop_cost(original_cost:
int) -> int:
	return
modifier_handler.get_modified_value(origi
nal_cost, Modifier.Type.SHOP_COST)
func _on_back_button_pressed() -> void:
	Events.shop_exited.emit()
func _on_shop_card_bought(card: Card,
gold_cost: int) -> void:
	char_stats.deck.add_card(card)
	run_stats.gold -= gold_cost
	_update_items()
func _on_shop_relic_bought(relic: Relic,
gold_cost: int) -> void:
	relic_handler.add_relic(relic)
	run_stats.gold -= gold_cost
	if relic is CouponsRelic:
		var coupons_relic :=
relic as CouponsRelic
	coupons_relic.add_shop_modifier
(self)
	_update_item_costs()
	else:
		_update_items()
func _on_blink_timer_timeout() -> void:
	shop_keeper_animation.play("blin
k")
	_blink_timer_setup()
```

Вихідний код файлу battle_over_panel.gd

```
class_name BattleOverPanel
extends Panel
const MAIN_MENU = "res://scenes/ui/main_menu.tscn"
enum Type {WIN, LOSE}
@onready var label: Label = %Label
@onready var continue_button: Button = %ContinueButton
@onready var main_menu_button: Button = %MainMenuButton
func _ready() -> void:
	continue_button.pressed.connect(func(): Events.battle_won.emit())
	main_menu_button.pressed.connect(get_tree().change_scene_to_file.bind(MAIN_MENU))
	Events.battle_over_screen_requested.connect(show_screen)
func show_screen(text: String, type: Type) -> void:
	label.text = text
	continue_button.visible = type == Type.WIN
	main_menu_button.visible = type == Type.LOSE
	show()
	get_tree().paused = true
```

Вихідний код файлу battle_ui.gd

```
class_name BattleUI
extends CanvasLayer
@export var char_stats: CharacterStats : set = _set_char_stats
@onready var hand: Hand = $Hand
@onready var mana_ui: ManaUI = $ManaUI
@onready var end_turn_button: Button = %EndTurnButton
@onready var draw_pile_button: CardPileOpener = %DrawPileButton
@onready var discard_pile_button: CardPileOpener = %DiscardPileButton
@onready var draw_pile_view: CardPileView = %DrawPileView
@onready var discard_pile_view: CardPileView = %DiscardPileView
func _ready() -> void:
	Events.player_hand_drawn.connect(_on_player_hand_drawn)
	end_turn_button.pressed.connect(_on_end_turn_button_pressed)
	draw_pile_button.pressed.connect(draw_pile_view.show_current_view.bind("Draw Pile", true))
	discard_pile_button.pressed.connect(discard_pile_view.show_current_view.bind("Discard Pile"))
func initialize_card_pile_ui() -> void:
	draw_pile_button.card_pile = char_stats.draw_pile
	draw_pile_view.card_pile = char_stats.draw_pile
	discard_pile_button.card_pile = char_stats.discard
	discard_pile_view.card_pile = char_stats.discard
func _set_char_stats(value: CharacterStats) -> void:
	char_stats = value
	mana_ui.char_stats = char_stats
	hand.char_stats = char_stats
func _on_player_hand_drawn() -> void:
	end_turn_button.disabled = false
func _on_end_turn_button_pressed() -> void:
	end_turn_button.disabled = true
	Events.player_turn_ended.emit()
```

Вихідний код файлу hand.gd

```
class_name Hand
extends HBoxContainer
const CARD_UI_SCENE := preload("res://scenes/card_ui/card_ui.tscn")
@export var player: Player
@export var char_stats: CharacterStats
func add_card(card: Card) -> void:
	var new_card_ui := CARD_UI_SCENE.instantiate() as CardUI
	add_child(new_card_ui)
	new_card_ui.reparent_requested.connect(_on_card_ui_reparent_requested)
	new_card_ui.card = card
	new_card_ui.parent = self
	new_card_ui.char_stats = char_stats
	new_card_ui.player_modifiers = player.modifier_handler
func discard_card(card: CardUI) -> void:
	card.queue_free()
func enable_hand() -> void:
	for card: CardUI in get_children():
		card.disabled = false
		if card.is_hovered():
	card.card_state_machine.on_mouse_entered()
func disable_hand() -> void:
	for card: CardUI in get_children():
		card.disabled = true
func _on_card_ui_reparent_requested(child: CardUI) -> void:
	child.disabled = true
	child.reparent(self)
	var new_index := clampi(child.original_index, 0, get_child_count())
	move_child.call_deferred(child, new_index)
	child.set_deferred("disabled", false)
```

Вихідний код файлу main_menu.gd

```
extends Control
const CHAR_SELECTOR_SCENE := preload("res://scenes/ui/character_selector.tscn")
const RUN_SCENE = preload("res://scenes/run/run.tscn")
@export var run_startup: RunStartup
@onready var continue_button: Button = %Continue
func _ready() -> void:
	get_tree().paused = false
	continue_button.disabled = SaveGame.load_data() == null
func _on_continue_pressed() -> void:
	run_startup.type = RunStartup.Type.CONTINUED_RUN
	get_tree().change_scene_to_packed(RUN_SCENE)
func _on_new_run_pressed() -> void:
	get_tree().change_scene_to_packed(CHAR_SELECTOR_SCENE)
func _on_exit_pressed() -> void:
	get_tree().quit()
```