Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Розробка 2D-гри на рушії Unity

(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-5
Вадим ПЕРЕПЕЛКО
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Комп'ютерна інженерія
(повна назва освітньої програми)

Керівник: ас. Володимир АРГУНОВ
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп’ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Перепелку Вадиму Віталійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Розробка 2D-гри на рушії Unity

затверджена наказом по університету від “ 26 ” травня 2025 р. № 424

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи Середовище розробки Unity; 2D pixel art графіка; шутер з видом зверху; гра з елементами roguelike; процедурна генерація; мова програмування C#;

4. Перелік питань, що потрібно опрацювати у роботі

1) аналіз питання та огляд існуючих рішень;

2) вибір технології розробки та інструментальних засобів;

3) розробка принципів роботи модулів гри;

4) реалізація ігрових модулів та механік;

5) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій Титульний слайд; Мета роботи; Жанр rogue-like; Аналіз сучасних проектів жанру; Постановка завдання; Процедурна генерація 1; Процедурна генерація 2; Система предметів та інвентар; Зброя та її параметри; Система ворогів; Інтерфейс; Туман війни; Висновки.

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	27.05.25-30.05.25	
2	Вибір технології розробки та інструменталь-	31.05.25-02.06.25	
	них засобів		
3	Розробка алгоритмічного забезпечення	03.06.25-05.06.25	
4	Розробка та відлагодження програмного	06.06.25-09.06.25	
	забезпечення		
5	Оформлення матеріалів кваліфікаційної роботи	10.06.25-11.06.25	
6	Подання кваліфікаційної роботи керівникові та	12.06.25-13.06.25	
	її попередній захист		
7	Подання кваліфікаційної роботи на рецензу-	14.06.25-16.06.25	
	вання		

Дата видачі завдання " 26 " травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ас. Володимир АРГУНОВ
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 128 с., 42 рис., 1 табл., 2 дод., 17 джерел.

КОМП’ЮТЕРНА ГРА, UNITY, ROGUELIKE, 2D ГРАФІКА, PIXEL ART, TOP-DOWN, ПРОЦЕДУРНА ГЕНЕРАЦІЯ РІВНІВ, ГЕЙМДЕВ.

Метою кваліфікаційної роботи є розробка прототипу комп'ютерної відеогри з елементами жанру roguelike, процедурної генерації рівня гри, бойовою системою, ворогами та системою предметів та інвентаря.

У ході виконання кваліфікаційної роботи необхідно проаналізувати розробку комп’ютерних ігор, жанр створюваної гри, схожі продукти, їх відмінності та особливості. Провести аналіз існуючих методів побудови ігрового світу та ігрового процесу. Побудова інтерфейсу та використання рушія Unity та його інструментарію для розробки.

ABSTRACT

Bachelor’s thesis: 128 pages, 42 figures, 1 tables, 2 appendices, 17 sources.

COMPUTER GAME, UNITY, ROGUELIKE, 2D GRAPHICS, PIXEL ART, TOP-DOWN, PROCEDURAL LEVEL GENERATION, GAME DEVELOPMENT.

The aim of the qualification work is to develop a prototype of a computer video game with elements of the roguelike genre, procedural game level generation, combat system, enemies, and a system of items and inventory.

In the course of the qualification work, it is necessary to analyse the development of computer games, the genre of the game being created, similar products, their differences and features. Conduct an analysis of existing methods of building the game world and game process. Build the interface and use the Unity engine and its development tools.

ЗМІСТ

ВСТУП 9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ 11
1.1 Аналіз підходів до створення комп'ютерних ігор 11
1.2 Дослідження жанру roguelike 13
1.2.1 Основні концепції 13
1.2.2 Побудова ігрового світу 15
1.3 Аналіз існуючих ігор жанру 15
1.3.1 The Binding of Isaac 15
1.3.2 Enter the Gungeon 17
1.3.3 Dead Cells 18
1.3.4 Hades 19
1.4 Методи генерації ігрового світу 20
1.4.1 Аналіз існуючих методів генерації 20
1.4.2 Обраний підхід до генерації 22
1.5 Логіка ігрового процесу 24
1.5.1 Керування персонажем 24
1.5.2 Інтерфейс 24
1.5.3 Реалізація бойової системи 27
1.5.4 Вороги та їх поведінка 27
1.6 Постановка завдання 28
2 ДОСЛІДЖЕННЯ ВИКОРИСТАНИХ ТЕХНОЛОГІЙ 30
2.1 Рушій Unity та його компоненти 30
2.2 Мова програмування C# та середовище Visual Studio Code 32
2.4 Графічні редактори 33
3 РОЗРОБКА ПРИНЦИПІВ РОБОТИ МОДУЛІВ ГРИ 34
3.1 Ігровий актор 34
3.1.1 Параметри ігрового актора 34

3.1.2 Система ефектів 35
3.1.3 Переміщення актора 35
3.1.4 Візуальна складова актора 36
3.2 Система зброї 36
3.3 Предмети та інвентар 38
3.4 Логіка кімнат та їх різновид 39
3.5 Інтерфейс 41
3.5.1 Міні-мапа 41
3.5.2 Інвентар 42
3.5.3 Параметри гравця 43
3.6 Процедурна генерація рівню 44
3.6.1 Алгоритм генерації 44
3.6.2 Створення кімнат та з'єднань між ними 45
3.6.3 Створення гілок 45
3.6.4 Створення циклів 46
3.6.5 Фінальна побудова рівню 47
3.7 Вороги 47
3.7.1 Побудова ворогів 47
3.7.2 Система поведінок 48
3.8 Туман війни 49
4 РЕАЛІЗАЦІЯ ІГРОВИХ МОДУЛІВ ТА МЕХАНІК 50
4.1 Кімнати та генерація 50
4.1.1 Кімнати 50
4.1.2 Генерація рівню 53
4.2 Предмети та інвентар 57
4.3 Ігрові актори 60
4.4 Система поведінок 62
4.5 Зброя та снаряди 64
4.6 Інтерфейс 65
4.7 Система туману війни 68

ВИСНОВКИ 70
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 72
ДОДАТОК А Графічний матеріал кваліфікаційної роботи 74
ДОДАТОК Б КОД МОДУЛІВ ГРИ 82
Б.1 Посилання на проєкт: 82
Б.2 Файли та методи 82
Б.2.1 Файл Exit.cs 82
Б.2.2 Метод TryPlaceRoomWithHallway() 83
Б.2.3 Файл DungeonGenerator.cs 84
Б.2.4 Файл LoopGenerator.cs 91
Б.2.5 Файл BranchGenerator.cs 95
Б.2.6 Файл Crystal.cs 97
Б.2.7 Файл Staff.cs 99
Б.2.8 Файл StaffStat.cs 100
Б.2.9 Файл Interactable.cs 101
Б.2.10 Файл ActorDamageController.cs 102
Б.2.11 Файл DamageModel.cs 105
Б.2.12 Файл UIItem.cs 107
Б.2.13 Файл Inventory.cs 110
Б.2.14 Файл GameActor.cs 114
Б.2.15 Файл BehaviorController.cs 115
Б.2.16 Файл BulletManager.cs 118
Б.2.17 Файл Projectile.cs 120
Б.2.18 Файл Map.cs 122
Б.2.19 Файл FogOfWarParticleSystem.cs 124

ВСТУП

Сфера розробки комп'ютерних ігор – одна з най динамічніших, і технологічно просунутих галузей сучасної цифрової індустрії. Геймдев охоплює величезну екосистему, що включає програмування, художній дизайн, музику, створення сценарію, психологію користувача, аналітику, маркетинг, продюсування та багато інших дисциплін. За останні десятиліття відеоігри з нішового захоплення перетворились на глобальний культурний і економічний феномен, здатний конкурувати з кіноіндустрією та телебаченням.

Ігрова індустрія демонструє стійке зростання, щороку приваблюючи мільйони нових гравців по всьому світу. Це стало можливим завдяки розвитку технологій, поширенню доступного інтернету, зростанню мобільного сегменту та вдосконаленню інструментів розробки. Сьогодні гра може бути створена не лише великою студією з сотнями співробітників, а й невеликою командою незалежних розробників або навіть одним ентузіастом. Поява доступних рушіїв, таких як Unity або Unreal Engine, демократизувала процес створення ігор, зробивши його доступним для широкого кола охочих.

Геймдев – це синтез творчості та технології. Створення гри – це не просто написання коду, а побудова цілісного інтерактивного світу з власною логікою, правилами, візуальною стилістикою, наративом та механіками взаємодії. Успішна гра – це результат глибокого розуміння людської поведінки, ігрових патернів, балансування та залучення. Вона має не лише працювати, а й викликати емоції, бути захопливою, мотивувати гравця повертатися знову і знову.

Важливу роль у цій екосистемі відіграє жанрова різноманітність. Кожен жанр по-своєму впливає на процес розробки, визначаючи структуру ігрового процесу, тип взаємодії, потреби в графіці, фізиці, логіці тощо. Особливо цікавою з точки зору геймдизайну та технічної реалізації є

категорія roguelike-ігор – жанру, що поєднує процедурну генерацію, перманентну смерть персонажа, високий рівень складності та високу реіграбельність. Від свого зародження у грі Rogue (1980) жанр пройшов довгий шлях еволюції, породивши безліч піджанрів і механічних інновацій.

Сучасні roguelike-ігри, попри візуальну простоту, вимагають складної архітектури систем: процедурної генерації рівнів, адаптивного керування ворогами, гнучких бойових систем, систем інвентаря та збереження прогресу. Для реалізації таких проєктів використовуються потужні рушії, серед яких Unity є одним із найпоширеніших. Його популярність пояснюється доступністю, великим набором вбудованих інструментів, підтримкою великої спільноти, кросплатформеністю та високим рівнем гнучкості. Разом із мовою програмування C#, Unity дозволяє реалізувати практично будь-який задум, забезпечуючи оптимальний баланс між продуктивністю та зручністю розробки.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Аналіз підходів до створення комп'ютерних ігор

Процес створення комп'ютерних ігор поєднує знання в галузі програмування, графічного дизайну, звукового оформлення, сценарного мистецтва та геймдизайну, тож, залежно від вимог до кінцевого продукту, рівня складності та доступних ресурсів, для покращення процесу розробки та фінального результату обирають різні підходи до розробки гри.

Першим розглянутим підходом буде розробка гри з нуля. Проводиться вона як для звичайного програмного забезпечення, без використання сторонніх комплексних редакторів. Цей підхід передбачає повний контроль над усіма аспектами гри: від рендерингу графіки до управління пам'яттю. Зазвичай для цього використовують мови низького рівня, як C або C++. Завдяки цьому розробники мають доступ до ресурсів комп'ютера з мінімальними накладними витратами. При цьому розробник самостійно реалізовує графіку, фізику, звукову систему, обробку введення, побудову логіки гри тощо.

Такий підхід часто використовується в навчальних цілях або в індустрії для створення високопродуктивних рішень, особливо коли мова йде про ігри для специфічного обладнання або систем (наприклад, вбудовані пристрої, ретро-консолі, унікальні архітектури).[1]

Для більш комплексних проектів можливе використання готових бібліотек для створення ігор. Наприклад використовувати готове рішення для розрахунку фізики, рендерингу, обробки звуку або інтерфейсу. Нехай дуже схоже на минулий метод, цей є досить популярним серед сучасних розробників які хочуть зберегти свободу у розробці але в той самий час зробити її більш гнучкою та простішою[2]. Гарним прикладом однієї з ігор із таким підходом, є «Noita» (рисунок 1.1). Однією з головних її механік є

фізична система, натхненна іграми з падаючим піском (зокрема, грою The Powder Toy); кожен піксель у Noita симулюється. Це означає, що, наприклад, дерево може горіти і зникати, створюючи при цьому пікселі диму, які зрештою можуть задушити гравця. Таку систему було б майже не можливо створити у сучасних повноцінних рушіях та зберегти гарну продуктивність у грі.

Рисунок 1.1 – Скріншот з гри «Noita»

Якщо у розробників немає часу або ресурсів створювати гру та компоненти для її розробки з нуля, існують вже готові комплексні програми для розробки ігор – ігрові рушії. Ігрові рушії (або game engines) – це спеціалізовані програмні комплекси, що надають інструментарій для створення, редагування, налагодження та запуску комп'ютерних ігор. Їхнє основне завдання – спростити та пришвидшити процес розробки, дозволяючи розробникам зосередитись на геймплейних механіках, дизайні рівнів, сценаріях і візуальному оформленні, а не на низькорівневому програмуванні.

Серед найпоширеніших рушіїв, що використовуються в індустрії, можна виділити Unity, Unreal Engine, Godot, GameMaker Studio та Construct. Кожен із них має свої сильні сторони, цільову аудиторію та типові сфери

застосування.

Таблиця 1.1 – Порівняння популярних ігрових рушіїв

Рушій	Мова програмування	Особливості	Ліцензія
Unity	C#	Потужний редактор, підтримка 2D/3D, велика спільнота, мультиплатформеність	Безкоштовний до певного доходу, комерційна
Unreal Engine	C++, Blueprints	Реалістична графіка, високопродуктивний рендер, візуальне програмування	Безкоштовний до $1 млн доходу
Godot Engine	GDScript, C#, C++	Відкритий код, легкий, зручний для 2D, активно розвивається	Повністю безкоштовний (MIT)
GameMaker Studio	GML (власна мова)	Орієнтований на 2D, легкий в освоєнні	Комерційна
Construct	Візуальне програмування	Не потребує коду, інтерактивний інтерфейс	Комерційна

1.2 Дослідження жанру roguelike

1.2.1 Основні концепції

Roguelike-ігри базуються на низці принципів, які стали основоположними для жанру. Історія жанру бере свій початок у 1980 році, коли вийшла гра Rogue, створена Майклом Тойєм, Гленном Вічманом та Кеном Арнольдом. Вона задала основні принципи, які згодом стали основою для цілого жанру: процедурна генерація рівнів (рисунок 1.2), покроковий геймплей, перманентна смерть персонажа (permadeath), високий рівень складності та глибока варіативність проходження. В наступні роки з'явилось безліч наслідувачів – таких як NetHack, ADOM, Angband, які розширювали концепції оригінальної гри, додаючи нові механіки, класи персонажів,

системи інвентарю та складніші структури підземель.

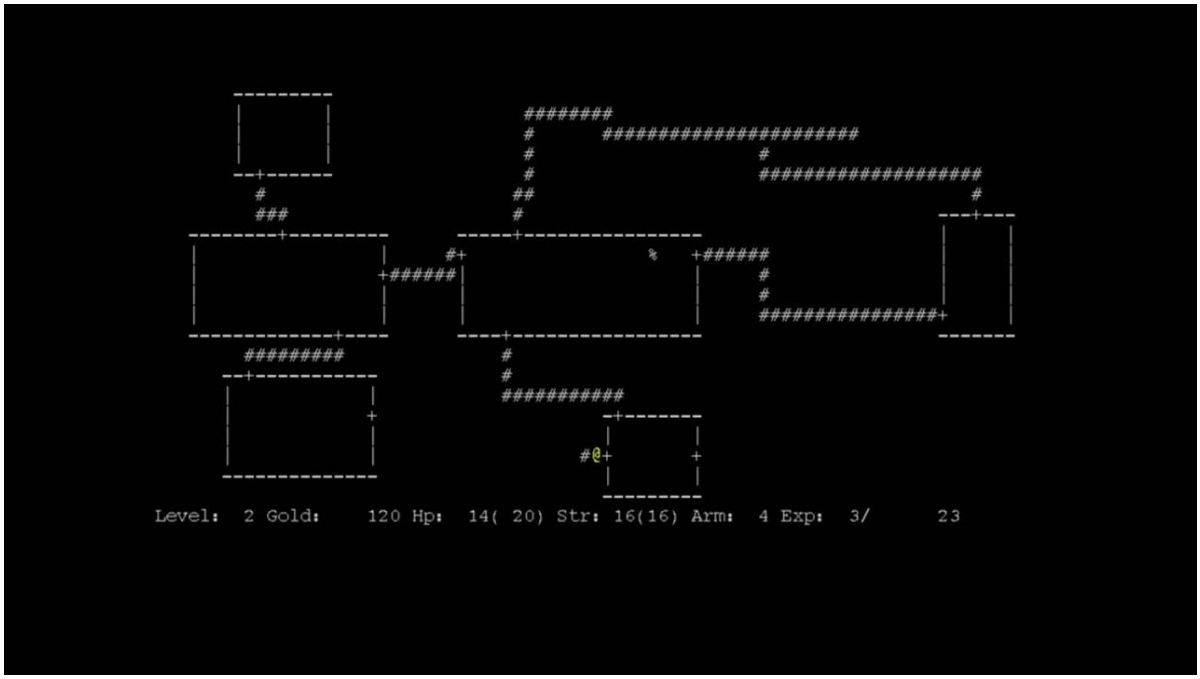

Рисунок 1.2 – Згенерований рівень з гри «Rogue»

У класичних roguelike-іграх після загибелі герой втрачає всі досягнення, і гра починається спочатку. Цей механізм формує сильну напругу під час проходження та змушує гравця приймати обережні й стратегічні рішення. Геймплей roguelike часто глибоко пов'язаний з елементами RPG: прокачування характеристик, збирання екіпірування, вивчення нових умінь, поступове освоєння ігрового світу.

Класичний roguelike відзначається покроковим геймплеєм та високим рівнем складності, але в сучасних реаліях roguelike ігри існують у різних формах, наприклад таких як динамічні шутери або екшн платформери. Ігри в жанрі часто є інтелектуальними викликами для гравця, де важливо планувати ходи, раціонально використовувати ресурси, адаптуватися до обмеженого огляду території або обставин. Основні механіки зазвичай передбачають випадкову генерацію карт, появу ворогів із різними шаблонами поведінки, глибоку систему інвентаря, а також систему розвитку персонажа через досвід чи знайдені артефакти.

Згодом жанр дав початок численним похідним формам.

Найпомітнішою з них стала категорія roguelite – ігор, які запозичили лише частину характеристик класичних roguelike, але спростили інші аспекти для ширшої аудиторії. У roguelite, наприклад, часто зберігаються певні елементи прогресу між проходженнями (мета-прогрес), геймплей може бути реального часу, а рівень складності більш помірний.

1.2.2 Побудова ігрового світу

У класичних roguelike-іграх ігровий світ зазвичай представляє собою підземелля або вежу з великою кількістю кімнат і коридорів, що генеруються динамічно під час кожного проходження. Структура таких світів побудована так, щоб надати гравцеві відчуття дослідження невідомого середовища – на початку гравець бачить лише частину карти, а решта відкривається в процесі руху. Часто реалізується механіка туману війни, де непереглянуті ділянки залишаються прихованими.

Кімнати з яких складається ігровий світ, зазвичай поділені на окремі рівні або поверхи, кожен з яких має свою складність, тип ворогів, пастки, нагороди. Важливе значення має також екологія ворогів: певні істоти можуть мешкати лише в конкретних зонах, а механіки взаємодії між об'єктами світу призводять до різноманітних результатів – наприклад, вибухова бочка, влучивши в ворога, може одночасно зруйнувати стіну та відкрити приховану кімнату.

Завдяки процедурному підходу світ у roguelike не фіксований – він більше схожий на живу структуру, яка щоразу створюється заново забезпечуючи різноманітність ігрового досвіду.

1.3 Аналіз існуючих ігор жанру

1.3.1 The Binding of Isaac

The Binding of Isaac (рисунок 1.3), є однією з найвідоміших і найвпливовіших ігор у жанрі roguelike/roguelite. Гра поєднує в собі елементи аркадного двовимірного екшну з процедурною генерацією рівнів. У центрі геймплею - дослідження підземель, що кожного разу створюються випадково. Основна механіка полягає у дослідженні кімнат, в яких гравець зіштовхується з хвилями ворогів, босами та пастками. Перманентна смерть змушує проходити гру з нуля при кожній поразці, що робить кожну спробу унікальною.

Найбільш комплексною системою геймплею є система предметів, яка дозволяє створювати несподівані комбінації ефектів. Кожен предмет змінює характеристики персонажа, а іноді кардинально змінює стиль гри, додаючи лазери замість сліз, автоматичну стрільбу, політ, отруєння ворогів тощо. У грі реалізовано понад 600 предметів, які взаємодіють між собою, дозволяючи формувати складні стратегічні побудови. Також додані механіки угод з дияволом і ангелом, альтернативні маршрути, секретні кімнати, активні здібності та система випробувань. Весь геймплей базується на динаміці прийняття рішень в умовах випадковості, що забезпечує глибоку реіграбельність і постійне відчуття новизни[3].

Рисунок 1.3 – Скріншот з гри «The Binding of Isaac»

1.3.2 Enter the Gungeon

Enter the Gungeon (рисунок 1.4) концентрується на надзвичайно швидкому темпі бою та майстерності ухиляння. Гравець керує одним із героїв, які потрапляють до процедурно згенерованого підземелля, де основною метою є виживання під шквалом куль у стилі bullet hell.

Ключова механіка гри – це перекати (dodge rolls), які на короткий час роблять персонажа невразливим, дозволяючи ухилятися від атак. Велике значення має вибір зброї: у грі понад 300 одиниць, і кожна з них має унікальну поведінку, включаючи гумористичні та сюрреалістичні ефекти. Деяка зброя стріляє словами, інша – бджолами або навіть музичними нотами.

Окрім стрілянини, важливими є механіки приховування за укриттями, активація тимчасових бонусів і взаємодія з NPC, що дозволяє відкривати нові предмети та полегшувати подальші спроби. У грі присутні секретні кімнати, пастки, альтернативні боси та різні маршрути проходження. Все це формує насичену геймплейну систему, яка базується на навичках гравця, миттєвій

реакції та тактичному мисленні в умовах постійної загрози[4].

Рисунок 1.4 – Скріншот з гри «Enter The Gungeon»

1.3.3 Dead Cells

Dead Cells поєднує механіки roguelite із платформною структурою метроїдванії, формуючи динамічний бойовий досвід із розгалуженою структурою рівнів. У центрі геймплею знаходиться бойова система, що вимагає точного таймінгу ударів, ухилянь та парирувань.

Гравець може обирати різні стилі бою завдяки широкому арсеналу зброї: від мечів і списів до арбалетів, гранат і пасток. Усі зброї мають різні типи атак, швидкість, ефекти та взаємодію з ворогами. Кожна смерть обнуляє прогрес, але дозволяє накопичувати клітини для постійного покращення – відкриття нових предметів, додаткових фляг з лікуванням чи нових гілок розвитку.

Однією з особливостей гри є система прокачки на ходу: гравець обирає, у що вкладати розвиток прямо під час забігу, що додає стратегічної глибини. Додатково гра має розгалужену карту з альтернативними маршрутами, доступ до яких відкривається після здобуття спеціальних рун. Dead Cells

також має елементи мобільності: подвійні стрибки, телепортацію, використання гаків і руйнацію перешкод, що додає вертикальності та різноманіття у проходженні.

Рисунок 1.5 – Скріншот з гри «Dead Cells»

1.3.4 Hades

Hades – це roguelite-гра яка поєднує інтенсивний бойовий процес із глибоким сюжетом, повністю озвученими діалогами та візуально привабливою стилізацією на тему грецької міфології. Гравець проходить череду арен, борючись із ворогами у динамічному бою, що базується на комбінації ударів, стрімких ривків і активації божественних здібностей. Кожен забіг формується випадково: дари від богів Олімпу змінюють геймплей шляхом додавання нових властивостей до базових атак. Наприклад, атака може отримати електричний ефект від Зевса або вибухову хвилю від Посейдона. Дари можна комбінувати, створюючи синергії, що підходять до конкретного стилю бою – ближній, дальній, магічний чи оборонний. Також є можливість модифікувати забіги через систему дзеркала ночі та контрактів, які підвищують складність, але додають більше нагород.

Після кожної смерті персонаж повертається у дім Аїда, де може покращити базу, спілкуватися з персонажами (рисунок 1.5), дізнаватися нові сюжетні подробиці або відкривати нову зброю. Таким чином, гра майстерно інтегрує елементи roguelite з наративною прогресією, де кожен новий забіг не лише змінює геймплей, а й розкриває нові сюжетні нитки.

Рисунок 1.6 – Скріншот сюжетного діалогу з гри «Hades»

1.4 Методи генерації ігрового світу

1.4.1 Аналіз існуючих методів генерації

Процедурна генерація – це метод автоматичного створення ігрового контенту на основі алгоритмів і випадковості. Її особливість полягає у тому, що замість того, щоб створювати кожен рівень вручну, розробники задають правила, за якими гра сама формує ігрове середовище. Перш за все, система визначає ключові елементи, які має включати рівень: стартову позицію, фінальну кімнату або бос-арену, проміжні кімнати, ігрові події, ворогів, скарби. Далі встановлюється послідовність генерації: які частини карти генеруються першими, які – після, і на основі чого приймаються рішення про їхнє розміщення. Наприклад, алгоритм може почати з генерації основного маршруту від початку до кінця, після чого додати гілки, що ведуть до

бонусних кімнат або небезпечних пасток. Якщо гравець має певні обмеження на пересування (наприклад, ключі, двері, телепорти), це також враховується під час генерації. У багатьох іграх важливо, щоб гравець не застряг без можливості продовжити гру – це означає, що логічні перевірки цілісності карти є обов'язковими етапами генерації. У сучасній розробці використовується безліч різних підходів до процедурної генерації, кожен з яких має свої переваги та обмеження. Опишемо найпоширеніші з них.

Алгоритми на основі сітки (Grid-based) – метод передбачає поділ простору на клітинки, у яких розміщуються кімнати чи коридори. Найпростіший варіант – заповнити сітку кімнатами у випадкових позиціях, а потім з'єднати їх коридорами. Часто використовується генерація лабіринтів, таких як DFS (Depth First Search) або Prim's Algorithm, які створюють повністю з'єднані простори без циклів. Вони ідеально підходять для створення підземель.

Алгоритми на графах (Graph-based) – Ігровий світ представляється у вигляді графа, де вузли – це кімнати, а ребра – проходи. Це дозволяє більш гнучко контролювати зв'язки між об'єктами, в тому числі додавати цикли, альтернативні маршрути, секретні шляхи. Зручно використовувати, коли потрібно сформувати логічну структуру, наприклад: старт, кілька проміжних подій , бос. Додаткові ребра додають циклічність і варіативність.

BSP (Binary Space Partitioning) – метод широко використовується для генерації dungeon-рівнів. Він працює за принципом поділу області на дві частини (горизонтально або вертикально), потім кожна з частин знову ділиться, поки не буде досягнуто певного мінімального розміру. Кожна отримана зона перетворюється в кімнату, а сусідні зони з'єднуються коридорами. BSP дозволяє генерувати складні, але логічно зв'язані рівні з контрольованими розмірами кімнат.

Алгоритми на основі шуму (Perlin Noise / Simplex Noise) – підхід використовується для створення більш природних, органічних форм та ландшафтів, поверхонь, розподілу об'єктів. Застосовується переважно в іграх

з відкритими світами або для формування хаотичних патернів. Для roguelike частіше слугує допоміжним інструментом – наприклад, для формування візуального наповнення або варіацій у типах кімнат.

Wave Function Collapse (WFC) – це алгоритм, що працює на основі аналізу шаблонів. WFC намагається скласти нову карту, використовуючи фрагменти існуючих, при цьому дотримуючись правил сумісності між сусідніми елементами. Його використовують у складних візуальних генераціях, де потрібно досягти органічності й стилістичної єдності без втрати варіативності.

1.4.2 Обраний підхід до генерації

У межах реалізації проєкту було обрано змішаний підхід до процедурної генерації ігрового світу, що поєднує елементи графової побудови, а також динамічного розміщення кімнат через виходи. Такий підхід забезпечує гнучкість структури, підтримку логічної зв'язності кімнат і можливість створення циклів та гілок.

Основна логіка побудови карти полягає в поетапному розміщенні кімнат, які з'єднуються між собою через спеціально визначені виходи. Кожна кімната має набір можливих виходів (напрямки північ, південь, захід, схід або довільні позиції), і під час генерації наступна кімната розміщується відповідно до одного з доступних виходів попередньої. Перед розміщенням проводиться перевірка на колізії з уже розміщеними кімнатами або коридорами – якщо конфлікт виявлено, генератор намагається інший вихід або іншу кімнату з пулу.

Цей підхід дозволяє створювати структурно осмислені підземелля, де гравець просувається від початкової кімнати до кінцевої, але може обирати різні шляхи, знаходити додаткові кімнати (торговці, скарби, пастки тощо), або навіть відкривати скорочені маршрути назад через цикли.

Ще однією особливістю обраного підходу є використання динамічних

кімнатних шаблонів. Кімнати не є однорідними – вони мають унікальну геометрію, тобто різні форми, декорації, заготовки ворогів та виходи. Завдяки цьому генератор створює не просто випадковий набір прямокутників, а завчасно заготовлені кімнати.

Для реалізації з'єднання між кімнатами застосовується система коридорів. Якщо під час розміщення наступної кімнати виникає колізія, система намагається вставити проміжний коридор, який дозволяє обійти перепону. У крайньому випадку генератор повертається на попередній етап, щоб обрати інший шлях або інший шаблон кімнати.

Окрему увагу в генерації було приділено створенню циклів – таких з'єднань, які дозволяють гравцеві повертатися в уже відвідані місця, відкривати нові маршрути до знайомих зон. Цикли генеруються на основі аналізу відстані між вже розміщеними кімнатами: коли виявлено, що остання кімната наблизилась до попередніх, генератор намагається створити альтернативне з'єднання, прокладаючи новий коридор.

1.5 Логіка ігрового процесу

1.5.1 Керування персонажем

Керування персонажем – одна з базових складових будь-якої гри, яка визначає, як гравець взаємодіє з ігровим світом. Цей елемент охоплює реалізацію руху, анімації, фізики, взаємодії з об'єктами та реагування на навколишнє середовище. У двовимірних (2D) екшен-іграх часто використовується схема WASD для руху та миші для прицілювання й стрільби, що дозволяє гравцю мати повний контроль над персонажем у реальному часі.

Гравець керує персонажем за допомогою клавіатури (для пересування) та миші (для стрільби або взаємодії). Персонаж може вільно переміщатися між кімнатами, збирати предмети, або вступати в бій з ворогами.

Гравець переміщується між кімнатами, які з'єднані між собою коридорами. Світ побудовано процедурно, що означає, що при кожному проходженні структура рівня відрізняється. Гравець відкриває карту світу поступово – за аналогією з системою туману війни, кімнати на мінікарті з'являються тільки після того, як гравець увійшов у них. Для цього реалізована система виявлення активних секторів карти, яка оновлює дані на мінікарті після входу в нову зону. Технічно, дослідження підтримується системою збереження стану кімнат – після того, як кімната пройдена, вона вважається безпечною і більше не активує ворогів при повторному вході та зберігає статус відкритої для туману. Це реалізовано через менеджер кімнат, який фіксує стан кожної зони гри. Не відкриті місця сховані туманом.

1.5.2 Інтерфейс

Інтерфейс користувача (UI) у відеоіграх виконує роль комунікаційного мосту між гравцем і внутрішніми механіками гри. Саме через інтерфейс

гравець отримує візуальні сигнали про свій стан, доступні ресурси, можливі дії, цілі та прогрес. У більшості roguelike-ігор інтерфейс має декілька постійно активних елементів (рисунок 1.7): індикатор здоров'я, кількість боєприпасів, ресурси (наприклад, золото або енергія), поточна зброя або активне уміння, а також міні-карта. Ці елементи зазвичай фіксуються на певній частині екрану, як-от верхній лівий чи нижній правий кут. Крім того, інтерфейс може містити тимчасові елементи – наприклад, спливаючі підказки, повідомлення про підбір предметів, зміни параметрів, активацію вмінь або отримання шкоди. Такі повідомлення дозволяють гравцю миттєво зреагувати на події без потреби постійно перевіряти інвентар чи мапу.

Рисунок 1.7 – Приклад інтерфейсу індикаторів з гри «Exit The Gungeon»

Окремо варто виділити інтерактивну карту рівня. У roguelike-іграх карта часто заповнюється поступово у процесі дослідження світу. Вона може мати вигляд міні-карти, що оновлюється у реальному часі, або відкриватися повністю на весь екран натисканням певної клавіші. На ній відображаються кімнати, які вже були відвідані, їх зв'язки, двері, закриті проходи, місцезнаходження гравця, ворогів, скринь, телепортів, торгівців або босів. Мапа (рисунок 1.8) – це інструмент навігації, який допомагає орієнтуватися у складній структурі рівня та приймати стратегічні рішення щодо маршруту руху.

Ще одним важливим компонентом є інвентар, що дозволяє переглядати й управляти усіма предметами, які гравець зібрав. У грі, над якою ведеться робота, інвентар містить систему керування зброєю та модифікаторами до неї. Гравець може відкривати інвентар, перетягувати предмети, змінювати їх

розташування, прикріплювати або від'єднувати модулі від зброї.

Рисунок 1.8 – Приклад інтерфейсу мапи з гри «Enter The Gungeon»

Інтерфейс відображає вплив предметів на характеристики, наприклад як змінилася траєкторія пострілу, швидкість, або додався ефект вогню чи замороження. Це дозволяє візуально оцінити результати комбінацій без потреби проводити багато тестів у бою.

Рисунок 1.9 – Приклад інтерфейсу інвентарю з гри «Minecraft»

1.5.3 Реалізація бойової системи

Бойова система побудована навколо активної стрілянини в реальному часі з видом згори, що є типовим для жанру roguelite. Гравець керує прицілом за допомогою миші, а стріляє у вибраному напрямку, використовуючи різні типи зброї. Зброя має унікальні характеристики – швидкість пострілу, відстань польоту, шкоду, тип снарядів – і може бути замінена чи покращена впродовж гри.

Особливістю системи є те, що зброя не є статичною – її властивості динамічно змінюються залежно від предметів, які гравець знаходить і додає до інвентаря. Ці предмети можуть суттєво змінити поведінку зброї: наприклад, змінити траєкторію польоту куль на дугоподібну або зигзагоподібну, зробити снаряди вогняними чи вибуховими, збільшити їхній розмір або ввести ефект пошкодження з часом (DoT). Таким чином, система модифікації дозволяє гравцеві формувати унікальний стиль гри шляхом комбінування ефектів і підбору предметів, які взаємодіють зі зброєю.

Окрім звичайної стрільби, в грі реалізовані активні вміння – спеціальні здібності, які гравець може використовувати після певного часу перезарядки або з витратою ресурсу. Їхня дія може включати масові атаки, тимчасові щити, прискорення, телепортацію тощо. Аналогічно до зброї, ці вміння також змінюють свою поведінку залежно від наявних предметів. Наприклад, активна здатність, яка створює ударну хвилю, може під впливом відповідного предмета накладати ефект спалення або притягувати ворогів.

1.5.4 Вороги та їх поведінка

Кожен ворог функціонує як автономна сутність зі своїм набором станів і поведінкових шаблонів. Базова структура поведінки враховує кілька ключових компонентів: навігацію по кімнаті, вибір цілі (зазвичай це гравець), реакцію на пошкодження та зміну станів (наприклад, коли ворог виявляє

гравця, коли переслідує, атакує або тікає при малій кількості здоров'я). Ця логіка реалізована через state machine (машину станів), де кожен ворог переходить між передбаченими поведінками залежно від умов: відстані до гравця, часу, власного здоров'я, перешкод тощо.

Вороги базуються на спільному абстрактному класі або інтерфейсі, що дозволяє розробнику швидко створювати нові типи, наслідуючи базову логіку та розширюючи її конкретними особливостями. Наприклад, одні вороги поводяться агресивно й намагаються безпосередньо атакувати гравця в ближньому бою, інші – тримають дистанцію та використовують снаряди, а деякі мають підтримувальні функції, як-от лікування союзників або виклик підкріплення. Система пересування ворогів враховує навколишнє середовище: вороги не проходять крізь стіни, обходять перешкоди, а при необхідності – перебудовують маршрут.

Також реалізована система відслідковування взаємодії ворогів із снарядами гравця: кожен ворожий об'єкт має хітбокс і систему обробки ушкоджень, яка враховує тип снаряда, ефекти (вогонь, отрута, уповільнення), а також чинники, пов'язані з обладнанням гравця.

1.6 Постановка завдання

Метою кваліфікаційної роботи, є створення ігрового 2D застосунку у жанрі roguelike побудованим на рушії Unity з використанням мови C#.

Перелік механік що мають бути реалізовані у грі:

- 2D піксель арт графіка з видом зверху;
- система процедурної генерації рівнів та проходів між ними;
- наявність прогресу по рівню, відкриття різних кімнат таких як магазини або кімнати із ворогами;
- керування персонажем, стрільба;
- інтерфейс із параметрами гравця, інвентарем та міні-мапою;
- система зброї та предметів які змінюють властивості зброї та

персонажа;

- система ворогів із різною поведінкою, атакуючими шаблонами та штучним інтелектом;
- система туману війни для приховання не вивчених та недосягаємих частин мапи.

2 ДОСЛІДЖЕННЯ ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

2.1 Рушій Unity та його компоненти

Unity – це потужний кросплатформенний рушій для розробки комп'ютерних ігор, який набув широкої популярності завдяки поєднанню гнучкості, функціональності та зручного інтерфейсу. Його особливість полягає в компонентній архітектурі, яка дозволяє створювати об'єкти гри й наділяти їх функціональністю шляхом додавання до них різних компонентів. Кожен ігровий об'єкт є своєрідною основою, до якої приєднуються елементи: фізика, візуалізація, поведінка через скрипти, анімації, взаємодія зі світлом або звуком.

Редактор Unity (рисунок 2.1) надає середовище, де можна в реальному часі працювати зі сценою, розміщувати об'єкти, налаштовувати світло, камери, шейдери, ефекти частинок і багато іншого. Усі зміни можна миттєво перевірити, активувавши режим гри без виходу з редактора. Така інтеграція дозволяє швидко тестувати геймплейні рішення та коригувати логіку без затримок на компіляцію чи повторне збирання проєкту.

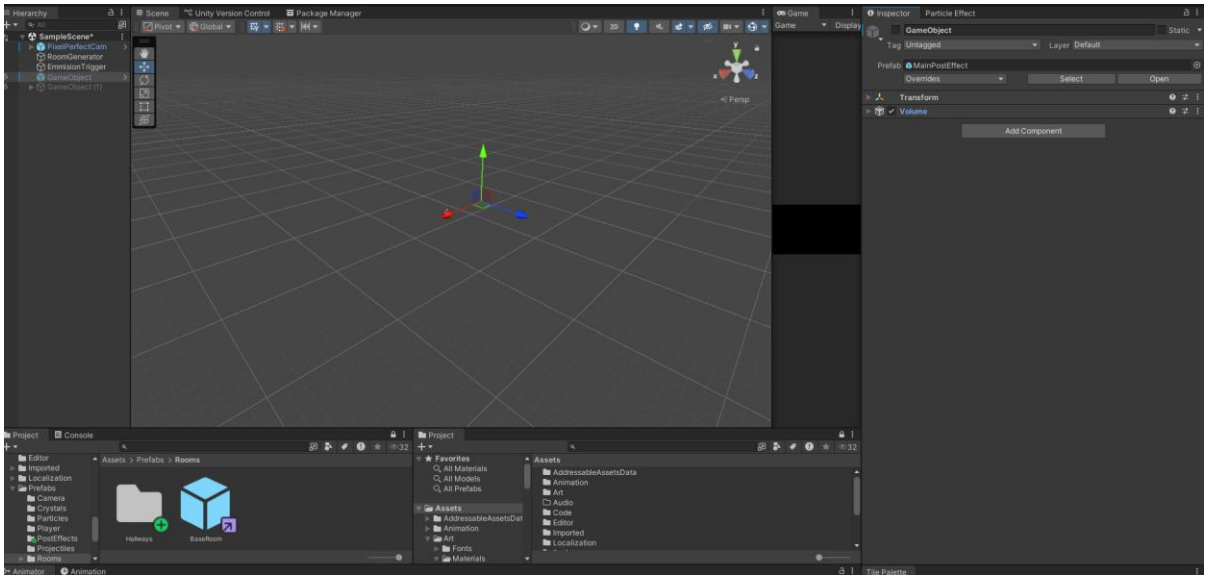

Рисунок 2.1 – Вікно редактора рушія Unity

Інструментарій Unity охоплює практично всі потреби сучасного розробника. Система побудови сцен дозволяє працювати з 3D та 2D простором, підтримує накладання плиток, сітку, фізику зіткнень, а також анімації з використанням контролерів станів. Можна легко організовувати логіку через C# скрипти, які обробляють взаємодії об'єктів, введення з клавіатури чи геймпада, а також реакцію на події у світі гри. Візуальне оформлення інтерфейсу реалізується за допомогою системи Canvas, яка надає гнучкі інструменти для створення меню, інвентарів, інформаційних панелей та інших елементів UI.

Unity також підтримує роботу з фізичними об'єктами у двох режимах – 2D і 3D, де за фізику відповідають вбудовані рушії, адаптовані до типу сцени. Це дає змогу реалізовувати складні системи руху, взаємодії, гравітації, відштовхування, вибухів та інші подібні ефекти. Рушій дозволяє задавати матеріали, які впливають на ковзання чи відскок об'єктів, використовувати тригери для виявлення подій, створювати зони з іншим типом гравітації або середовища.

Ще однією з ключових переваг Unity є його модульна структура. Розробник може підключати лише ті пакети, які необхідні для конкретного проєкту, таким чином зменшуючи навантаження на редактор і прискорюючи розробку. Бібліотека Asset Store відкриває доступ до тисяч плагінів, моделей, текстур, звуків, а також до складних систем штучного інтелекту, діалогових вікон, процедурної генерації або навіть повноцінних шаблонів ігор, які можна адаптувати під власні потреби.

Одним з важливих рішень у контексті проєкту стала підтримка 2D функціоналу та Tilemap системи, яка забезпечує зручне створення рівнів з плиток. Завдяки цьому стало можливим реалізувати систему процедурної генерації кімнат і коридорів, працювати з мапами на рівні логіки, уникати колізій і створювати геймплейні ситуації на основі випадковості. Робота з анімаціями, яка в Unity здійснюється через Animator Controller, дозволила

легко налаштувати поведінку ворогів, змінювати їх стани та створювати живі візуальні ефекти.

Важливу роль також відіграє система керування введенням. Unity підтримує як стару, так і нову Input System, яка дозволяє абстрагуватися від конкретних пристроїв і працювати з абстрактними подіями: натискання кнопки, рух миші, сенсорне торкання. Це дало можливість налаштувати управління персонажем, змінювати поведінку залежно від контексту гри та адаптувати керування під потреби roguelike-жанру.

Обираючи Unity як основний рушій для реалізації проєкту, було враховано як зручність розробки, так і наявність необхідних інструментів для створення гри з процедурною генерацією, бойовою системою, гнучким інтерфейсом, кастомною логікою предметів та складною поведінкою ворогів.

2.2 Мова програмування C# та середовище Visual Studio Code

C# використовується в Unity як основна мова програмування і була логічним вибором для написання всіх скриптів у проєкті. Вона поєднує високу продуктивність із зручним синтаксисом і дозволяє створювати масштабовану, структуровану логіку гри.

У проєкті C# відповідає за всі елементи геймплею: рух персонажа, обробку зіткнень, генерацію кімнат, поведінку ворогів, зміну характеристик зброї, систему умінь, управління інтерфейсом та інвентарем. Завдяки об’єктно-орієнтованому підходу було легко структурувати код, розділити його на модулі та забезпечити повторне використання окремих компонентів. Застосування подій, делегатів і систем зіставлення станів дозволило реалізувати взаємодію між об'єктами без жорсткого зв’язування компонентів.

Завдяки використанню делегатів, подій, абстракцій та інтерфейсів, вдалося реалізувати гнучкі системи, які легко адаптуються під нові механіки. Наприклад, модифікатори зброї реалізовані як окремі компоненти, що підключаються до основної зброї й змінюють її поведінку. Такий підхід

дозволяє швидко експериментувати з новими видами атак або ефектів, не змінюючи базовий код[5].

Для написання коду було обрано Visual Studio Code – сучасне, легке середовище розробки, яке надає всі необхідні можливості для роботи з C# та Unity. Воно дозволяє зручно організовувати проєкт, швидко переходити між файлами, використовувати автодоповнення, знаходити помилки компіляції та працювати з контролем версій[6].

Середовище легко інтегрується з Unity, забезпечуючи комфортну розробку без потреби вручну оновлювати налаштування. Плагіни для роботи з C#, форматування коду, підсвічування синтаксису та автозапуск перевірки помилок допомогли підтримувати проєкт у впорядкованому вигляді.

2.4 Графічні редактори

Для додаткової обробки графіки використовувалися Photoshop та Piskel для роботи із піксель артом. Ці редактори дозволяють працювати з шарами, прозорістю, масками та палітрами кольорів, що необхідно при створенні UI-елементів, ефектів, іконок предметів та текстур.

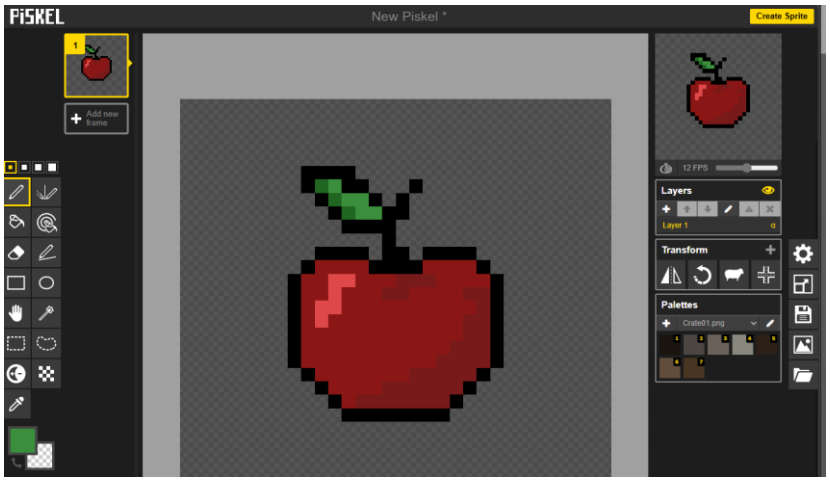

Рисунок 2.2 – Вікно pixel art редактора Piskel

3 РОЗРОБКА ПРИНЦИПІВ РОБОТИ МОДУЛІВ ГРИ

3.1 Ігровий актор

3.1.1 Параметри ігрового актора

Для керування гравцем та системою ігрових персонажів буде створено ігрового актора. Ігровий актор виконує роль контейнера для поведінки, параметрів, анімацій, обробки ушкоджень, навігації та візуалізації.

Ігровий актор – це персонажі що існують у грі, переміщуються по світу, атакують, мають різні властивості та параметри, і керуються як гравцем так і штучним інтелектом. Завдяки модульному підходу, кожен актор легко налаштовується під конкретну роль, а ключові системи взаємодіють між собою через визначені зв'язки (рисунок 3.1).

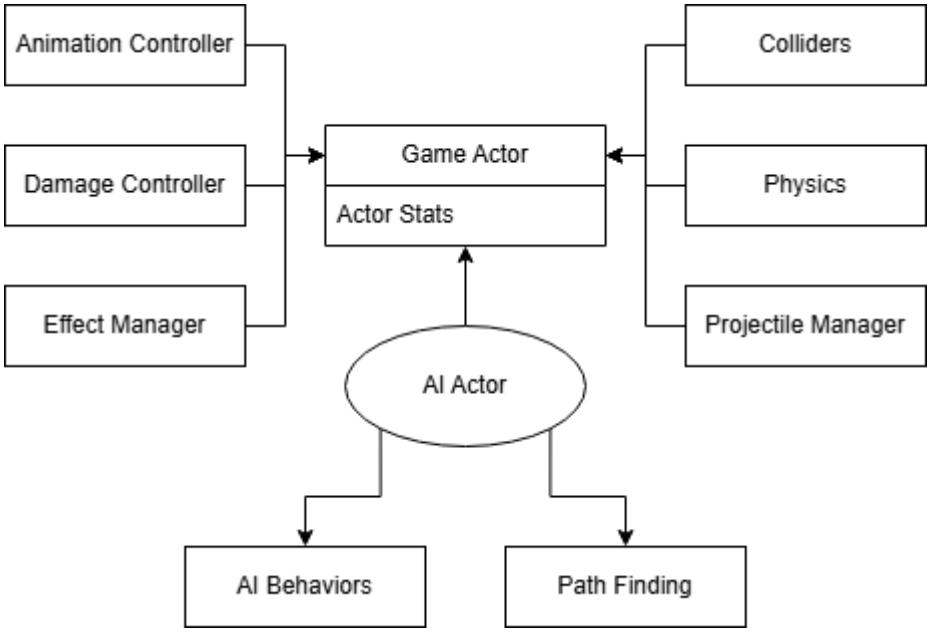

Рисунок 3.1 – Схема побудови ігрового актора

Кожен актор містить набір статистик, що визначають його базові можливості -швидкість та опір до певних типів шкоди (вогонь, електрика

тощо). Вони можуть змінюватись динамічно під час гри – наприклад, через ефекти, бонуси або ушкодження. Ці характеристики можуть використовувати інші модулі і на їх основі виконувати певні дії із певними параметрами, наприклад фізичний модуль змінює швидкість тіла на основі значень у статистиці актора.

3.1.2 Система ефектів

Актор має систему, яка відповідає за додаткову поведінку при отриманні специфічних типів шкоди. Коли актор отримує шкоду, скрипт обробки шкоди (Damage Controller) аналізує тип шкоди та перевіряє відповідні коефіцієнти сприйняття з ActorStats. Якщо сприйняття не дорівнює нулю, активується відповідний ефект. Ефекти мають свій скрипт виконання який керує тим що відбувається коли актор знаходиться під певним ефектом.

Ефекти, що можуть бути накладені на актора:

- вогонь з певною періодичністю наносить певну кількість шкоди актору;
- магія зменшує швидкість снарядів запущений цим актором;
- електрика з певною періодичністю заморожує актора на місці і робить його не активним;
- отрута з певною періодичністю наносить невелику кількість шкоди актору, але може розповсюдитись на сусідніх акторів;
- вода зменшує швидкість цього актора;
- кров робить так, що деякі снаряди випущені цим актором можуть наносити шкоду його союзникам.

3.1.3 Переміщення актора

Для переміщення акторів використовується дві системи, і змінюються в

залежності від того чи це ігровий або не ігровий персонаж.

Для актора яким керує гравець, використовується модуль 2D фізики вбудований у рушій і в залежності від вводу гравця, тіло до якого прикріплений актор буде рухатися у заданому напрямку. Для неігрових персонажів використаний модуль NavMeshAgent який, за принципом пошуку шляхів на площині, буде переміщувати актора до заданої кінцевої точки, оминаючи перешкоди на шляху. Площина для переміщення буде створюватися на основі ігрового світу побудованого процедурним генератором.

3.1.4 Візуальна складова актора

Актор також відповідає за візуальну репрезентацію: анімації та спрайти. На основі кута обертання актора визначається, в який бік він дивиться, і запускається відповідна анімація (вгору, вниз, вліво, вправо). Кут обертання може залежати від того в який бік він рухається, або в якій стороні знаходиться його ціль для атаки. Якщо актор рухається, автоматично вмикається анімація бігу. Також реалізовано дзеркальне відображення спрайта для коректного візуального вигляду під час зміни напрямку.

3.2 Система зброї

Система зброї в грі побудована на взаємодії трьох основних компонентів: самої зброї , снарядів, які вона випускає, та допоміжних елементів – таких як кристали, що можуть змінювати властивості як зброї, так і снарядів. Вся система розроблена як модульна й розширювана, що дозволяє гравцю отримувати нові тактичні можливості шляхом комбінування елементів.

Коли гравець активує зброю, вона ініціює процес атаки, який починається з перевірки своїх активних параметрів – таких як напрямок

стрільби, швидкість, тип шкоди. Ці параметри формуються на основі внутрішніх налаштувань зброї, а також впливу встановлених у неї кристалів. Кристали виступають як модифікатори, які можуть змінювати поведінку зброї (наприклад, додати шкоду вогнем або дозволити випускати декілька снарядів одночасно). Кожен кристал має як пасивні ефекти, що постійно змінюють параметри зброї, так і активні, які можуть бути запущені вручну.

Після формування всіх параметрів відбувається запуск снаряду. Снаряд – це окремий об'єкт у світі гри, який має власний життєвий цикл, починаючи з моменту створення і до моменту знищення. Він рухається в заданому напрямку з визначеною швидкістю. Рух снаряда також може бути змінений активними ефектами від кристалів чи типу зброї (рисунок 3.2).

У момент зіткнення снаряду з ціллю (ворогом або гравцем) відбувається перевірка, чи здатен цей снаряд завдати шкоди цій цілі. Якщо так, система визначає тип завданої шкоди (наприклад, вогонь, електрика) і передає цю інформацію в систему здоров'я цілі. Система здоров'я враховує чутливість до кожного типу шкоди та активує відповідні ефекти. Наприклад, вогняна шкода може запустити ефект горіння, який завдає шкоди з часом, а електрична – зменшити швидкість пересування.

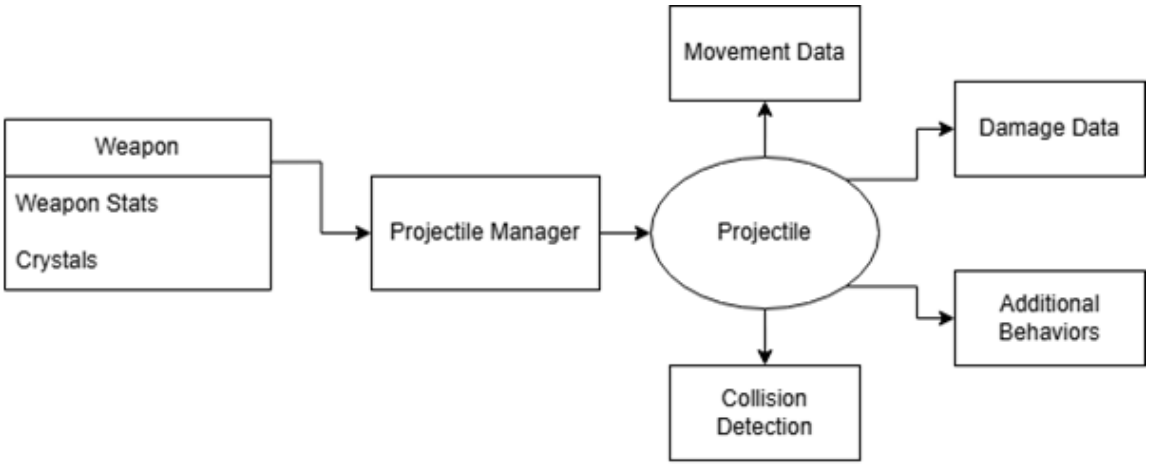

Рисунок 3.2 – Схема роботи логіки снарядів

Ця система дозволяє створити складну бойову взаємодію, де результат залежить не лише від базових характеристик зброї, а й від її налаштування

через кристали, поточної ситуації в бою та типу ворогів. Вона легко масштабується, оскільки підтримує додавання нових типів снарядів, ефектів шкоди або кристалів без потреби змінювати базову структуру.

3.3 Предмети та інвентар

Інвентар у грі виконує роль посередника між гравцем та ігровим світом, дозволяючи збирати, зберігати й взаємодіяти з предметами. Його основна мета – надати гравцю зручний і гнучкий спосіб керувати знайденими речами, такими як кристали чи магічна зброя, без зайвих складнощів.

Система предметів побудована навколо кількох основних типів об'єктів. Найважливіші з них – це кристали та посохи (зброя). Коли гравець підходить до предмета у світі, він може взаємодіяти з ним, після чого той зникає зі сцени та переміщується до інвентаря. Там кожен тип предмета автоматично потрапляє до свого відповідного місця: посох – у слот зброї, кристал – до сітки кристалів. Якщо місця недостатньо, система динамічно розширює інвентар, додаючи нові рядки комірок (рисунок 3.3).

Важливою частиною системи є взаємодія між кристалами та зброєю. Через інтерфейс інвентаря гравець може кріпити кристали до посоха, щоб посилити його властивості або надати йому нові ефекти (наприклад, змінити тип шкоди або покращити характеристики). Кристали встановлюються в спеціальні слоти на зброї, і кожен з них може мати унікальний вплив. У будь-який момент гравець може зняти кристал зі зброї, щоб замінити його іншим або повернути до сітки зберігання. Таким чином, інвентар виступає не лише як місце для зберігання, а і як механізм кастомізації зброї, що напряму впливає на стиль гри.

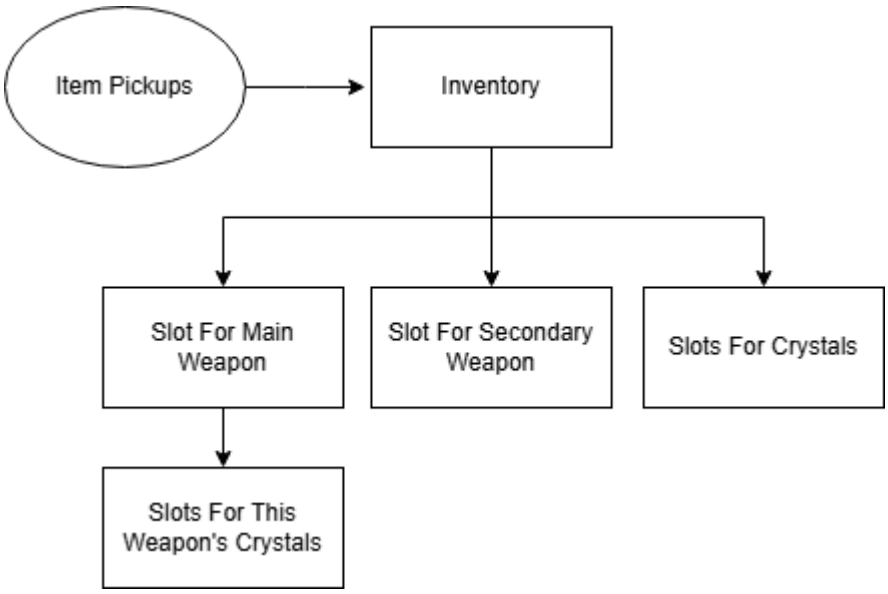

Рисунок 3.3 – Схема побудови інвентарю гравця

Щоб полегшити розуміння предметів, система використовує тултіпи – спливаючі підказки, які з'являються при наведенні курсору на об'єкт у UI. У тултіпі відображається назва предмета, його тип, і за потреби – додаткова інформація про властивості чи ефекти. Це дозволяє гравцю швидко орієнтуватися у вмісті інвентаря, не відкриваючи окремих вікон опису. Важливо, що тултіп працює інтерактивно: зникає при відведенні курсору і автоматично оновлюється при зміні предмета в комірці.

Інвентар також має просту систему відкриття та закриття з анімацією, що робить його частиною загального візуального досвіду гри. Його не потрібно постійно тримати відкритим – він доступний лише тоді, коли гравцю це потрібно, не порушуючи основного ігрового процесу.

У підсумку, ця система створює не просто сховище для речей, а інструмент, що інтегровано у бойову та ролеву частину гри: дозволяє збирати ресурси, модифікувати зброю, експериментувати з властивостями і швидко орієнтуватися у власному спорядженні.

3.4 Логіка кімнат та їх різновид

Кімнати – це основні модулі, з яких складається рівень. Вони

визначають простір, у якому відбувається ігрова активність. Рівень – це набір кімнат, з'єднаних між собою через коридори. Кожна кімната виконує певну функцію в межах проходження: початкова слугує стартовою точкою, бойові кімнати випробовують гравця, магазин дозволяє купувати різні предмети, а кімната з босом – це кульмінація всього шляху. Хоча їхня структура подібна, логіка поведінки відрізняється залежно від типу (рисунок 3.4).

Початкова кімната відкрита з самого початку. У ній нема ворогів, а її призначення – дати гравцеві стартову позицію та перший орієнтир. Магазин теж є безпечним простором, в якому розміщено предмети, з якими можна взаємодіяти. На противагу їм, звичайні бойові кімнати реагують на присутність гравця – коли він входить, система виявлення спрацьовує, і кімната закривається. Це означає, що з'являються вороги, блокуються виходи, і покинути її можна лише після повного очищення. Кімната з босом працює за тим самим принципом, але в ній зосереджено все на одному сильному супротивникові, перемога над яким відкриває шлях до наступного рівня чи нагороди.

Щоб зрозуміти, чи гравець зайшов у кімнату, в кожну кімнату вбудовано спеціальну зону виявлення – невидимий тригер. Він слідкує за об'єктами, що входять і виходять. Коли гравець потрапляє всередину, активується логіка входу: розкривається кімната на мапі, зникає туман війни, починається бій (якщо це бойова кімната). Виходи, які ведуть до інших приміщень або коридорів, теж можуть стати частково видимими, дозволяючи гравцю зорієнтуватися у просторі.

Туман війни – це механізм приховування ще невідкритих зон. До того, як гравець увійде в кімнату, вона затемнена, і її не видно ні на мапі, ні у світі. Як тільки входить гравець, туман для цієї кімнати прибирається – і вона стає частиною відкритого світу. Цей ефект поширюється також на деякі прилеглі зони, наприклад на коридори, які ведуть до інших кімнат. Усе це робить дослідження рівня поступовим – нові простори відкриваються тільки після фізичного наближення.

Коли кімната відкривається, вона додається до мінімапи. Це спрощена репрезентація рівня, створена на основі фактичної геометрії кімнати. Спеціальний алгоритм сканує підлогу й генерує текстуру, яка точно відповідає формі кімнати. Таким чином, на мінімапі буде зображення, збудоване за даними форм кімнати.

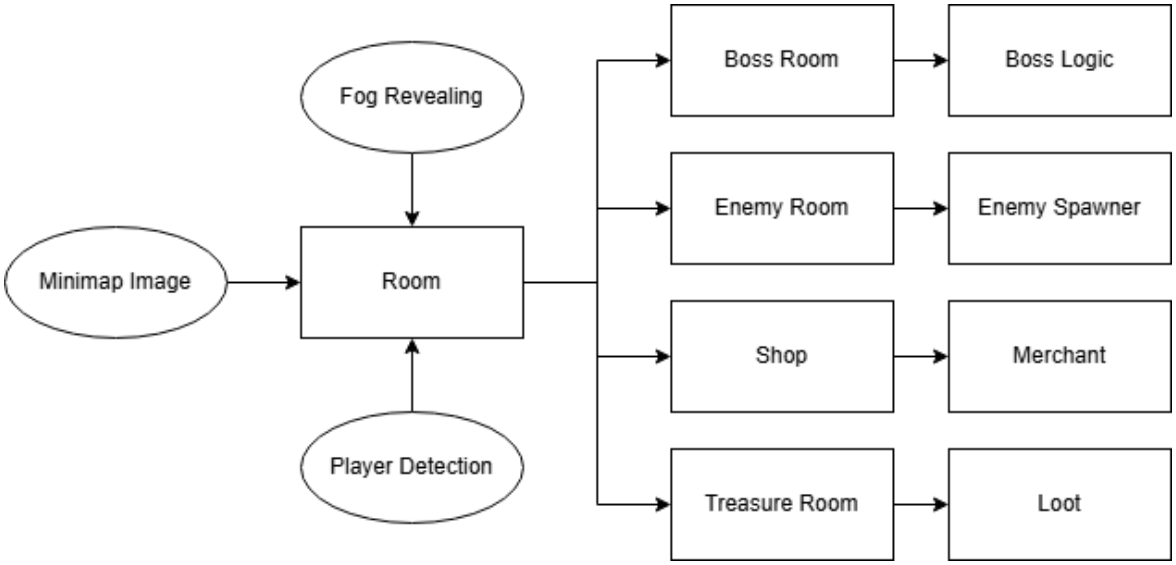

Рисунок 3.4 – Схема побудови модулів кімнат

На концептуальному рівні система кімнат працює як послідовна мережева структура: гравець рухається від кімнати до кімнати, взаємодіє з їх вмістом, відкриває нові ділянки карти й просувається ближче до фінального виклику. Кожна кімната зберігає інформацію про свої виходи, зв'язки з іншими кімнатами, тип і стан – відкрита вона, завершена, активна чи ще прихована. Усе це дозволяє реалізувати динамічний, нелінійний, але контрольований процес дослідження.

3.5 Інтерфейс

3.5.1 Міні-мапа

Міні-мапа – це постійно активна навігаційна система, яка в реальному часі показує положення гравця та відкритих ним кімнат. Спочатку вся мапа

прихована і заповнена туманом війни. Коли гравець заходить у нову кімнату, вона відкривається – спочатку візуально, а потім також стає видимою на мапі. Разом із кімнатою на мапі з'являються й коридори, які до неї ведуть, іконки важливих точок (наприклад, магазинів, босів, особливих кімнат).

На мапі також відображається сама іконка гравця, яка рухається у реальному часі. Завдяки цьому можна орієнтуватися навіть під час переміщення між кімнатами. Для зручності гравець може натиснути певну клавішу і збільшити міні-мапу, відкривши її у повноекранному режимі. У цьому вигляді вона не лише показує всю структуру рівня, а й дозволяє переглянути, які кімнати ще не були досліджені або де залишились незачищені області (рисунок 3.5).

Положення, форма та з'єднання кімнат ґрунтуються на реальній генерації рівня, тобто мапа не створюється окремо – вона витягує дані напряму з генератора, відображаючи лише ті частини, які вже відкрив гравець.

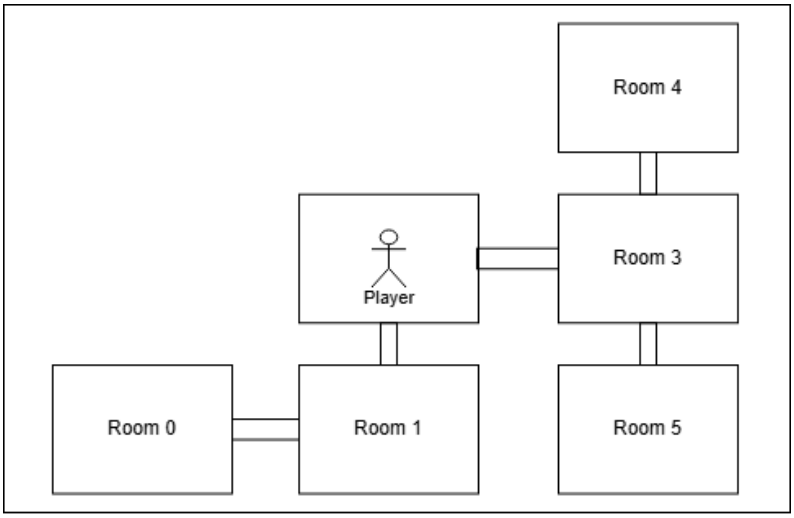

Рисунок 3.5 – Схема вигляду вікна міні-мапи

3.5.2 Інвентар

Гравець може перетягувати предмети між слотами. Якщо предмет сумісний із цільовим слотом, він буде поміщений туди. Інакше – повернеться на початкове місце або взагалі не дозволить переміщення. Інтерфейс

розділений на два блоки: місце зберігання неактивних предметів, та керування основною зброєю (рисунок 3.6).

Окрім того, предмети мають вбудовану підтримку тултіпів – тобто інформаційних вікон, які з'являються при наведенні курсору. Тултіп динамічно підтягує дані про назву, опис, рідкість, ефекти і навіть взаємодію з іншими предметами, якщо такі є. Завдяки цьому гравець може миттєво оцінити властивості предмета без відкриття окремого вікна опису. Якщо предмет активний (наприклад, зілля або модифікатор), гравець може використати його прямо з інвентаря. Якщо це кристал, його можна вставити у зброю шляхом перетягування у відповідний роз'єм, або ж витягнути назад.

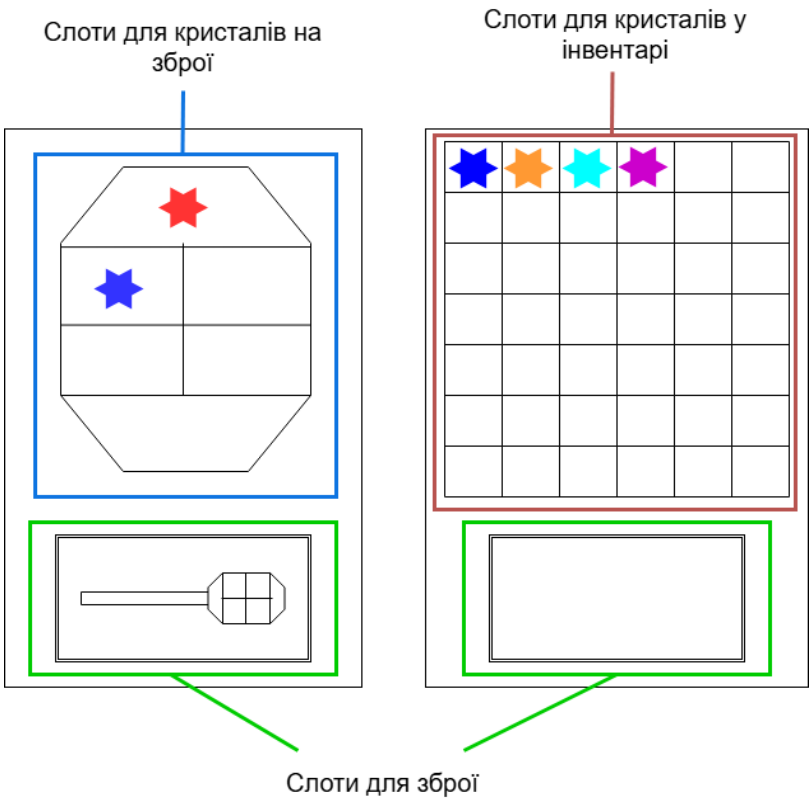

Рисунок 3.6 – Схема вигляду інвентаря гравця на екрані

3.5.3 Параметри гравця

Стан гравця постійно відображається на екрані у спрощеній формі. Єдине, що постійно змінюється в динаміці бою – це здоров'я, тому воно візуалізується чітко й інтуїтивно. При цьому існує розділення між основним

здоров'ям і тимчасовим, яке може зникнути після одного удару. Інша важлива характеристика – гроші. Вони відображаються як постійний ресурс, який накопичується з часом і може витрачатись у ключових точках, наприклад, у магазині (рисунок 3.7).

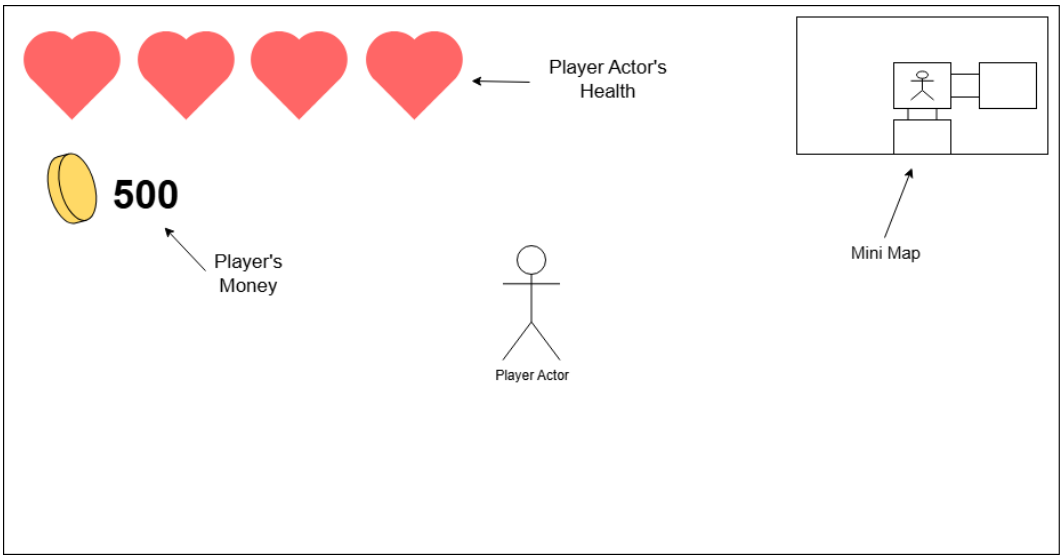

Рисунок 3.7 – Схема інтерфейсу параметрів гравця

3.6 Процедурна генерація рівню

3.6.1 Алгоритм генерації

Для створення ігрового світу було вирішено використовувати процедурну генерацію. Завчасно створені кімнати будуть автоматично виставлятися в ігровому світі і з’єднуватися між собою. Це дозволить створити різноманітний геймплей, в якому кожне проходження буде відрізнятися від попереднього. Для цього необхідно розробити надійну систему в якій можна виставити кімнату, перевірити чи вона не накладається на інші об’єкти які вже були виставлені, і за необхідності переміщена в нове місце.

Але просто виставити кімнати аби як і з’єднати їх прямими проходами не достатньо. Тоді проходження буде занадто прямолінійним, та передбачуваним. Саме тому система була розділена на декілька видів під-

генерації: гілки та цикли. Спочатку створюються гілки та цикли окремо, а потім об’єднуються разом у велику систему. Після остаточної генерації кімнати активуються, гравець з’являється у початковій кімнаті і гра починається.

3.6.2 Створення кімнат та з'єднань між ними

Кожна кімната має завчасно обрані місця для з’єднання з іншими кімнатами – виходи. Виходи можуть мати різне місце-положення та направлення. Коли кімната виставляється, система має прирівняти обрані протилежні виходи двох кімнат та з’єднати їх коридором. Якщо це неможливо зробити бо кімнати або коридор накладаються на вже виставлені об’єкти, має обратися інша пара виходів поки не буде знайдено робочий варіант. Для перевірки накладання кімнат, буде використовуватися система тригерів у Unity.

3.6.3 Створення гілок

Концепт створення гілок працює так, що кімнати виставляються таким чином щоб не перетинатись між собою, не закручуватись, та вести у відносно одному напрямку роблячи не великі відхилення. Наприклад якщо напрямок генерації гілки вправо, то кімнати можуть виставлятися тільки вправо, наверх та вниз. Таким чином створюється група кімнат яку можна буде легко приєднати до інших систем, з’єднуючи першу кімнато з визначеною іншою кімнатою з іншої системи (рисунок 3.8).

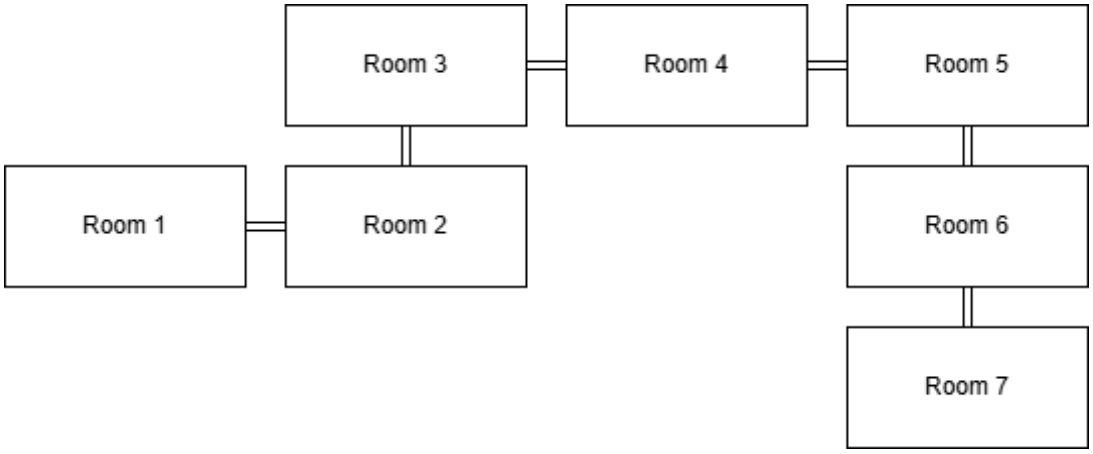

Рисунок 3.8 – Схема вигляду гілки що складається з семи кімнат

3.6.4 Створення циклів

Для створення циклів було розроблено наступний алгоритм. Спочатку кімнати виставляються в ряд враховуючи їх розміри та з’єднуються коридорами. Після цього береться остання виставлена кімната та переноситься до відкритого виходу першої, початкової кімнати. Потім береться наступна з кінця кімната і так само переноситься до тієї кімнати яка була перенесена до цього. Це робиться до тих пір поки відстань від перенесеної кімнати до початкової не буде більшої ніж від наступної у черзі кімнати до початкової. Після виставлення останньої кімнати, дві крайні кімнати з’єднуються особливим непрямим коридором щоб замкнути створений цикл (рисунок 3.9). Така система дозволяє створювати замкнуті системи з кімнат, не зважаючи на їх розміри та різну форму.

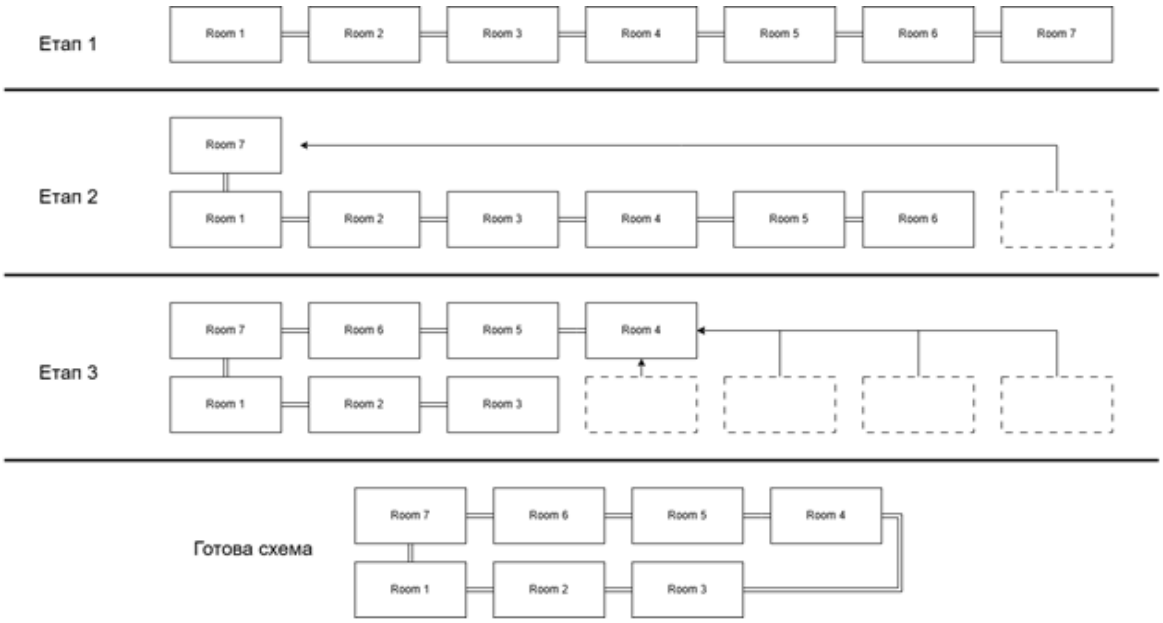

Рисунок 3.9 – Схема алгоритму створення циклів з кімнат

3.6.5 Фінальна побудова рівню

Після створення усіх гілок та циклів окремо, починається їх з’єднання в єдину об’єднану систему. Один за одним групи кімнат виставляються у світі з’єднуючись між собою коридорами та проводячи перевірки на перетин з іншими кімнатами. Так само як і підчас з’єднання кімнат, ідеться пошук підходящого виходу з кімнати до якої приєднується система. Після виставлення останньої групи кімнат проводяться останні підготовки перед початком гри та гравець отримує керування над своїм персонажем.

3.7 Вороги

3.7.1 Побудова ворогів

Вороги у грі – це актори які керуються комп’ютером та створюють виклики для гравця. Кожен ворог з’являється у відведеному для нього місці у світі, та атакує гравця коли той знаходиться у його кімнаті. Для того щоб

комп’ютер мав змогу керувати актором ворога було додано додаткові модулі для взаємодій з цим актором. Модуль системи поведінок дозволяє створювати та додавати для ворогів унікальні або загальні поведінки такі як атака гравця, або пересування у випадковому напрямку. Модуль NavMeshAgent дозволяє надавати актору ворога цільову точку у світі до якої він буде рухатись оминаючи перешкоди (рисунок 3.10).

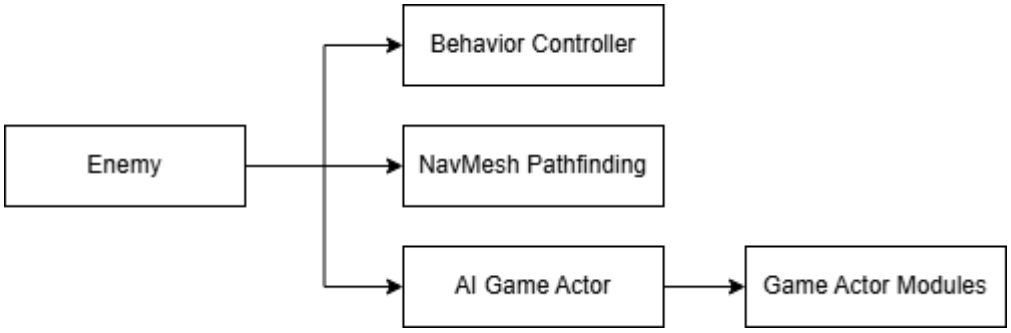

Рисунок 3.10 – Схема модулів ворогів

3.7.2 Система поведінок

Система поведінок дозволяє створювати скрипти для виконання ворогом, які мають певні умови активації та пріоритет. Наприклад для проведення далекобійної атаки актор ворога має знаходитись на відстані 10 одиниць від актора гравця, та не мати перешкод між ними. Система сортує усі доступні для виконання поведінки за пріоритетами та обирає найбільш пріоритетний варіант. У разі збігу пріоритетів обирається одна випадкова поведінка із доступних варіантів. Також поведінки можуть мати затримку у вигляді часу який має пройти між останнім часом ця поведінка була виконана та тим коли можна буде її ще раз використати. Це дозволить зробити систему менш хаотичною та дозволить гравцю перевести дух між атаками або не дасть ворогу безкінечно виконувати одну і ту саму поведінку без перестану (рисунок 3.11).

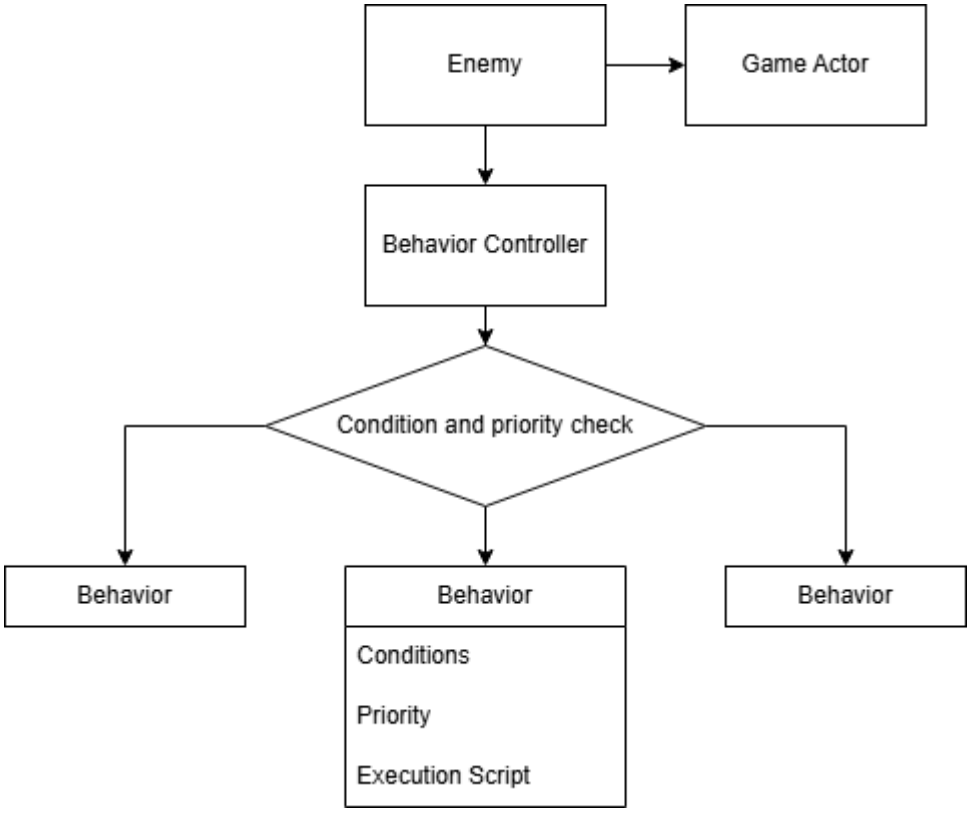

Рисунок 3.11 – Схема роботи контролера поведінок для ворогів

3.8 Туман війни

Туман війни буде використовуватися для приховання не відкритих та недоступних частин світу, наприклад кімнати в яких ще не бував гравець. Туман складається з великої кількості хмаринок які своїм тілом прикривають частини екрану гравця. Для створення ефекту глибини хмари можуть мати різні відтінки та бути не статичними.

Для оптимізації хмари мають бути видимими та розраховувати свою позицію лише у межах видимої області камери гравця. Для цього хмари можуть переміщуватися в залежності від позиції камери створюючи ефект безкінечного туману, хоча він існує лише у невеликих межах навколо камери. Наприклад якщо камера рухається вліво, тоді хмари які були за межами правої частини камери, переміщуються щоб заповнити простір зліва куди рухається камера.

4 РЕАЛІЗАЦІЯ ІГРОВИХ МОДУЛІВ ТА МЕХАНІК

4.1 Кімнати та генерація

4.1.1 Кімнати

У грі кожна кімната реалізована як окремий префаб[8] (заготовка), який містить усі необхідні компоненти для генерації, взаємодії з гравцем і логіки гри (рисунок 4.1). Основою кімнати є Tilemap, який відповідає за візуальне представлення підлоги, стін та інших елементів. Разом із компонентом TilemapRenderer, що відображає плитки, використовується TilemapCollider2D, який дозволяє враховувати фізичні межі кімнати в системі колізій Unity. Кімната має спеціальний компонент RoomCollider – обгортка над колайдером, який дозволяє перевіряти наявність перетинів з іншими кімнатами під час генерації або визначати, коли гравець заходить у кімнату.

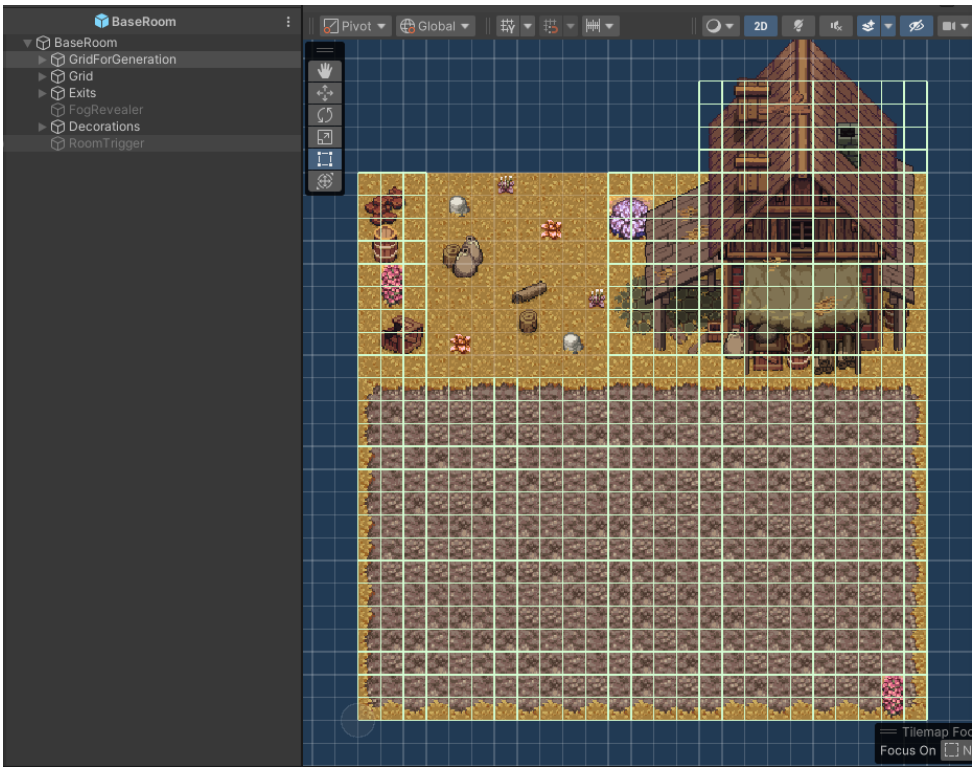

Рисунок 4.1 – Вигляд заготовки стандартної кімнати, та її елементи

Ще однією ключовою частиною структури є система виходів. Кожна кімната має набір виходів, які можуть бути з'єднані з іншими кімнатами. Вихід (додаток Б.2.1) зберігає інформацію про напрямок (Vector2Int), стан (вільний чи зайнятий), а також пов'язану кімнату з якою він з'єднаний (додаток Б.2.1). Саме завдяки виходам алгоритм генерації може будувати коридори між кімнатами або створювати петлі. Всі ці данні можливо вносити до налаштувань кімнати через редактор у Unity. Завдяки цьому можна зручно задавати параметри окремих виходів та додавати нові (рисунок 4.2).

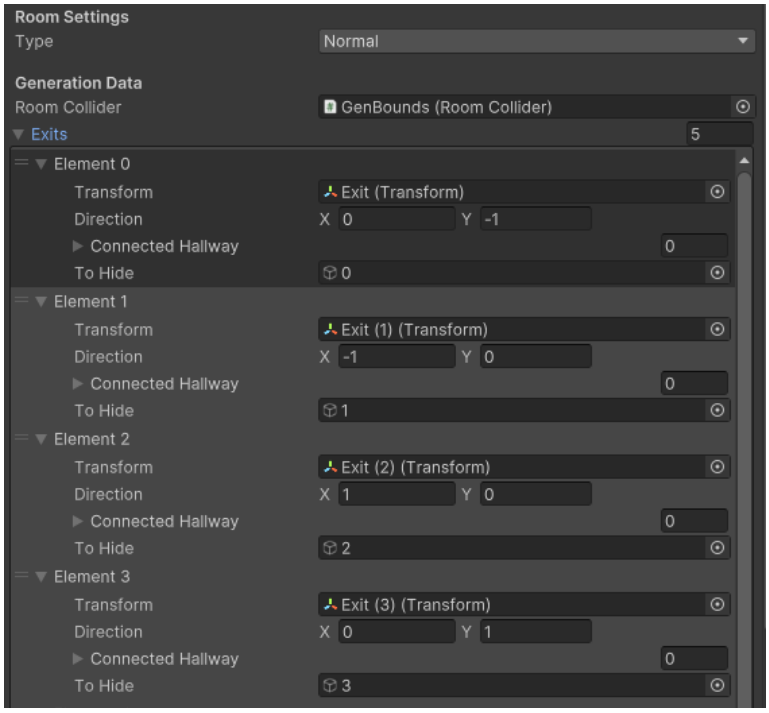

Рисунок 4.2 – Вигляд налаштувань виходів у редакторі

Існує кілька типів кімнат, кожна з яких виконує свою роль у геймплеї. Стартова кімната розташовується першою й слугує точкою входу для гравця, вона не містить ворогів і не має додаткової логіки. Бойова кімната, навпаки, запускає спеціальний скрипт керування хвилями ворогів одразу після того, як гравець входить до неї. Цей скрипт, реалізований у вигляді компонента WaveController, відповідає за появу ворогів у певних точках, слідкує за їх кількістю і після знищення всіх супротивників завершує хвилю, відкриваючи двері або активуючи подальші події. Так само як і налаштування виходів, можна налаштовувати хвилі через серіалізацію у редакторі. Можна додавати

ворогів які будуть з’являтися у певних місцях під певної хвилі (рисунок 4.3). Магазин – це ще один тип кімнати, який виконує допоміжну функцію. У ньому немає ворогів, зате гравець може придбати нові предмети або взаємодіяти з NPC.

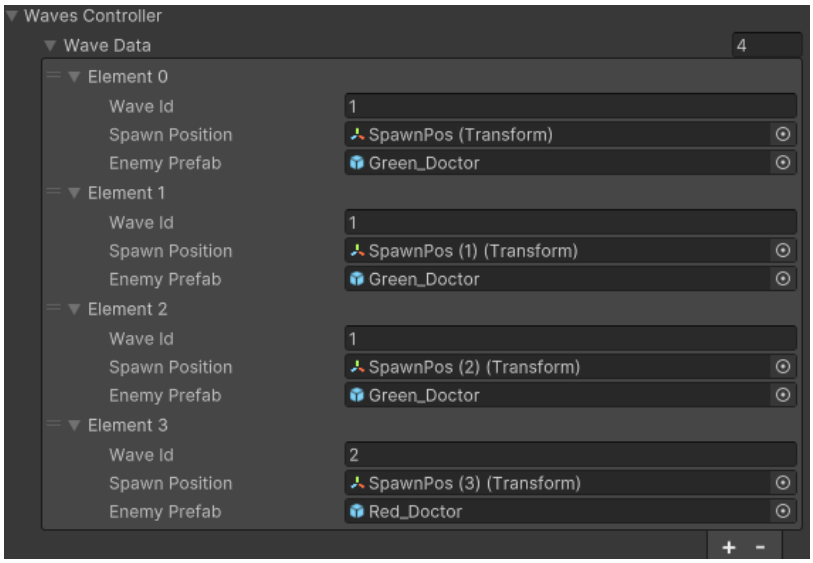

Рисунок 4.3 – Налаштування хвиль ворогів у редакторі

Сама логіка входу в кімнату реалізована через тригер. В межах кімнати знаходиться BoxCollider2D[9], що дозволяє йому не блокувати рух, але спрацьовувати при вході об'єктів. Коли гравець заходить у таку кімнату, перевіряється її тип і активується відповідна логіка – наприклад, у бойовій кімнаті це запуск хвилі ворогів.

Важливою технічною частиною реалізації кімнат є система Tilemap. Tilemap – це текстура розділена на окремі клітинки які використовуються як спрайти якими можна заповнювати простір та створювати ландшафт на сітці (рисунок 4.4). Вона дозволяє швидко створювати кімнати довільної форми, задавати колізії, а також створювати зображення кімнати для використання на мінікарті. Це досягається шляхом обходу плиток Tilemap і створення текстури, де пікселі відповідають наявності або відсутності плитки у відповідному місці. У результаті можна згенерувати точне зображення кімнати з прозорими областями, які не заповнені плитками. Пізніше ці зображення використовуються для формування мінікарти в реальному часі.

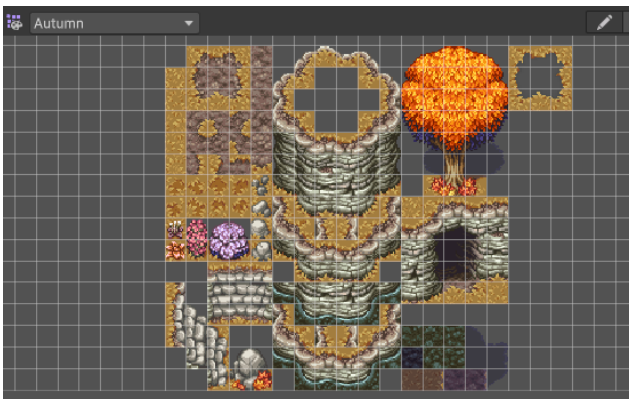

Рисунок 4.4 – Вигляд редактору TileMap

У процесі генерації кімнати з'єднуються між собою через систему виходів. Алгоритм спочатку вибирає вільний вихід попередньої кімнати, шукає до нього протилежний вихід у новій кімнаті, розміщує нову кімнату так, щоб виходи збігалися, і при необхідності створює між ними коридор (додаток Б.2.2). Коридори також можуть мати повороти, якщо пряме з'єднання неможливе через перепони. Завдяки цьому структура рівня набуває логічної зв'язності, а сам ігровий процес залишається динамічним. Сам коридор складається з сегментів які виставляються у ряд між потрібними виходами (рисунок 4.5). Сегменти мають різний вигляд для більш органічного вигляду проходу між кімнатами.

Рисунок 4.5 – Вигляд з’єднання між двома кімнатами

4.1.2 Генерація рівню

Основою системи побудови рівнів у грі є DungeonGenerator – головний клас, який керує процедурною генерацією кімнат, їх з'єднанням та формуванням логіки розміщення зон на основі попередньо заданої структури,

та може бути налаштований у редакторі (рисунок 4.6). Цей клас взаємодіє зі спеціальними класами генераторів, кожен із яких відповідає за побудову певного типу частини рівня – гілок або циклів (додаток Б.2.3). Також у ньому задається поверхня NavMeshSurface[10] на якій буде згенеровані області по яких зможуть рухатися актори що керуються штучним інтелектом.

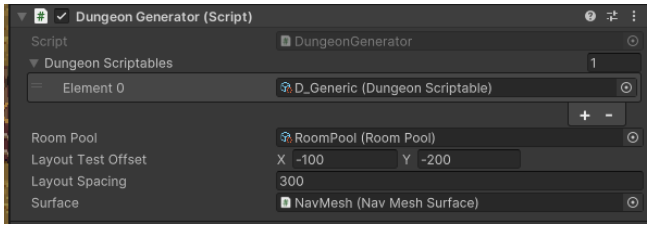

Рисунок 4.6 – Налаштування DungeonGenerator у редакторі

У проєкті для опису структури майбутнього підземелля використовується ScriptableObject[11] клас DungeonScriptable, який містить список елементів типу DungeonLayout. Кожен DungeonLayout зберігає інформацію про тип сегменту (branch або loop), список кімнат, які він має містити, а також параметри з'єднання з іншими сегментами, включаючи тип з'єднання (через першу, останню або випадкову кімнату). Використання ScriptableObject дозволяє створювати DungeonScriptable як окремі файли та редагувати їх перед використанням (рисунок 4.7). DungeonScriptable задається у редакторі DungeonGenerator.

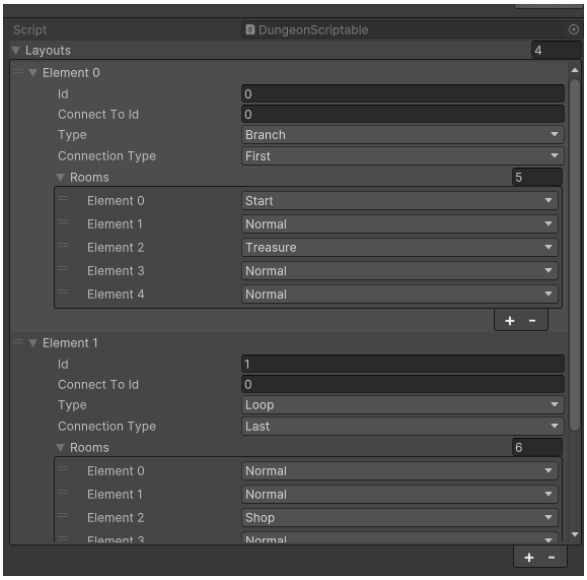

Рисунок 4.7 – Налаштування DungeonScriptable

При старті гри DungeonGenerator випадково вибирає одну з доступних конфігурацій підземелля (DungeonScriptable) і починає поетапно створювати всі зазначені DungeonLayout. Кожен такий сегмент обробляється окремо, для нього створюється кореневий об'єкт (layoutRoot), який розміщується далеко від основної сцени. Це дозволяє спочатку згенерувати всі макети окремо, без ризику накладання чи перетинів між ними, і лише після цього почати їх з'єднання. Генерація самого макету виконується окремими класами, які наслідують LayoutGenerator. Існують два основних типи: LoopGenerator та BranchGenerator.

LoopGenerator – створює петлеподібну структуру, де остання кімната з'єднується назад із початковою. Спочатку розміщуються кімнати послідовно в одному напрямку, використовуючи вільні виходи. Після цього останні кімнати по черзі з'єднуються назад до стартової кімнати через спеціальні коридори з поворотами (додаток Б.2.4). Це дозволяє побудувати замкнуту петлю (рисунок 4.8).

Рисунок 4.8 – Вигляд створеного циклу з кімнат

BranchGenerator – створює лінійну гілку, яка починається з вхідної кімнати (іноді переданої з іншого генератора) і послідовно додає нові кімнати, підбираючи виходи у заданому напрямку. Кожна нова кімната

з'єднується з попередньою через hallway (коридор), який автоматично подовжується, якщо при розміщенні виникає колізія. (додаток Б.2.5) Це дозволяє побудувати гілку з кімнат (рисунок 4.9).

Рисунок 4.9 – Вигляд створеної гілки з кімнат

Кожен LayoutGenerator обробляє свої кімнати, намагаючись з'єднати їх без перетинів. Якщо при з'єднанні двох кімнат виникає колізія (використовуючи RoomCollider і Physics2D.OverlapCollider[12]), система або пробує інші виходи, або замінює поточну кімнату на іншу того ж типу з пулу кімнат.

Після завершення генерації всіх макетів, DungeonGenerator починає розміщення макетів у світі. Для цього він послідовно бере кожен макет і з'єднує його з попереднім. Для з'єднання вибираються пари виходів (один із кінцевої кімнати попереднього макету, інший з початкової кімнати нового макету), і кореневий об'єкт (layoutRoot) макету переміщується у таку позицію, щоб виходи збіглися, та будується коридор між макетами (рисунок 4.10). Якщо виявляється, що макет накладається на вже згенеровану частину, то процес з'єднання повторюється з іншими виходами або сегментами, поки не знайдеться підходящий варіант.

Рисунок 4.10 – Вигляд згенерованих кімнат після з’єднання макетів

Цей підхід дозволяє повністю розділити етапи побудови та з'єднання макетів, спростити генерацію та уникнути перетинів. Кожен LayoutGenerator працює ізольовано до моменту вставлення в загальну структуру рівня, що також дозволяє в майбутньому зручно реалізовувати попередній перегляд або тестування окремих частин карти.

4.2 Предмети та інвентар

У грі реалізована модульна система предметів та інвентарю, що дозволяє гравцеві збирати, переміщати, використовувати та взаємодіяти з різними об'єктами в межах одного механізму. Основу становить абстракція предмету, що реалізована через базовий скрипт Item. Всі типи предметів – як зброя (додаток Б.2.7), так і кристали (додаток Б.2.6), наслідують цей клас і мають спільну структуру властивостей та методів. Це дозволяє легко масштабувати систему, додаючи нові типи предметів у майбутньому.

Кожен предмет має основні характеристики: назву, тип, іконку для

інвентарю та опис. Крім цього, певні предмети мають унікальні властивості. Наприклад, кристали – це пасивні модифікатори, які впливають на характеристики персонажа, зокрема атаку, здоров'я, швидкість тощо. Вони містять список об'єктів типу StaffStat (додаток Б.2.8), які мають поля StatType, ApplyType (додавання або множення) та числове значення. Зміна статів застосовується при активації або видаленні кристала з інвентарю. Усе це можна змінювати через редактор обраного предмету. На рисунку можна побачити налаштування магічно кристалу, який додає магічну шкоду до снарядів, та підвищує їх швидкість на 3 одиниці (рисунок 4.11).

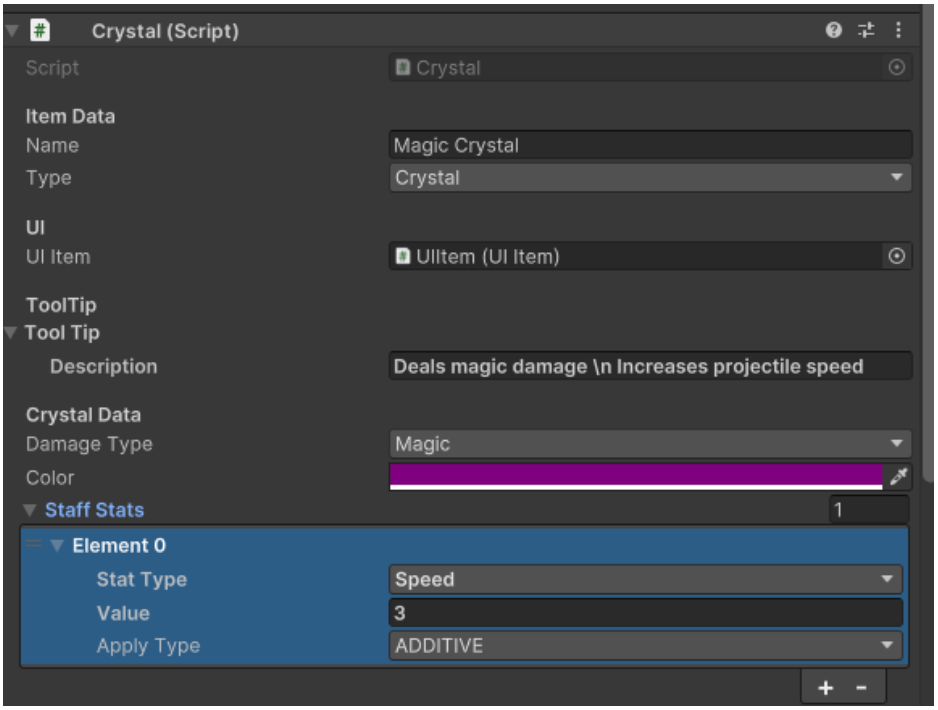

Рисунок 4.11 – Вигляд налаштувань магічного кристалу

Для взаємодії предметів із гравцем використовується система взаємодії, побудована на компоненті Interactable. Усі об'єкти, які можуть бути підібрані або активовані, мають цей компонент. Він відповідає за спрацьовування тригера, коли гравець входить у зону взаємодії, та передає об'єкт до інвентарю гравця (рисунок 4.12).

Рисунок 4.12 – Предмет, який можна підібрати, позначається білим контуром

Інвентар представлений у вигляді масиву слотів, кожен з яких може містити один предмет. Гравець може переміщати предмети між слотами, або викидати їх. Кожен слот має логіку взаємодії з предметом. Предмети можна переміщувати між слотами, накладати на зброю або викидати назад у світ (додаток Б.2.12). Для візуального представлення використовується Inventory (додаток Б.2.13), який зв'язаний із даними інвентарю і оновлюється при кожній зміні. Інвентар має можливість відображення лише активних слотів або всієї сітки, а також автоматичного підбору предметів. При наведенні на предмет, з’явиться вікно із інформацією про цей предмет, та ефектом який буде якщо його викласти на зброю (рисунок 4.13).

Рисунок 4.13 – Вигляд вікна інформації про предмет

Кожна дія з інвентарем супроводжується відповідним ефектом у грі. Наприклад, коли гравець підбирає кристал, статистика його персонажа оновлюється автоматично. Коли гравець змінює зброю – стара деактивується, а нова активується і починає використовуватися при натисканні кнопки

атаки.

Візуальну складову забезпечено UI-елементами на Canvas[13]. Кожен слот – це кнопка з зображенням і рамкою, яка підсвічується, коли слот активний або вибраний. Анімації для відкриття інвентарю, перетягування предметів і підказки взаємодії створюють приємний та інтуїтивний інтерфейс.

Рисунок 4.14 – Вигляд інвентарю гравця

4.3 Ігрові актори

У грі реалізовано систему ігрових акторів – об'єктів, які можуть взаємодіяти між собою, отримувати пошкодження, переміщатися, виконувати поведінку та мати власні характеристики. Усі актори реалізовані через базовий клас GameActor (додаток Б.2.14), від якого наслідуються гравець та різні типи ворогів. У класі реалізована система визначення напряму погляду (в залежності від кута actorRotation), а також зміна стану анімації.

Компонент Animator[14], прив'язаний до GameActor, дозволяє плавно відображати зміну станів. У проєкті використовується система Animator Controller (рисунок 4.15) зі змінними булевими даними для різних умов виконання анімацій, таких як біг або отримання шкоди. Вони оновлюються

кожен кадр відповідно до стану актора.

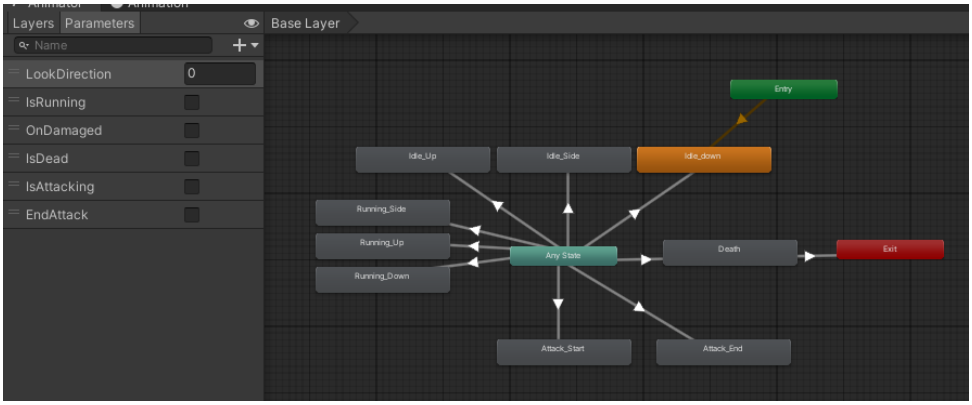

Рисунок 4.15 – Схема роботи контролера анімацій актора

Кожен актор в грі має компонент ActorDamageController (додаток Б.2.10), який відповідає за обробку отриманого пошкодження. Цей компонент визначає кількість здоров'я, тривалість невразливості після удару, колір підсвічування та обробку реакції на удар. Він також взаємодіє зі снарядами (Projectile), реагуючи на попадання в залежності від того, чи може цей снаряд завдати шкоди конкретному типу актора.

Система пошкоджень DamageModel (додаток Б.2.11) визначає, як актор приймає удари. Вона враховує тип пошкодження, візуальний ефект (рисунок 4.16), можливу анімацію та частинки. Цей компонент також зберігає окремі налаштування для кожного типу пошкодження, дозволяючи по-різному реагувати на вогонь, електрику, фізичну силу тощо. Наприклад, при отриманні електричного удару ворог може змінити колір, показати іскри та сповільнитися на кілька секунд.

Рисунок 4.16 – Вигляд акторів під різними ефектами

Для гравця використовується компонент керування, який обробляє

введення через систему Input System від Unity. Це забезпечує рух, анімації та стрільбу в потрібному напрямку. Гравець має зону зіткнення, та тригери для взаємодії з об'єктами. Вороги реалізовані як нащадки GameActor з навігаційною системою, побудованою на базі NavMeshAgent. Кожен ворог має контролер поведінки, який визначає, що робити в даний момент: переслідувати гравця, атакувати, патрулювати або відпочивати. Поведінку ворогів детально описано в окремому підпункті системи поведінки.

Актори можуть бути знищені, якщо їх здоров'я знижується до нуля. У цьому випадку активується відповідний візуальний ефект, об'єкт знищується.

4.4 Система поведінок

Для забезпечення інтелектуальної поведінки ворогів у грі було реалізовано модульну систему керування діями штучного інтелекту. Основна її мета – забезпечити ворогам здатність реагувати на зміну ситуації: наближатися до гравця, атакувати, чекати, патрулювати. Кожен тип ворога має індивідуальний набір можливих дій, обраний на основі умов середовища, наявності цілі, пріоритетів та затримок.

В центрі архітектури знаходиться клас BehaviorController (додаток Б.2.15), який закріплюється за кожним актором типу AIActor. Цей контролер відповідає за вибір та виконання актуальної поведінки з пулу поведінок, що попередньо додаються до нього під час ініціалізації.

Контролер зберігає всі поведінки у вигляді структурованого словника, де ключем є тип поведінки (BehaviorType – attack, move, idle), а значенням – список відповідних поведінкових об'єктів. Використання SortedDictionary дозволяє організувати пріоритети між типами поведінки: дії з вищим пріоритетом перевіряються раніше, і при наявності умов – виконуються замість менш важливих.

Крім того, для кожного типу поведінки в контролері зберігається інформація про затримку (cooldown) – час, який має пройти з моменту

останнього виконання дії цього типу, перш ніж її знову можна буде активувати.

Кожна поведінка реалізується як окремий клас, що наслідує базовий абстрактний клас BehaviorBase. У цьому класі визначено методи перевірки на початок виконання, корутина[15] яка описує поведінку, метод перевірки можливості переривання виконання поведінки.

У кожному кадрі BehaviorController виконує метод Tick(), у якому відбувається перевірка наявності доступних до виконання поведінок з урахуванням затримок, умов виконання та пріорітетів поведінок. Якщо знайдено поведінку, яка відповідає всім умовам, вона активується через StartBehavior(), який зупиняє попередню дію і запускає нову корутину. Корутини у Unity дозволяють асинхронно виконувати логіку (наприклад, рух до гравця або цикл стрільби), не блокуючи інші дії.

У рамках гри реалізовано кілька типів поведінок, серед яких:

- idle – актор стоїть на місці та нічого не робить. Використовується як дефолтна поведінка у відсутності інших варіантів;
- wander – випадкове блукання по кімнаті, актуальне для ворогів у пасивному стані;
- approach – ворог наближається до гравця на задану дистанцію;
- attack – серія поведінок для атаки з різними шаблонами: стрільба по колу, спіраль, суператака тощо. Вони мають тривалість виконання та кулдаун.

Особливістю є те, що вороги можуть мати одразу кілька атак з різною складністю, які перемикаються залежно від ситуації – наприклад, ворог може почати з простої атаки, а після затримки виконати складнішу.

Завдяки використанню корутин, дії не блокують виконання розрахунку кадру гри й можуть завершуватись як автоматично, так і вручну – наприклад, після знищення ворога або коли поведінку потрібно примусово перервати. Це дозволяє динамічно реагувати на зміну ситуації без збоїв у логіці AI.

4.5 Зброя та снаряди

Система стрільби у грі реалізована за допомогою декількох класів, які тісно взаємодіють між собою. Основним компонентом, відповідальним за стрільбу гравця, є скрипт PlayerShooter. При натисканні кнопки миші гравець ініціює створення снаряда. Напрямок, у якому цей снаряд буде рухатись, розраховується на основі положення миші у світових координатах. Це дозволяє гравцеві стріляти у довільному напрямку, що є типовим для шутерів із вільною стрільбою (twin-stick).

Снаряди створюються не безпосередньо у PlayerShooter, а за допомогою централізованого менеджера BulletManager (додаток Б.2.16). Цей клас відповідає за створення та знищення снарядів, а також зберігає посилання на всі активні об'єкти. У BulletManager містяться методи, які дозволяють створити снаряд із зазначенням параметрів: позиції запуску, швидкості, напрямку, типу шкоди та її сили. Завдяки цьому механізму можна створювати різні типи снарядів як для гравця, так і для ворогів, просто вказавши відповідні значення.

Клас Projectile (додаток Б.2.17) реалізує логіку самого снаряду. Після створення він отримує напрямок руху, обчислює швидкість та переміщується у відповідному напрямку щосекунди. У проекті реалізовано обмеження часу життя снаряду: кожен має параметр lifeTime, після завершення якого він автоматично знищується, навіть якщо не влучив у ціль. Це дозволяє уникнути перенасичення сцени надлишковими об'єктами.

Снаряди можуть бути простими або з особливою поведінкою. У другому випадку снаряд має прикріплений об'єкт класу BulletScript, який може реалізувати довільну поведінку: наприклад, спіральну траєкторію, пульсування, затримку перед стартом або пошук цілі. Для цього Projectile делегує методи оновлення позиції та швидкості на Bullet, якщо у нього активовано прапор overrideMovement. Таку систему зручно масштабувати – для створення нового патерну достатньо реалізувати свій підклас Bullet.

Коли снаряд стикається з ціллю, він передає свої параметри класу ActorDamageController. Цей компонент перевіряє, чи дійсно снаряд може завдати шкоди: наприклад, гравець не повинен отримувати шкоди від своїх власних снарядів. Якщо усі умови виконані, шкода застосовується. При цьому також враховується тип шкоди, адже актор може бути вразливим або стійким до певних її видів. Додатково активується візуальний ефект отримання шкоди через DamageModel (рисунок 4.17).

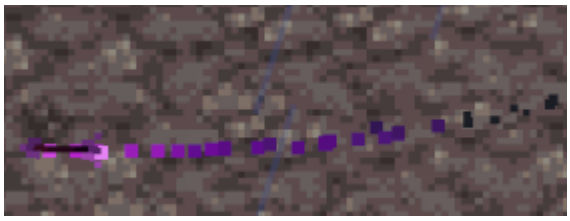

Рисунок 4.17 – Вигляд снаряду із магічним ефектом

Окрему роль у візуалізації відіграє параметр Rotatable. Якщо він увімкнений, снаряд автоматично обертається у напрямку свого руху, що покращує візуальне сприйняття стрільби. Також можна використовувати різні візуальні варіанти снарядів залежно від типу шкоди або джерела атаки, що дозволяє створити візуальну ідентичність для кожного типу ворога чи зброї.

4.6 Інтерфейс

Інтерфейс гравця виконує кілька ключових завдань: він надає гравцеві постійний візуальний доступ до стану його персонажа, а також орієнтує у просторі за допомогою міні-мапи. Структурно інтерфейс побудований на основі системи Canvas у Unity, з використанням UI-елементів, таких як Image, Text та RectTransform[13].

Параметри гравця — це перше, що гравець бачить під час гри. Сюди відносяться показники здоров’я, синього здоров’я (якщо воно використовується як тимчасовий захист чи щит), а також інші

характеристики, такі як кількість предметів, зброї чи ресурсів. Основні дані беруться з класу ActorDamageController, який зберігає поточне та максимальне здоров’я, а також з ActorStats, який може містити додаткову інформацію, таку як бонус до шкоди або швидкість. Інтерфейс постійно оновлюється відповідно до цих даних. Наприклад, якщо гравець отримав шкоду, нове значення здоров’я з ActorDamageController.Health автоматично оновлює UI-елемент здоров’я.

Здоров'я гравця відображається як набір сердець-контейнерів, кожне з яких може бути порожнім, заповненим чи підсвіченим синім кольором – щоб показати тимчасове здоров’я (рисунок 4.18). Цей підхід дозволяє гравцеві миттєво оцінити, скільки залишилося постійного здоров'я і скільки – тимчасових точок життя. Це реалізовано з використанням Layout Group, до якого динамічно додаються або видаляються Image-компоненти у формі сердець. При зміні значення здоров'я в ActorDamageController відбувається перерахунок кількості заповнених та синіх сердець: наприклад, якщо гравець отримав пошкодження, відповідна кількість сердець спливає в нульовий стан із прозорістю, а якщо отримано тимчасовий захист (BlueHealth), то він додається у вигляді окремих синіх сердець після основних.

Оновлення UI виконується безпосередньо у методі, що слухає події зміни здоров'я: він очищає поточний набір сердець у контейнері, а потім, враховуючи Health і BlueHealth, створює стільки ж нових Image-елементів із відповідними спрайтами (порожнє, червоне чи синє серце). Використання Layout Element і самого Horizontal Layout Group гарантує, що всі серця рівномірно вирівняні та змінюють своє положення автоматично при зміні їхньої кількості.

Рисунок 4.18 – Інтерфейс параметрів гравця

Другим важливим елементом інтерфейсу є міні-мапа, яка дозволяє гравцеві орієнтуватися у світі гри, бачити вже пройдені кімнати, коридори та поточне положення персонажа. Вона реалізована як окрема секція UI з вкладеними об'єктами: MapRect, Mask, RoomsTransform, де кожен Room представляється окремим Image[16] (рисунок 4.19).

Під час запуску гри кожна кімната створює текстуру, яка відповідає її формі. Це здійснюється з класу Map (додаток Б.2.18), який проходиться по Tilemap підлоги кімнати й малює піксель білого кольору там, де є плитка, та прозорий — де її немає. Таким чином створюється текстура, яка точно відповідає формі кімнати. Ця текстура конвертується в Sprite, який застосовується до Image елементу на міні-мапі. Кожен з таких RoomImage прикріплюється до roomsTransform, і його позиція на мапі розраховується за допомогою методу WorldToMinimapPosition(), який масштабно переносить позицію кімнати з простору світу в координати UI. MapRect містить Animator, який дозволяє збільшувати міні-мапу щоб краще побачити які кімнати знаходяться навколо гравця.

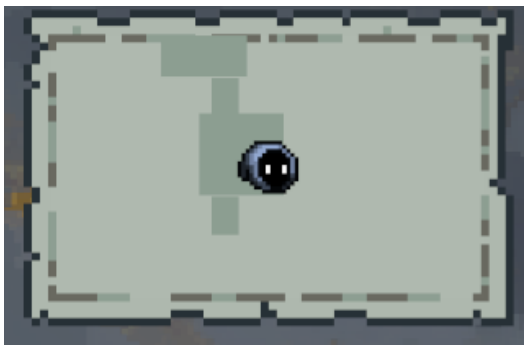

Рисунок 4.19 – Вигляд міні-мапи

Особливістю реалізації міні-мапи є те, що кімнати не просто відображаються — вони зміщуються відповідно до руху гравця. Об’єкт roomsTransform, який містить усі зображення кімнат, зміщується у зворотному напрямку відносно руху камери, створюючи ефект руху гравця по мапі. Таким чином, сам гравець на мапі завжди залишається по центру, що значно покращує орієнтацію у просторі.

Щоб міні-мапа не виходила за межі свого вікна, використовується маска, яка обрізає будь-які елементи за межами прямокутної області. Це дозволяє створити акуратну, інтегровану частину інтерфейсу без зайвих відволікаючих елементів.

4.7 Система туману війни

У нашому проєкті система туману війни реалізована за допомогою власної системи частинок, яка малює сотні хмаринок довільного розміру і кольору, що плавно рухаються навколо камери й поступово зникають там, де гравець уже побував або куди відкрилися нові кімнати (рисунок 4.20). Основна ідея полягає в тому, щоб кожна частинка мала власну позицію у світових координатах і невеликий випадковий шум руху, а після виходу за межі сфери огляду (прямокутника навколо камери) вона не видаляється, а переміщується на протилежний бік. Такий підхід гарантує рівномірне покриття простору, без стрибків або дір у тумані.

Відкриття нових ділянок контролюється фізичними тригерами з шаром FogRevealer: коли частинка опиняється всередині колайдера , її прозорість плавно зменшується до нуля. Це дозволяє створити ефект поступового висвітлення простору без миттєвих сплесків прозорості. Навпаки, якщо гравець відходить від раніше відкритої зони, частинки знову нарощують свою непрозорість згідно з налаштованою швидкістю згасання (fadeSpeed), створюючи відчуття повільного затухання туману.

Для рендерингу використовується Graphics.DrawMeshInstanced[17]. Частинкам задаються матриця трансформу (позиція й масштаб), колір із градієнта й UV-зсув в текстурному атласі (щоб мати різні форми хмаринок без додаткових матеріалів). Усе це передається в MaterialPropertyBlock, що дозволяє малювати сотні частинок за один виклик і підтримувати прозорість та різноманітність вигляду. Створення власної системи (додаток Б.2.19) дозволило оптимізувати контроль над такою великою кількістю частинок та зберегти продуктивність гри.

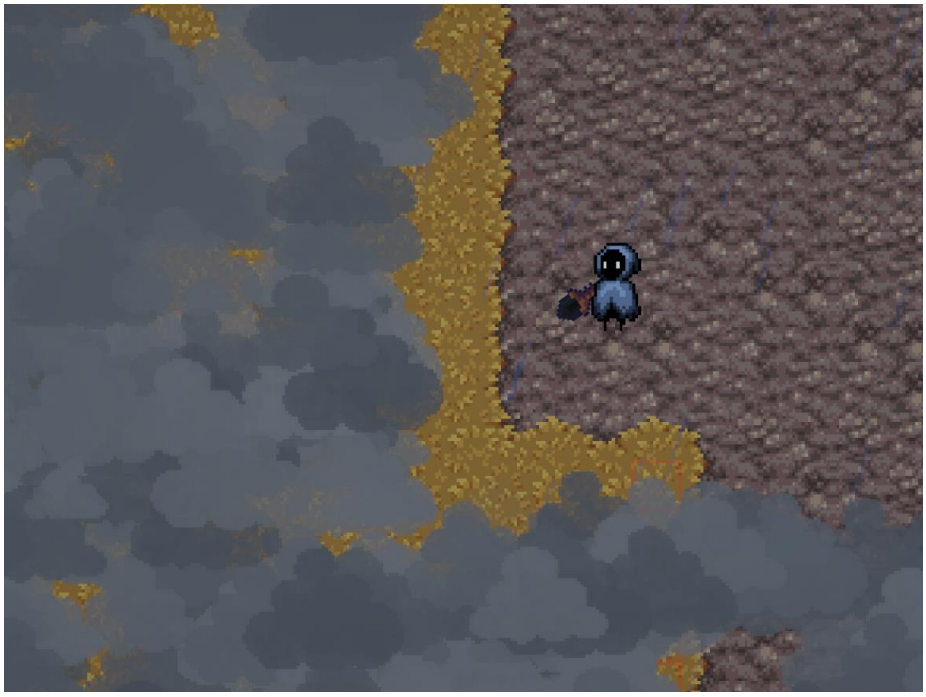

Рисунок 4.20 – Вигляд Туману війни над недоступною територією

ВИСНОВКИ

У кваліфікаційній роботі було розроблено 2D roguelike-гру на рушії Unity з використанням мови програмування C#, яка поєднує процедурну генерацію рівнів, бойову систему, інвентар, штучний інтелект ворогів та інтерактивний інтерфейс. Актуальність створення такої гри обумовлена зростанням популярності інді-ігор жанру roguelike та необхідністю дослідження ефективних методів побудови реіграбельних систем. Отриманий продукт дозволяє демонструвати переваги процедурної генерації та модульної архітектури, що є важливим внеском у розробку гнучких ігрових механік.

Проведено теоретичний аналіз існуючих підходів до створення roguelike-ігор Для процедурної генерації рівнів було використано алгоритм випадкового розміщення кімнат з подальшим з'єднанням їх коридорами, що забезпечує різноманітність ігрових просторів при збереженні прохідності карти. Бойова система спроектована на основі обробки колізій і менеджера станів персонажів, що дає змогу моделювати атаки й отримання пошкоджень. Штучний інтелект ворогів реалізовано через контролер поведінки, який переходять між режимами патрулювання, переслідування і атаки залежно від положення гравця, – такий підхід забезпечує адаптивну реакцію супротивників без значних витрат ресурсів. Інтерактивний інтерфейс розроблено, враховуючи принципи ергономіки та зворотного зв'язку з користувачем. Обрані підходи до розробки, зокрема використання Unity та C#, обґрунтовано їхньою поширеністю у 2D-розробці.

У результаті реалізації вдалося досягти повноцінного функціонування усіх запланованих елементів гри. Побудовано систему процедурної генерації рівнів, яка формує зв'язні та функціональні карти при кожному запуску. Бойова механіка працює з урахуванням різних параметрів зброї, впливу модифікаторів та ефектів пошкодження. Реалізовано систему керування

інвентарем і взаємодії з предметами, що дає змогу змінювати ігрову стратегію. Вороги демонструють базову самостійну поведінку, що створює виклик для гравця.

Подальший розвиток гри може включати розширення контенту – додавання нових видів ворогів, предметів, зброї та унікальних кімнат. Суттєвим кроком може стати покращення штучного інтелекту супротивників для створення глибшої тактики. Існує потенціал для реалізації системи мета-прогресу, збереження частини розвитку між проходженнями, що підвищить зацікавленість гравців. Додаткові технічні перспективи полягають у портативності проєкту на різні пристрої, підтримці мультиплеєру, а також оптимізації продуктивності.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Адамс Е. Основи проектування ігор / Ернест Адамс ; пер. з англ. – Київ : Наш Формат, 2021. – 464 с.

2. Rollings A., Adams E. Andrew Rollings and Ernest Adams on Game Design. – New Riders, 2003. – 648 p.

3. The Binding of Isaac Wiki [Електронний ресурс]. URL: https://bindingofisaacrebirth.fandom.com/ (дата звернення: 23.05.2025).

4. Enter the Gungeon Wiki [Електронний ресурс]. URL: https:// enterthegungeon.wiki.gg / (дата звернення: 23.05.2025).

5. Microsoft. C# Programming Guide [Електронний ресурс] / Microsoft. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 23.05.2025).

6. Visual Studio Code Documentation [Електронний ресурс] / Microsoft. URL: https://code.visualstudio.com/docs (дата звернення: 23.05.2025).

7. Unity Version Control Documentation [Електронний ресурс] / Unity Technologies. URL: https://docs.unity.com/version-control (дата звернення: 23.05.2025).

8. Unity Manual, Instantiating Prefabs [Електронний ресурс] / Unity Technologies. – URL: https://docs.unity3d.com/Manual/InstantiatingPrefabs.html (дата звернення: 18.06.2025).

9. Unity Manual, 2D Colliders [Електронний ресурс] / Unity Technologies. URL: https://docs.unity3d.com/Manual/Collider2D.html (дата звернення: 18.06.2025).

10. Unity Scripting API, NavMesh [Електронний ресурс] / Unity Technologies. URL: https://docs.unity3d.com/ScriptReference/AI.NavMesh.html (дата звернення: 18.06.2025).

11. Unity Manual, ScriptableObject [Електронний ресурс] / Unity Technologies. URL: https://docs.unity3d.com/Manual/class-ScriptableObject.html

(дата звернення: 18.06.2025).

12. Unity Manual, Physics 2D [Електронний ресурс] / Unity Technologies. URL: https://docs.unity3d.com/Manual/Physics2D.html (дата звернення: 18.06.2025).

13. Unity Manual, UI Toolkit (Canvas & UI) [Електронний ресурс] / Unity Technologies. URL: https://docs.unity3d.com/Manual/UIToolkit.html (дата звернення: 18.06.2025).

14. Unity Manual, Animator Controllers [Електронний ресурс] / Unity Technologies. URL: https://docs.unity3d.com/Manual/AnimatorControllers.html (дата звернення: 18.06.2025).

15. Unity Manual, Coroutines [Електронний ресурс] / Unity Technologies. URL: https://docs.unity3d.com/Manual/Coroutines.html (дата звернення: 18.06.2025).

16. Unity Manual, UI Image [Електронний ресурс] / Unity Technologies. URL: https://docs.unity3d.com/Manual/script-Image.html (дата звернення: 18.06.2025).

17. Unity Scripting API – Graphics.DrawMeshInstanced [Електронний ресурс] / Unity Technologies. URL: https://docs.unity3d.com/ScriptReference/Graphics.DrawMeshInstanced.html (дата звернення: 18.06.2025).