Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук

Бакалаврська кваліфікаційна робота на тему:

Розробка 2D-гри в жанрі Roguelike з процедурною генерацією рівнів

Виконав: студент 4 курсу, групи 2КН-21б
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)
Сергій СКРИЦЬКИЙ
(прізвище та ініціали)

Керівник: асистент. каф. КН
Єгор СІЛАГІН
(прізвище та ініціали)
«04» червня 2025 р.

Рецензент: PhD, ст.викл. каф. САІТ
Ольга ВОЙЦЕХОВСЬКА
(прізвище та ініціали)
«05» червня 2025 р.

Допущено до захисту
Зав. кафедри проф., д.т.н. Андрій ЯРОВИЙ
«06» червня 2025 р.

Вінниця ВНТУ - 2025 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 122 Комп'ютерні науки
Освітньо - професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
проф., д.т.н. Андрій ЯРОВИЙ
«05» травня 2024р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
Скрицькому Сергію Дмитровичу

1. Тема роботи: Розробка 2D-гри в жанрі Roguelike з процедурною генерацією рівнів.
керівник роботи: Сілагін Єгор Олексійович, асистент кафедри КН
затверджені наказом вищого навчального закладу "20" березня 2025 року № 97
2. Термін подання студентом роботи 30 травня 2025 р.
3. Вихідні дані до роботи :
Мова програмування: C#;
Середовище розробки: Unity;
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
формулювання об'єкта, предмета, мети та задач подальшого дослідження; аналіз предметної області, огляд жанру roguelike та сучасних інструментів для розробки 2D-ігор;проєктування архітектури гри, система управління гравцем, ворогами та процедурну генерацію;програмна реалізація гри із використанням рушія Unity.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень):
Загальна архітектура програмного модуля гри; UML-діаграма класів системиблок-схеми алгоритмів управління гравцем, логіки ворогів, процедурної генерації рівнів; об'єктно-орієнтована модель даних про кімнати, об'єкти, персонажів і події гри.

6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-3	Єгор СІЛАГІН, асистент, каф. КН		

7. Дата видачі завдання 05 травня 2025 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва та зміст етапів	Термін виконання		Примітка
		початок	кінець	
1	Визначення об'єкта, предмета, мети та задачі подальшого дослідження	06.05.25	07.05.25	
2	Аналіз предметної області, існуючих методів, алгоритмів та інструментів для створення програмного модуля індивідуального планування спортивних тренувань	08.05.25	11.05.25	
3	Проектування програмних модулів для індивідуального планування спортивних тренувань	27.05.25	28.05.25	
4	Програмна реалізація програмного модуля індивідуального планування спортивних тренувань	30.05.25	01.06.25	
5	Оформлення пояснювальної записки	02.06.25	04.06.25	

Студент _____ Сергій СКРИЦЬКИЙ
(підпис) (прізвище та ініціали)

Керівник роботи _____ Єгор СІЛАГІН
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота містить 73 сторінки формату А4, включає 19 рисунків, 1 таблицю, список використаних джерел налічує 18 найменування.

Дослідження присвячено розробці 2D roguelike гри з процедурною генерацією рівнів. Основною метою роботи є створення програмного продукту, який поєднує в собі елементи випадкової генерації, бойової механіки, динамічної складності та повторюваності, використовуючи сучасні засоби програмування для створення ігор.

Було реалізовано архітектуру гри з розділенням на логічні модулі: генерація рівнів, бойова система, управління гравцем, штучний інтелект ворогів, інтерфейс користувача. Побудовано UML-діаграму класів, що описує основні об’єкти та взаємозв’язки між ними. Рівні генеруються процедурно за допомогою модифікованого алгоритму пошуку в глибину, що дозволяє формувати унікальні карти з гарантією зв’язності та варіативністю.

Unity було обрано як основний рушій для реалізації гри завдяки широким можливостям візуалізації, роботі з Tilemap, підтримці анімації, фізики та взаємодії з користувачем. Мова програмування C# використовується для реалізації логіки взаємодії між об’єктами гри, обробки зіткнень, реалізації штучного інтелекту та бойової системи.

Проведено тестування стабільності генерації рівнів із використанням фіксованих seed-значень, а також порівняльний аналіз застосованого алгоритму з іншими підходами (BSP, Perlin Noise) за критеріями реіграбельності, збалансованості та глибини геймплею.

Результатом роботи є функціональний прототип гри, що демонструє можливості процедурної генерації, гнучкості архітектури та потенціал до масштабування.

Ключові слова: roguelike, 2D-гра, процедурна генерація , генерація рівнів, seed.

ABSTRACT

The bachelor's thesis contains 73 pages in A4 format, includes 18 figures, 1 tables and the list of references contains 18 items.

The research is devoted to the development of a 2D roguelike game with procedural level generation. The main goal of the work is to create a software product that combines elements of random generation, combat mechanics, dynamic complexity, and repeatability, using modern programming tools for game development.

The game architecture was implemented with a division into logical modules: level generation, combat system, player control, enemy artificial intelligence, and user interface. A UML class diagram was constructed, describing the main objects and the relationships between them. Levels are generated procedurally using a modified depth-first search algorithm, which allows for the creation of unique maps with guaranteed connectivity and variability.

Unity was chosen as the main engine for the game's implementation due to its extensive visualization capabilities, work with Tilemap, support for animation, physics, and user interaction. The C# programming language is used to implement the logic of interaction between game objects, collision handling, artificial intelligence, and the combat system.

The stability of level generation was tested using fixed seed values, and a comparative analysis of the applied algorithm with other approaches (BSP, Perlin Noise) was conducted based on the criteria of replayability, balance, and gameplay depth.

The result of the work is a functional prototype of the game that demonstrates the capabilities of procedural generation, architectural flexibility, and scalability potential.

Keywords: roguelike, 2D game, procedural generation, level generation, seed.

ЗМІСТ

ВСТУП

Актуальність досліджень. Індустрія відеоігор є однією з найдинамічніших і найперспективніших сфер сучасних інформаційних технологій. Вона постійно розвивається завдяки стрімкому зростанню обчислювальних потужностей, доступності інструментів розробки та зростаючому інтересу з боку користувачів різного віку. Особливої популярності набули інді-ігри, які завдяки креативності розробників здатні запропонувати унікальні ігрові механіки та неповторну атмосферу.

Останніми роками на фоні зростаючого комерційного тиску “високобюджетні проекти великих студій” дедалі частіше втрачають довіру гравців через одноманітний геймплей, надмірну монетизацію, численні технічні проблеми та брак інновацій. У той час як великі компанії часто орієнтуються на прибуток і масовість, “незалежні розробники (інді-студії)” створюють ігри з глибокими механіками, нестандартними ідеями та високою увагою до деталей. Це дозволяє їм отримувати схвальні відгуки гравців та здобувати визнання в ігровій спільноті.

У сучасному інформаційному просторі відеоігри стали не лише формою розваги, ай важливим засобом творчого самовираження, навчання та моделювання складних систем.

Одна з основних проблем, яка існує у більшості реалізацій подібного жанру, — обмежена варіативність контенту, що виникає внаслідок використання примітивних або шаблонних методів процедурної генерації. Це знижує реіграбельність гри, викликає відчуття повторюваності, а також ускладнює створення збалансованих і водночас цікавих рівнів. Класичні методи, як-от BSP-розбиття чи шум Перліна, хоч і забезпечують певну варіативність, часто не гарантують ігрової логіки, зручної навігації між кімнатами чи глибини геймплею.

Процедурна генерація контенту є важливою технологією в сучасній ігровій індустрії, яка дозволяє автоматично створювати великі обсяги

унікального ігрового середовища без необхідності ручного проектування кожного елемента. Завдяки цьому розробники можуть зменшити витрати на створення контенту, а гравці — отримати багатий і динамічний досвід.

Запропонована у межах цієї роботи 2D roguelike-гра допоможе оптимізації процедурної генерації рівнів через алгоритмічний підхід на основі збереження структурної зв'язаності з логічним розташування кімнат та контекстного розміщення ігрових об'єктів а також інтеграції з FSM для реалізації інтелектуальної поведінки ворогів різного типу.

Метою дослідження є покращення ігрового досвіду гравців у 2D roguelike-іграх. Досягти цього планується шляхом покращення та оптимізації алгоритмів процедурної генерації рівнів.

Предметом дослідження є методи, алгоритми та архітектурні рішення, що забезпечують розробку функціонального, масштабованого та варіативного ігрового середовища з високим рівнем реіграбельності.

Об'єктом дослідження є процес створення 2D-гри у жанрі roguelike з використанням процедурної генерації рівнів, керуванням гравцем, ворогами та реалізацією бойової системи.

Задачі подальшого дослідження:

1. Виконати програмну реалізацію 2D roguelike-гри з використанням алгоритмів процедурної генерації рівнів.

2. Провести тестування реалізованої гри в різних конфігураціях рівнів із використанням фіксованих seed-значень для перевірки стабільності та варіативності;

3. Здійснити порівняльний аналіз отриманих результатів із класичними методами генерації BSP, Perlin noise та іншими підходами за критеріями реіграбельності, збалансованості та глибини геймплею;

4. Оформлення пояснювальної записки.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз ключових характеристик ігрового жанру roguelike та тенденцій його розвитку

Жанр roguelike міцно закріпив за собою місце в сучасному ігровому світі. Він не лише вижив після появи на зорі 80-х, але й перетворився на значний плацдарм для незалежних творців відеоігор. Все почалося з Rogue (1980), гри, що визначила стандарти: випадкова генерація локацій, висока складність, остаточна смерть героя — саме ці елементи заклали фундамент жанру. Від назви цієї гри й пішла назва “roguelike” [1].

Рисунок 1.1 – Елемент геймплею з гри Rogue

Розробка ігор у стилі roguelike – це доволі комплексний процес, що потребує використання специфічних технічних прийомів, ретельно продуманого геймдизайну та глибоких знань алгоритмізації. Однією з найважливіших рис цього жанру є впровадження процедурної генерації рівнів, яка зазвичай досягається шляхом застосування алгоритмів на кшталт random walk, binary space partitioning, cellular automata, wave function collapse [2]. Використання таких алгоритмів дає можливість формувати нелінійні, унікальні просторові структури рівнів, що позитивно впливає на різноманітність ігрового досвіду.

Значну роль у розробці roguelike ігор відіграє балансування складності. У зв’язку з наявністю механіки перманентної смерті, важливо досягти рівноваги між викликом для гравця та можливістю досягнення поступового прогресу. Створення геймплейних сценаріїв має забезпечувати підвищення складності відповідно до зростання навичок користувача, уникаючи при цьому ефекту «несправедливих поразок» .

Мотивація до повторного проходження також є критичним компонентом проектування roguelike ігор. Це досягається через впровадження механізмів розблокування нового контенту, випадкових подій, альтернативних шляхів проходження та багаторазового використання змінних ігрових умов [3].

З технічної точки зору, для створення проектів у стилі roguelike застосовують широкий спектр програмних інструментів, з яких найбільш поширеними є:

Unity — ігровий рушій, що забезпечує підтримку 2D-графіки та пропонує безліч готових рішень [4];

Godot Engine — безкоштовний і відкритий рушій, який надає можливості гнучкого програмування та зручний інтерфейс редактора [5];

1. Процедурна генерація рівнів.
2. Складність і «несправедливість»
3. Випадковість. Розташування ворогів, ігрових об'єктів, приміщень цілком підпорядковане випадковості. Цей фактор непередбачуваності ключова риса жанру.

У 2008 році, під час конференції International Roguelike Development Conference, було презентовано термін “Berlin Interpretation”. Це спроба окреслити рамки жанру через сукупність певних вимог. Але вже з 2010-х стало очевидно, що roguelike зазнає трансформацій. Зокрема, виникають проекти, які поєднують класичні особливості з новаторськими підходами – їх називають roguelite. Вони можуть мати полегшені механіки смерті, збереження досягнень або навіть бойову систему в реальному часі замість покрокової.

Сучасне ігрова спільнота дедалі частіше відчуває розчарування від великобюджетних проектів (ААА-ігор).

Ігри, на які покладаються значні надії, випускаються в технічно недосконалому стані – з помилками, поганою оптимізацією та неякісним контентом. Часто створюється враження, що геймери виступають у ролі бета-тестувальників, ще й сплачують за це повну ціну. Крім технічних проблем, індустрія страждає від перенасичення клонами, однотипними франшизами, агресивною монетизацією, сезонними пропусками та мікротранзакціями, які дедалі більше замінюють сам ігровий процес. Усе це підриває довіру до великих студій.

На цьому тлі інді-сцена переживає справжнє піднесення. Незалежні розробники, не обмежені комерційними рамками, можуть створювати проекти з душею: оригінальні, глибокі, а головне — чесні перед гравцем. Саме в жанрі roguelike багато інді-хітів здобули визнання завдяки унікальному геймплею та високій реіграбельності.

До найбільш вражаючих прикладів останніх років варто зарахувати:

1. The Binding of Isaac: Rebirth (2014) – похмура гра на регіліозних мотивах з процедурною генерацією рівнів та глибокою різноманітністю предметів[6];
2. Dead Cells (2018) – динамічна roguelite-метроідванія;
3. Enter the Gungeon (2016) – яскравий шутер з великою кількістю зброї та абсурдним гумором.

Сучасні тенденції розвитку жанру демонструють, що сьогодні roguelike це не лише ізометричні підземелля з піксельним зображенням. Жанр активно практикує експерименти з піджанрами. Наприклад:

Slay the Spire – колекціонування карт поєднується з тактичною системою бою;

Cult of the Lamb – гра, що поєднує симулятор ферми, менеджмент культу та елементи roguelike.

Ці проекти не просто зберігають жанрову ідентичність, але й виходять за її рамки, приваблюючи нову аудиторію та відкриваючи нові обрії ігрового дизайну.

1.2 Аналіз методів процедурної генерації ігрових рівнів у roguelike-іграх

Процедурна генерація ігрового середовища стала невід'ємною частиною створення сучасних roguelike ігор. Цей підхід дозволяє створювати унікальні ігрові світи для кожного проходження, що значно збільшує реіграбельність. На відміну від ручного проектування рівнів, процедурна генерація економить час розробників і водночас забезпечує динамічне та адаптивне будування ігрового світу.

Розглянемо основні способи, які використовуються для створення двовимірних рівнів із кімнатами, коридорами, перешкодами та об'єктами, а також принципи їх практичного застосування в roguelike-іграх.

BSP (Binary Space Partitioning) – один із найпоширеніших алгоритмів. Його суть полягає в послідовному розділенні ігрової площини на прямокутні підобласті через рекурсивний поділ простору, чергуючи горизонтальні та вертикальні розрізи. У кожній утвореній зоні формуються кімнати, які потім з'єднуються коридорами. Цей метод забезпечує структурований рівень із контрольованими розмірами приміщень та гарантованою зв'язністю всіх зон. BSP найкраще підходить для створення чітких, логічно побудованих приміщень типу підземель, чи технічних споруд. Міносом методу є геометрична передбачуваність, яка потребує додаткової декоративної або механічної обробки [6].

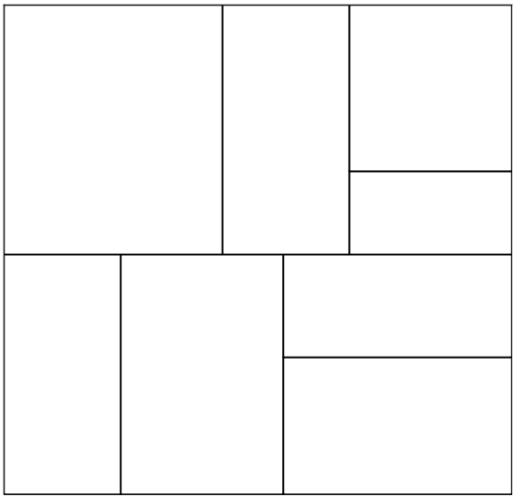

Рисунок 1.2 – Розділення площини алгоритмом BSP

Клітинкові автомати часто використовуються для моделювання природних ландшафтів. Алгоритм базується на сітці клітин, стан яких змінюється за певними правилами залежно від стану сусідніх клітин. Спочатку сітка наповнюється випадковими значеннями, і протягом кількох ітерацій формуються цілісні зони, що нагадують печери чи природні тунелі. Цей підхід створює органічне середовище без правильної геометрії, але може призводити до утворення ізольованих ділянок, що вимагає додаткових механізмів перевірки прохідності.

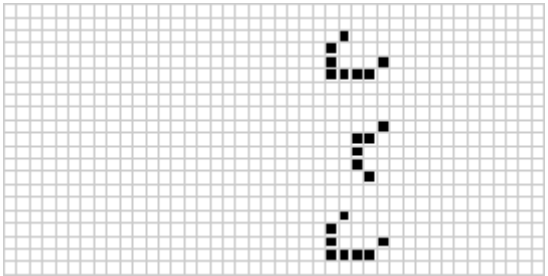

Рисунок 1.3 – Приклад використання методу клітиного автомату “гра життя”

Метод випадкового блукання грунтується на хаотичному заповненні простору. Віртуальний агент починає рух із певної точки і випадково змінює напрямок, залишаючи після себе прохід. Так утворюється звивиста мережа коридорів, яка створює ефект несподіванки для гравця. Метод ідеально

підходить для лабіринто-подібних рівнів, проте може створювати тупики або непридатні для гри ділянки, тому часто потребує додаткових модифікацій[7].

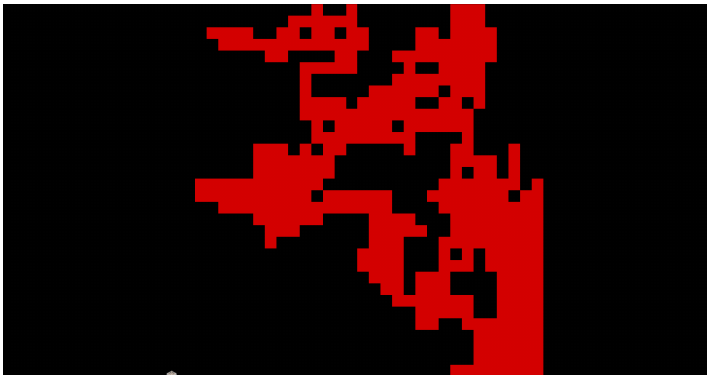

Рисунок 1.4 – Зразок карти, створеної за допомогою алгоритму випадкового блукання

Побудова рівнів на основі графа кімнат (Room Graph) передбачає створення логічної структури, де вузлами слугують кімнати, а ребрами – зв'язки між ними. Такий підхід дозволяє визначити сценарій проходження, розташування ключових подій, альтернативні шляхи та розгалуження. У roguelike іграх це дає можливість реалізувати складніші механіки: системи вибору, логічні пастки, сюжетні розгалуження. Перевага методу – контроль над ігровою логікою, а недолік – складність просторової реалізації без конфліктів у сітці [7].

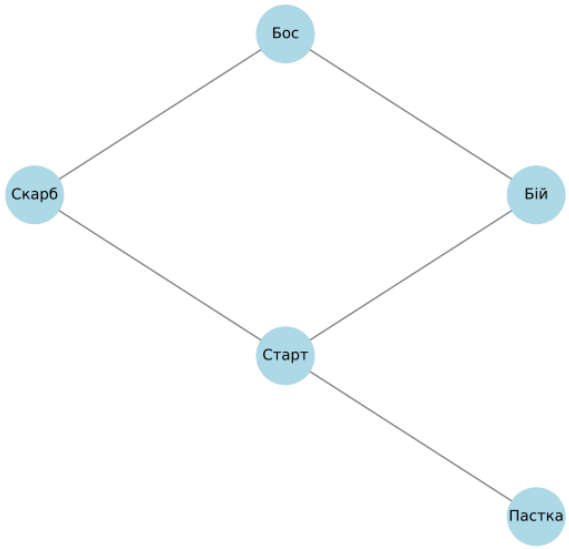

Рисунок 1.5 – Граф кімнат

Алгоритм пошуку в глибину (DFS, Depth-First Search) часто використовується як один із базових підходів до процедурної генерації рівнів у формі графа кімнат. Основна ідея полягає в поступовому додаванні нових кімнат до ігрового простору через рекурсивне заглиблення: починаючи зі стартової точки, алгоритм обирає один із доступних напрямків (наприклад, вгору, вниз, ліво, вправо) і створює нову кімнату, після чого виконує той самий процес із неї. Коли всі сусіди заповнені або досягнуто заданої кількості кімнат, алгоритм повертається до попередньої кімнати (бектрекінг) і продовжує генерацію.

Такий підхід дає змогу побудувати зв'язний граф приміщень, забезпечуючи при цьому простоту реалізації та керовану варіативність структури рівня. Алгоритм легко комбінується з іншими методами (наприклад, шаблонною генерацією вмісту кімнат або класифікацією за типами: бойова, бонусна, босова).

Основними перевагами DFS є:

– гарантована зв'язність кімнат, що виключає появу ізольованих зон;

– простота контролю складності рівня шляхом обмеження глибини або кількості кроків;

– підтримка генерації на основі seed, що дозволяє створювати детерміновані рівні.

До потенційних недоліків можна віднести: нелінійність розташування кімнат у сітці, яка іноді призводить до труднощів з відображенням карти, та невисоку ступінь симетрії у згенерованій структурі без додаткової постобробки.

Шумові функції, зокрема перлін-шум, генерують псевдовипадкові, але плавно змінювані дані, що дозволяє формувати природні ландшафти. У roguelike іграх цей метод рідко використовується для безпосереднього формування кімнат, але часто застосовується для створення додаткових шарів

карти – рівнів складності, типів біомів, кліматичних умов. Особливо ефективний перлін-шум у відкритих світах із зональним поділом.

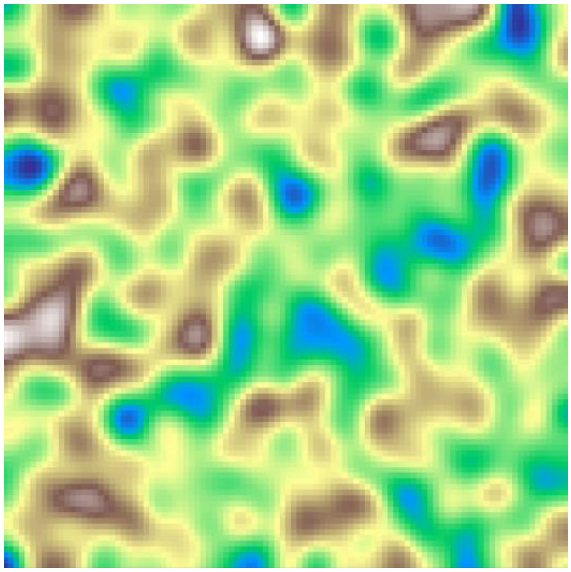

Рисунок 1.6 – Генерація зон за допомогою перлін-шуму

Важливо усвідомлювати, що в реальних проектах жоден з описаних алгоритмів не застосовується окремо. Вдалі ігри зазвичай поєднують сильні риси декількох методів. Наприклад, BSP може визначати загальну структуру рівня, а клітинкові автомати – деталізувати внутрішнє наповнення кімнат. Таке комбінування забезпечує як логічну зв'язність, так і візуальну різноманітність.

Отже, алгоритми процедурної генерації є базою , для розробки рівнів у roguelike іграх. Їх вибір та втілення суттєво впливають на якість проекту, визначаючи варіативність, складність і динамічність ігрового процесу.

У процесі реалізації гри в жанрі roguelike планується застосувати метод детермінованої процедурної генерації кімнат на основі фіксованих seed-значень та алгоритму пошуку в глибину (DFS). Для оцінки ефективності обраного методу доцільно здійснити порівняльний аналіз з класичними алгоритмами генерації рівнів, такими як BSP (Binary Space Partitioning), Perlin Noise, та іншими підходами, що застосовуються у розробці процедурних світів.

Порівняльні критерії:

1. Реіграбельність – здатність створювати унікальний ігровий досвід при кожному проходженні;
2. Збалансованість рівнів – логічне розміщення кімнат, ворогів і нагород;
3. Глибина геймплею – різноманіття взаємодій, просторів та тактичних можливостей гравця.

Таблиця 1.1 – Порівняльний аналіз методів процедурної генерації

Метод генерації	**Реіграбельність**	**Збалансованість**	**Глибина геймплею**	**Коментар**
DFS + Seed	Висока	Висока	Середня	Простий у реалізації, забезпечує стабільну структуру та хорошу зв’язність
BSP	Висока	Середня	Висока	Широко застосовується в класичних roguelike (напр. Dwarf Fortress), дає змогу будувати лабіринти, зони та зали
Perlin Noise	Середня	Низька	Висока	Ефективний для плавних природних форм (ландшафти, біоми), проте менш придатний для чітких кімнат
Cellular Automata	Середня	Середня	Середня	Часто використовується для генерації печерних систем, складніший контроль над структурою
Prefab-based	Низька	Висока	Низька	Ручна робота з шаблонами дозволяє створити чіткі структури, однак втрачається варіативність
Wave Function Collapse	Низька	Висока	Висока	Потужний, але складний у реалізації, потребує налаштування правил

Обраний підхід на основі DFS-обходу з фіксованими seed-значеннями забезпечує високу реіграбельність і передбачувану збалансовану структуру, що є достатнім для гри середньої складності в жанрі roguelike. У порівнянні з BSP, наш метод є простішим у реалізації, проте менш гнучким для створення розгалужених ігрових просторів.

Перехід до більш складних алгоритмів, таких як Wave Function Collapse або BSP, доцільний на етапі масштабування гри або під час створення великих та нелінійних локацій з багаторівневою глибиною.

1.3 Аналіз сучасних ігрових засобів та ігор жанру roguelike

Оглядаючи ринок відео ігор за допомогою платформи Steam можна зафіксувати закономірність, що дедалі частіше інді проекти потрапляють у топи продажів і значна кількість це саме ігри жнанру roguelike .

На основі відкритих даних Steam, а також аналізу ключових проектів за кількістю рецензій, активності спільноти та присутності у стримінгових сервісах, можна виокремити низку ігор, що стали культовими в межах жанру. Розглянемо найпопулярніші з них які впливають не тільки на розвиток жанру roguelike а й в цілому інді розробки.

The Binding of Isaac: Rebirth – це одна з найвідоміших ігор жанру ключовим чинником, що забезпечив грі культовий статус, стала її несподівано глибока та гнучка механіка, прихована під візуально спрощеним піксельним дизайном. Незважаючи на зовнішню простоту, The Binding of Isaac: Rebirth поєднує в собі елементи dungeon crawler, twin-stick shooter та жанр roguelike, який робить її доступною для новачків завдяки своїй популярності, але складною та винагороджувальною для досвідчених гравців. Вона стала прикладом того, як авторський підхід до жанру, дизайну та механіки може не лише конкурувати з великими студіями, а й формувати власну спільноту [8].

Рисунок 1.7 – Вирізка з гймплею гри The Binding of Isaac: Rebirth

Одним із головних інноваційних рішень у грі є система процедурної генерації рівнів. Кожен рівень створюється динамічно, шляхом випадкового

формування графа кімнат, у якому кожна вершина — це заздалегідь створена кімната з певною функціональністю (нагорода, магазин, бос, секретка тощо). Алгоритм генерації забезпечує структурну цілісність рівня — гарантуючи зв'язність усіх зон і поступове ускладнення на кожному поверсі. Кімнати обираються з великого набору шаблонів, які доповнюються випадковими ворогами, предметами, пастками чи ефектами. Завдяки цьому кожне проходження стає унікальним, а кількість можливих комбінацій перевищує мільйони варіантів.

Важливо зазначити, що гра не обмежується лише генерацією структури карт. Випадковості піддаються також вороги, предмети, покращення, модифікатори поведінки персонажа. Усі ці компоненти взаємодіють між собою — створюючи ефект "ігрової хімії", де кожна дія може мати несподівані наслідки. Баланс гри досягається не через жорстку складність, а через високий ступінь ризику та винагороди. Наприклад, гравець може обрати ризиковану гілку проходження, яка призведе до отримання сильніших предметів, або ж залишитися на безпечному маршруті, але з меншими шансами на перемогу. Велика роль у цьому належить системі торгів із «дияволом», яка дозволяє обмінювати частину здоров'я персонажа на потужні посилення. Такий підхід стимулює тактичне мислення та постійне ухвалення рішень у стресових ситуаціях. Ще однією причиною культовості The Binding of Isaac є її тематична сміливість. Сюжет, що побудований навколо біблійних мотивів, дитячих травм, релігійного фанатизму та абсурду, створює глибоко символічний світ, у якому гравець кожного разу занурюється глибше — як у прямому, так і в метафоричному сенсі.

Розглянему іншу гру Noita це неймовірно технічно складна гра тому, що вона моделює увесь світ як систему, в якій кожен піксель має фізичну природу. Головною особливістю Noita є повна симуляція матеріального середовища: вода, лава, газ, слиз, кислотні речовини, вибухові матеріали, електрика, поліморфін та інші субстанції поводяться згідно з власними фізичними властивостями [9].

Рисунок 1.8 – Приклад фізичної взаємодії електрики з водою у Noita

Світ гри піддається постійним фізичним трансформаціям, і навіть незначні дії гравця або випадкові взаємодії між елементами можуть призвести до непередбачуваних наслідків. Це забезпечує високий рівень варіативності, навіть при повторному проходженні одних і тих самих ділянок.

На відміну від багатьох інших проектів, де процедурна генерація обмежується створенням карти або набором кімнат, у Noita вся логіка світу побудована навколо динамічної, детермінованої генерації, яка забезпечує повну відтворюваність та варіативність.

Світ гри поділений на біоми, кожен з яких має власний набір параметрів: тип ґрунту, матеріали, вороги, об'єкти, освітлення, розміри проходів. Наприклад, перші рівні (Mines, Coal Pits) мають сітчасту структуру з великою кількістю матеріалів, які легко піддаються руйнуванню, тоді як глибші шари (Snowy Depths, Hiisi Base) містять більш тверді породи, складніші маршрути і небезпечні елементи оточення (електричні кабелі, токсичні басейни, вибухові пристрої тощо).

Генерація відбувається на основі перлиноподібного псевдовипадкового шуму (Perlin-like noise) із застосуванням дизайнерських правил на рівні шаблонів та симетрій. Важливою особливістю є те, що генерація є детермінованою: для кожного конкретного “сіду” (початкового значення генерації) світ буде однаковим, що дозволяє гравцям та спільноті повторювати, досліджувати або порівнювати проходження при фіксованих умовах.

Кожна клітинка карти представлена вокселем з власним фізичним матеріалом. У грі реалізовано понад 100 типів матеріалів — від звичайного каменю до лави, газів, кислот, магічних слизів і крові. Усі матеріали мають властивості: щільність, горючість, провідність, в'язкість, реактивність. При процедурному розміщенні елементів гри, генератор враховує можливі взаємодії: наприклад, поруч із джерелами вогню не розміщуються легкозаймисті гази, якщо це не передбачено геймдизайнерами як пастка.

Таким чином, процедурна генерація в Noita — це не лише топологія, а і взаємодія речовин. Гравець може обрушити стелю, щоб залити ворогів кислотою, спровокувати ланцюгову реакцію або вбити себе випадковим електричним імпульсом через мокрий одяг. Це створює постійні складності де ви помираєте не тільки від складних ворогів а й від своєї жадності коли намагаєтесь підібрати золотий самородок і стаєте в полум’я яке і вбиває вас.

Система заклинань у грі це ще один чиник чому гра настільки реіграбільна та популярна. У Noita реалізовано одну з найглибших і найгнучкіших систем заклинань у жанрі roguelike. Загалом у грі 424 унікальних заклинань які потрібно комбінувати між собою. Якщо взяти посередній посох в грі у якому 12 слотів то можемо вирухувати яка кількість комбінацій з повтореннями буде $424^{12} \approx 4.66 * 10^{31}$ це робить систему жезлів Noita однією з найгнучкіших заклинальних систем у відеоіграх, з нескінченним простором для експериментів, побудови алгоритмів та дослідженням внутрішньої логіки [9].

Розглянемо ще одру гру яка постійно посягає найвищі позиції у крамниці Steam ця гра це Dead Cells — приклад гри, яка поєднує жанри roguelike та metroidvania. У грі реалізовано динамічний платформер з бойовою системою в реальному часі, процедурною генерацією рівнів на основі шаблонів, а також системою перманентного покращення певних характеристик гравця між проходженнями. Особливістю гри є плавний контроль персонажа, висока анімаційна якість і розгалуження маршрутів під час проходження.

Рисунок 1.9 – Вирізка з геймлею Dead Cells

Dead Cells використано складніший підхід до генерації рівнів на основі предзаданих шаблонів із подальшим випадковим компонуванням блоків. Структура рівня генерується як послідовність сегментів, які підбираються відповідно до заданих правил. Розміщення ворогів, предметів, платформ та пасток реалізується динамічно в межах кожного блоку. Можна сказати, що гра використовує змішаний метод BSP та шаблонної генерації. Алгоритм поділяє карту на області, після чого кожна область наповнюється контентом із випадковими варіаціями. Застосовується також рівнева генерація — на кожному новому рівні зростає складність ворогів, а доступ до нових гілок відкривається поступово [10].

1.4 Висновок до розділу 1

У ході проведеного аналізу було встановлено, що жанр roguelike продовжує активно розвиватися та зберігати свою актуальність завдяки унікальному поєднанню геймплейних характеристик — таких як процедурна генерація, перманентна смерть, висока складність, динамічне оновлення середовища та глибока система варіацій. Саме ці риси визначають ключові особливості жанру та впливають на його популярність серед гравців різного рівня підготовки.

Було проаналізовано сучасний стан індустрії розробки відеоігор та виявлено основні проблеми, які відкривають можливості для інді-проектів. Зокрема, зростаюча комерціалізація великих (AAA) проектів, стандартизація геймплею та втрата креативності сприяли формуванню сталого інтересу до інноваційних рішень, притаманних іграм незалежних розробників. Особливої уваги заслуговують roguelike-проекти, які завдяки процедурній генерації здатні забезпечити унікальний досвід для кожного користувача навіть без значних витрат на контент [11].

Було проведено систематичний аналіз алгоритмів, що застосовуються для процедурної генерації ігрового середовища. Розглянуто ключові підходи: BSP (Binary Space Partitioning), клітинкові автомати, метод випадкової ходи, графи кімнат(DFS) та генерацію за допомогою шумових функцій. Встановлено, що ефективне використання процедурної генерації передбачає не лише вибір базового алгоритму, а й адаптацію його до потреб конкретної гри: забезпечення зв’язності рівня, контролю за складністю, варіативності луту, та дотримання логіки світу гри.

Також проведено порівняння з класичними алгоритмами, такими як BSP та Perlin Noise, продемонстрували перевагу обраного методу в аспектах реіграбельності та керованості складністю.

Здійснено огляд найуспішніших сучасних ігор жанру roguelike, таких як The Binding of Isaac: Rebirth, Noita та Dead Cells, з особливою увагою до їхньої архітектури, процедурної генерації, бойових систем і технологічної реалізації. З’ясовано, що попри жанрову спільність, кожна з цих ігор реалізує власну модель процедурної генерації: від модульного компонування (Dead Cells), до симуляції кожного пікселя світу (Noita).

Отже, за результатами аналізу можна зробити висновок, що ефективна реалізація roguelike-гри потребує комплексного підходу до формування геймплейної структури, гнучкої системи генерації контенту, балансу між випадковістю та дизайном, а також відповідного вибору інструментів для програмної реалізації.

2. ПРОЕКТУВАННЯ ROGUELIKE ГРИ З ПРОЦЕДУРНОЮ ГЕНЕРАЦІЄЮ РІВНІВ

2.1 Проектування архітектури roguelike гри з процедурною генерацією

Побудова архітектури є ключовим етапом під час підготовки до реалізації ігрового проекту, оскільки саме на цьому рівні визначається, як окремі підсистеми будуть функціонально взаємодіяти між собою. У межах дослідження передбачається створення гри жанру roguelike, в основі якої закладено принцип процедурної генерації рівнів, та адаптивну структуру взаємодії з ігровим середовищем [12]. Запланована архітектура має базуватися на компонентно-орієнтованій моделі, яку підтримує рушій Unity [13]. Застосування мови C# дозволить побудувати гнучку систему класів, які відповідатимуть за окремі аспекти функціонування гри. Очікується, що основним координуючим компонентом стане GameManager — модуль, який відповідатиме за ініціалізацію гри, завантаження рівнів, контроль основного стану гри та інтеграцію з іншими системами, зокрема інтерфейсом користувача, генератором рівнів і системою збереження.

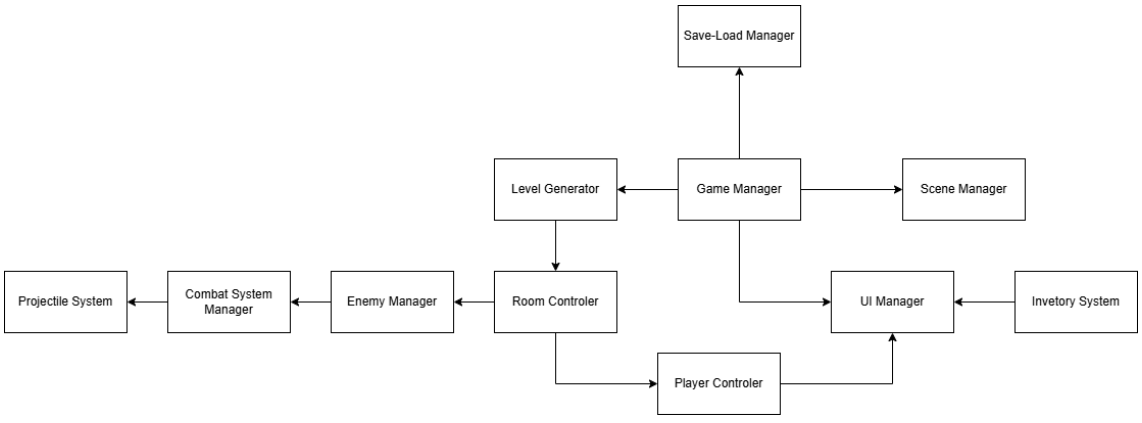

Рисунок 2.1 – Загальна блок схема архітектури гри

Передбачається, що ігровий простір формуватиметься процедурно, тобто матиме унікальну структуру кожного разу під час запуску. Для цього

планується розробити модуль генерації рівнів, що створюватиме сітку з кімнатами, з'єднаними логічними переходами. Ці кімнати матимуть різні функції: бойові зони, приміщення з босом, секретні кімнати тощо. Керування їхнім вмістом і сценаріями буде зосереджено в компоненті RoomController, який ініціюватиме внутрішні події залежно від контексту.

У грі передбачено наявність модуля PlayerController, що має забезпечити логіку управління персонажем. У межах його функціоналу очікується реалізація переміщення, стрільби, ухилення, збору предметів, взаємодії з середовищем. Цей компонент також повинен координуватися з іншими підсистемами — бойовою логікою, інвентарем, інтерфейсом. Бойова система буде організована на основі окремого модуля, умовно позначеного як CombatSystem. Планується, що саме він відповідатиме за створення снарядів, обробку їхньої траєкторії, зіткнення з об’єктами, а також за розрахунок завданої шкоди. Для оптимізації роботи буде передбачено механізм повторного використання об’єктів (пулінг), що дозволить зменшити навантаження на систему під час масових боїв.

Окрему роль у структурі системи відіграватиме EnemyManager — диспетчер, який опікуватиметься створенням, оновленням та контролем супротивників. Передбачається впровадження поведінкової логіки на основі FSM (скінченний автомат), що дозволить задавати супротивникам динамічні сценарії (переслідування, стрільба, уникнення, тощо). Це забезпечить варіативність бою та підвищить загальну динаміку гри.

Для зберігання та керування зібраними предметами буде реалізовано інвентарну систему. Вона передбачатиме класифікацію предметів за типами (пасивні, активні), а також матиме інтерфейс, через який гравець зможе активувати або змінювати обладнання. Модуль InventorySystem має бути тісно інтегрований із бойовою логікою та інтерфейсом користувача. Користувацький інтерфейс, представлений у вигляді HUD, індикаторів стану, шкал та інформативних панелей, проектується як окрема підсистема. Передбачається, що оновлення UI відбуватиметься реактивно, на основі сигналів, що надходять

від інших модулів, зокрема PlayerController та InventorySystem. Для цього буде реалізовано відповідний UI Manager, який слідкуватиме за оновленням візуальних елементів у режимі реального часу.

У межах проекту також заплановано створення системи збереження, яка зможе фіксувати ключові дані про стан гравця, інвентар, розміщення в кімнатах та інші параметри, критичні для відновлення гри. Система буде працювати на основі серіалізації даних у форматі, сумісному з файловою системою, й дозволятиме зчитування під час запуску гри.

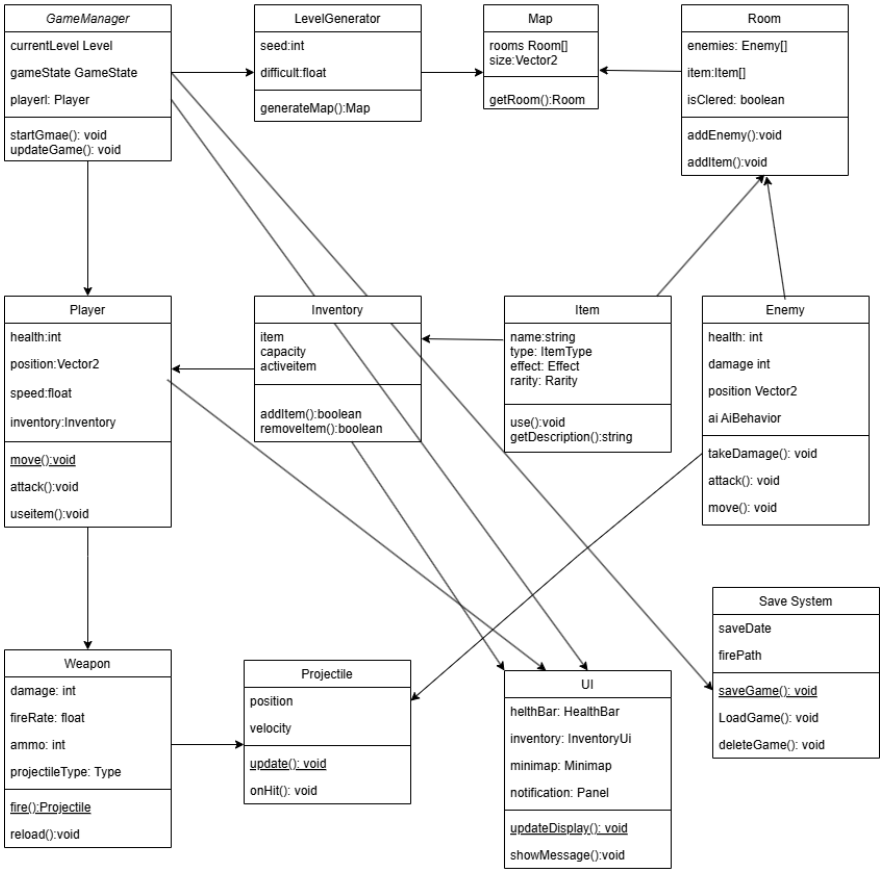

Рисунок 2.2 – UML діаграма класів та їх взаємодія

Таким чином, сформована архітектурна модель гри передбачає багаторівневу структуру з чітко окресленими модулями. Вона має забезпечити незалежність функціональних блоків, підтримку процедурної генерації, можливість масштабування функціоналу та спрощене тестування.

2.2 Розробка алгоритму процедурної генерації ігрових рівнів

Процедурна генерація є одним з ключових механізмів реалізації гри у жанрі roguelike, оскільки саме вона забезпечує динамічність, реіграбельність і значну варіативність, дозволяючи формувати унікальні рівні при кожному запуску гри. У межах даного проекту передбачається реалізація алгоритму процедурної генерації, що базується на побудові зв’язного графа кімнат із подальшим наповненням кожної з них відповідно до заданих типів і параметрів складності.

Базова ідея полягає в представленні рівня у вигляді ненаправленого графа, де вузлами виступають кімнати, а ребрами — переходи між ними. Такий підхід забезпечує гарантовану зв’язність рівня, що критично важливо для ігрової логіки. Графова структура також дозволяє здійснювати ефективний контроль над кількістю кімнат, їхнім взаємним розташуванням, можливістю обходу та пошуку альтернативних маршрутів.

Для побудови графа передбачається використання модифікованого алгоритму пошуку в глибину (Depth-First Search, DFS). Генерація починатиметься зі стартової кімнати, від якої по черзі створюватимуться нові кімнати у випадкових напрямках. Алгоритм зберігатиме інформацію про вже створені координати, що дозволить уникнути їх дублювання. Після досягнення заданої кількості кімнат, побудова графа припиняється [14].

Наступним етапом стане класифікація кімнат за функціональним призначенням. Кімната, що знаходиться на найбільшій відстані від старту, буде обрана як кімната з босом. Декілька інших кімнат отримають статуси бонусних, магазинів, пасток або секретних зон. Решта буде маркована як звичайні бойові кімнати. Така схема забезпечить гравцю як основну гілку проходження, так і можливість побічного дослідження.

Усередині кожної кімнати планується використовувати шаблонну генерацію вмісту. Кожен шаблон — це заздалегідь підготовлена конфігурація розташування ворогів, об'єктів оточення, укриттів та предметів. Обираючи

шаблон, система враховуватиме тип кімнати, її складність та відстань від початкової точки. Такий підхід дозволить зберегти керованість ігрового балансу, не жертвуючи при цьому процедурною варіативністю.

Для посилення відтворюваності буде реалізовано підтримку генерації на основі seed — числового параметра, що задає початковий стан генератора випадкових чисел. Це дозволить тестувати специфічні конфігурації рівнів, створювати щоденні виклики або змагальні режими з однаковим розміщенням кімнат для всіх гравців.

На завершальному етапі генерації система перевірятиме коректність створеного рівня: зв’язність усіх кімнат, наявність старту та кінця, можливість логічного проходження та відсутність конфліктів між об’єктами. У разі помилок буде застосовано повторне формування з використанням іншого seed або обмеженої реконфігурації частини графа.

Також доцільно розглянути альтернативні підходи до процедурної генерації. Зокрема, метод BSP (Binary Space Partitioning) дозволяє ділити простір на прямокутні області з подальшим створенням кімнат у межах них. Цей підхід добре підходить для dungeon-like структур. Метод Perlin Noise використовується для генерації природних ландшафтів або лабіринтів із плавними переходами. Алгоритми Клітинного автомату ефективні для створення печероподібних рівнів або органічних структур. Проте у контексті гри з чіткою системою переходів, контрольованим дизайном і прогресивною складністю саме графовий підхід на основі DFS був визнаний найбільш доцільним. Отже генератор враховуватиме наповнення кімнат залежно від їхнього типу та розташування у графі. Стартові кімнати будуть вільними від ворогів і міститимуть підказки або базові ресурси, тоді як центральні вузли можуть мати посилених противників або квестові елементи. Кінцева кімната, що веде до наступного рівня або боса, буде розміщуватись на максимальній відстані за графовою метрикою, що забезпечить природне зростання складності.

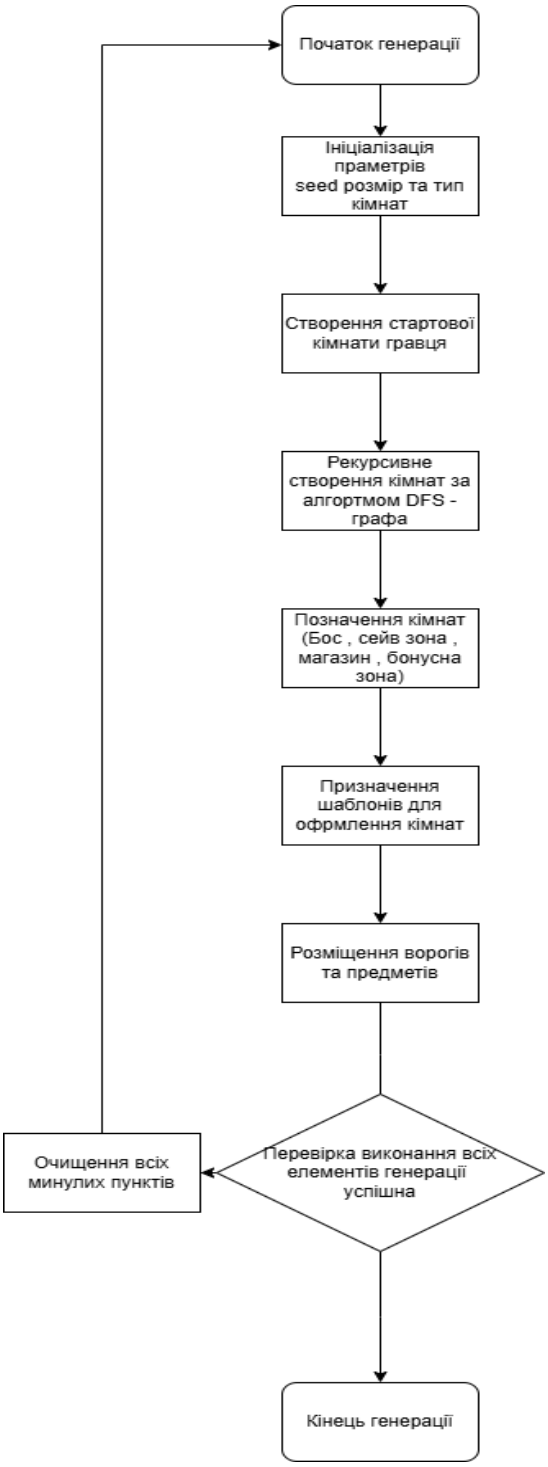

Рисунок 2.3 – Блок схема генерації рівнів

Таким чином, реалізована структура процедурної генерації забезпечить необхідний рівень унікальності кожного проходження, дозволить гнучко налаштовувати складність та різноманітність кімнат, а також забезпечить технічну зручність для подальшої розробки й тестування.

2.3 Розробка модулів керування

Одним із ключових елементів будь-якої гри жанру roguelike є ефективно реалізована бойова система та відповідна їй логіка управління гравцем і супротивниками. У межах розробки гри особлива увага приділяється саме модульності цих компонентів, що забезпечує гнучкість у реалізації, підтримку розширень та зручність тестування.

У модулі керування гравцем передбачається реалізація таких основних функціональних можливостей: переміщення у двох площинах (використовуючи фізичну або просту координатну систему), стрільба та зміна напрямку атаки, взаємодія з об'єктами (використання предметів, відкриття дверей), а також облік стану здоров’я, тимчасових ефектів та смерті. Логіка управління буде реалізована як окремий клас, що взаємодіє з фізичним тілом гравця через події, дозволяючи легко від’єднувати або доповнювати поведінку.

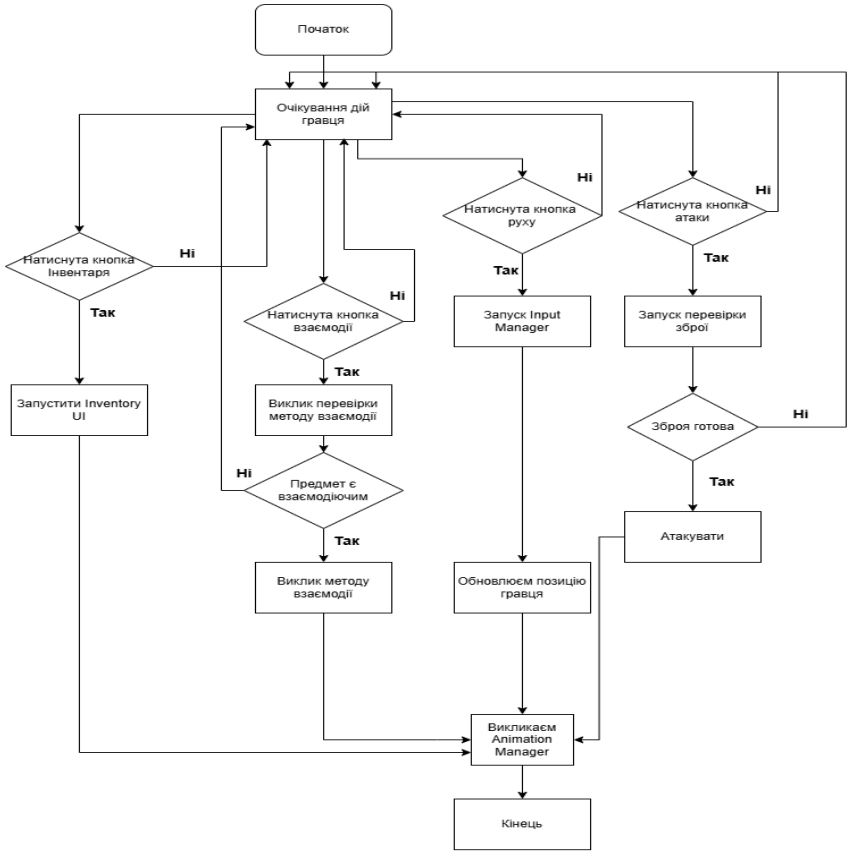

Рисунок 2.4 – Блок-схема логіки модуля управління гравцем

На наведеній блок-схемі зображено основну послідовність дій, що відбуваються під час обробки введення від користувача в модулі керування гравцем. Після старту гри система переходить у стан очікування дій від гравця. На цьому етапі перевіряється натискання основних кнопок: відкриття інвентаря, взаємодії з об'єктами, пересування або атаки. У випадку натискання клавіші для відкриття інвентаря запускається відповідний інтерфейс (Inventory UI). Якщо натиснута кнопка взаємодії, викликається метод перевірки на наявність інтерактивного об'єкта в напрямку руху гравця. У разі підтвердження – викликається відповідна взаємодія з об'єктом. При отриманні інпуту на рух ініціюється виклик менеджера введення, після чого розраховується нова позиція гравця. Аналогічно, при натисканні клавіші атаки виконується перевірка готовності зброї, і якщо атакувальна дія дозволена, викликається метод атаки. Незалежно від типу дії, по завершенню відбувається оновлення анімаційного стану через виклик Animation Manager, після чого потік логіки повертається до очікування нових дій від гравця.

Контроль ворогів передбачає розподіл поведінки на три основні групи: патрулювання, переслідування та атака. Кожен тип ворога матиме окремий набір параметрів агресії, зони виявлення та стилю атаки (дальня, ближня чи вміння). Для реалізації поведінки буде використано простий машинний підхід (FSM), де кожен ворог має набір станів і переходів між ними в залежності від подій у середовищі. Наприклад, ворог переходить зі стану патрулювання у стан переслідування при вході гравця в радіус виявлення. Важливим компонентом бойової системи є система снарядів (projectile system), яка реалізовує логіку створення об'єктів, що рухаються по траєкторії, взаємодіють із ворогами або середовищем, викликають вибухи, статусні ефекти тощо. Кожен снаряд має набір властивостей: швидкість, радіус ураження, тип ефекту. Створення снаряда відбувається при виклику атаки з боку гравця або ворога, і далі його рух обробляється через фізику рушія.

Обробка влучань передбачає врахування зіткнень між снарядами та коллайдерами ворогів, при цьому можуть викликатись події типу "Шкода",

"Відштовхуання", "Ефект статусу". Завдяки цьому реалізується універсальна модель ураження, яку можна масштабувати під різні типи зброї та ворогів. Система обробки шкоди підтримує множинні типи ефектів: безпосередню втрату здоров’я, сповільнення, отруєння, підпал, тимчасову дезорієнтацію.

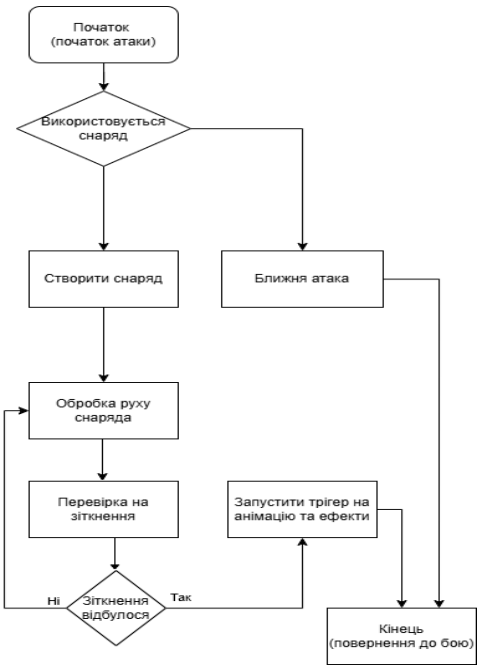

Рисунок 2.5 – Блок схема взаємодії Projectile System

Додатково реалізується механізм відштовхування для гравця та ворогів, що забезпечує візуальну динаміку бою та дозволяє створювати комбінації зі снарядами і пастками.

Архітектурно ці модулі реалізовані як незалежні компоненти, що прикріплюються до об'єктів на сцені через систему компонентів і подій. Такий підхід забезпечує гнучкість і масштабованість. Наприклад, при створенні нового типу ворога достатньо наслідувати базовий клас Enemy та перевизначити методи для його поведінки.

У майбутньому доцільно розширити систему бою шляхом додавання спеціальних типів атак, динамічних елементів арени (перешкоди, реактивні стіни), а також впровадити кооперативний режим, де кожен гравець керується окремим модулем логіки, синхронізованим через мережеву систему.

2.4 Висновок до розділу 2

У результаті виконання проектної частини було розроблено загальну архітектуру програмного модуля гри у жанрі 2D roguelike з процедурною генерацією рівнів. Проведено структурний аналіз основних складових ігрової системи, сформовано логіку їхньої взаємодії та визначено підходи до реалізації окремих компонентів, зокрема генератора рівнів, бойової системи та модуля управління гравцем.

Було детально обґрунтовано вибір ігрового рушія, мов програмування та засобів розробки. Побудовано UML-діаграму класів, яка наочно демонструє структуру об’єктів та їхні взаємозв’язки. Також сформовано загальну архітектуру гри, яка забезпечує модульність і масштабованість.

Розглянуто підхід до реалізації алгоритму процедурної генерації рівнів. Було визначено базову модель побудови рівня у вигляді графа кімнат із подальшим маркуванням функціональних зон.

Обрано метод модифікованого пошуку в глибину (DFS) як основу для нарощування структури мапи, з урахуванням обмежень кількості кімнат, координат та взаємозв’язку між ними. Додатково було розглянуто альтернативні алгоритми генерації та аргументовано доцільність використання саме графового підходу для даної гри.

Спроектовано логіку взаємодії гравця, ворогів та бойових об’єктів. Деталізовано основні модулі керування гравцем (PlayerController), поведінки ворогів (з використанням FSM-моделі) та обробки бойових ситуацій (CombatManager).

Також було побудовано блок-схему обробки дій гравця, а та описано принципи архітектурного розподілу обов’язків між класами, що забезпечує простоту розширення та модифікації гри в майбутньому.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ ГРИ В ЖАНРІ ROGUELIKE

3.1 Обгрунтування вибору інструментів технічної реалізації

Для реалізації 2D Roguelike гри з елементами процедурної генерації, бою та дослідження було обрано мову програмування C# у поєднанні з ігровим рушієм Unity. Це рішення було продиктоване багатьма факторами, включаючи специфіку цільової платформи (Windows / Android / WebGL), вимоги до реалізації інтерактивного ігрового процесу, необхідність гнучкої модуляції гри, роботи з 2D-графікою, фізикою та анімацією, а також забезпечення високої продуктивності при мінімальному споживанні системних ресурсів.

При виборі технології враховувалися такі критерії: підтримка об'єктної моделі, наявність розвиненого візуального інтерфейсу редагування, легкість масштабування гри, продуктивність з тайловими картами, спрайтами та колізіями, а також можливість створення адаптивного інтерфейсу користувача для взаємодії з користувачем [14].

В процесі огляду порівнювалися два основні варіанти: Unity (C#) та Pygame (Python). Порівняння з альтернативою: Python (Pygame)

Python з бібліотекою Pygame також був розглянутий як потенційна платформа для розробки. Цей варіант вирізняється простотою синтаксису та низьким порогом входу.

Проте:

1. Відсутність вбудованих компонентів для Tilemap, фізики, UI, анімацій значно ускладнює реалізацію ігрової логіки.
2. Графічні можливості обмежені порівняно з Unity.
3. Менша продуктивність при складній логіці (вороги, снаряди, генерація кімнат).
4. Немає підтримки редактора сцен, що змушує створювати сцени вручну у коді.

За результатами порівняння інструментів, основною технологією для реалізації гри було обрано Unity з C#. Таке поєднання забезпечує повну підтримку всіх необхідних аспектів проекту: процедурної генерації кімнат, управління боєм, реалізації ШІ противника, обробки подій та візуалізації піксельної графіки. Завдяки широким можливостям Unity стало можливим реалізувати функціональність гри Roguelike на сучасному рівні, з гнучкою структурою, високою продуктивністю та привабливим візуальним дизайном.

3.2 Програмна реалізація гри в жанрі Roguelike

На етапі програмної реалізації ігрові модулі розроблялися поетапно відповідно до заздалегідь визначеної архітектури. Гра була реалізована в жанрі Roguelike, який включає в себе процедурну генерацію кімнат, елементи випадковості, бойову систему, прогресію складності та переміщення між кімнатами. Програмна реалізація здійснювалася за допомогою середовища розробки Unity та мови програмування C#.

Центральним елементом архітектури є генератор рівнів “LevelGenerator.cs” , який динамічно створює кімнати, об’єднані між собою у вигляді графа. Вихідною точкою є стартова кімната, після чого за алгоритмом пошуку в глибину (DFS) додаються нові кімнати на сітці координат. Для кожної з них визначається розташування відносно інших кімнат, а також наявність з’єднань у вигляді дверей або порталів.

1. Генератор рівня “LevelGenerator.cs” зберігає інформацію про позиції, з’єднання та типи кімнат (звичайна, кімната з босом). На основі унікального значення “seed” створюється відтворювана карта, що забезпечує як варіативність, так і можливість повторного проходження одного й того ж рівня. Для кожної кімнати використовуються тайли, які розміщуються за допомогою Tilemap-системи Unity у файлі “TileRoomGenerator.cs”, включаючи стіни, підлогу, декоративні елементи та двері. Після знищення всіх ворогів у кімнаті з’являється портал, що дає змогу гравцеві перейти далі.

2. Модуль управління гравцем “PlayerController.cs”

У модулі PlayerController реалізовано рух гравця, взаємодію з об’єктами та базову логіку атаки. Гравець має змогу переміщуватися у будь-якому напрямку, використовуючи клавіші WASD, а також атакувати ворогів ближньою або дальньою атакою в залежності від відстані.

Система бою реалізована у файлі “PlayerCombat.cs”, де передбачено вибір типу атаки, перевірку на наявність ворога у зоні дії, а також створення об'єкта снаряду “Projectile.cs” у разі дальньої атаки.

Кількість здоровя , та система смерті гравця реалізована у “PlayerHealth”

3. Модуль ШІ ворогів “Enemy.cs” та “EnemyFSM.cs ”

Для керування поведінкою ворогів використовується Finite State Machine (FSM)**.** Кожен ворог має набір станів, зокрема: патрулювання, переслідування гравця, атака, відступ та смерть. Реалізація FSM дозволяє легко розширювати логіку ворогів та додавати нові типи, зокрема ближнього бою “EnemyFSM.cs” та дальнього бою “RangedEnemyFSM.cs”.

Вороги мають власні характеристики здоров’я, шкоди та дальності виявлення гравця. Після смерті ворога викликається відповідна подія, яка сигналізує про його знищення, що дозволяє, наприклад, активувати портал для переходу до наступної кімнати “EnemyRoomController.cs”.

4. Модуль порталу “Portal.cs”

Після знищення всіх ворогів у кімнаті в центрі з’являється портал, що дозволяє гравцеві перейти до наступної доступної кімнати. Телепортація здійснюється шляхом переміщення гравця до позиції, пов’язаної із сусідньою кімнатою у генераторі рівнів.

Портал реалізовано як об’єкт із компонентом BoxCollider2D з увімкненим IsTrigger. У скрипті “Portal.cs” перевіряється колізія з гравцем та ініціюється переміщення до наступної кімнати. Таким чином реалізується логіка прогресивного проходження.

Ключовим елементом структури гри roguelike є процедурна генерація рівня. Файл LevelGenerator.cs реалізує модифікований алгоритм пошуку в

глибину (DFS) для побудови ігрового світу, який забезпечує створення взаємопов’язаних кімнат у псевдовипадковому порядку.

```
void Start()
{
    Random.InitState(seed);
    GenerateLevel();}
```

Початок роботи генератора ініціюється методом Start(), де встановлюється початкове значення генератора випадкових чисел та викликається головна функція побудови рівня — GenerateLevel().

```
void GenerateRoomDFS(Room current)
```

Для процедурної генерації використовується рекурсивна функція GenerateRoomDFS, яка у випадковому порядку обирає напрямки (вгору, вниз, вліво, вправо), створюючи нові кімнати в межах допустимої кількості maxRooms. Результатом є орієнтований граф з кімнатами як вершинами та дверима як ребрами.

```
public class Room
{
    public Vector2Int gridPosition;
    public bool visited = false;
    public Dictionary<Direction, Room> connections =
new Dictionary<Direction, Room>();
}
```

Кожна кімната описується через координати на сітці (gridPosition), ознаку відвідування та словник з’єднань з іншими кімнатами (connections). Це дозволяє легко побудувати карту рівня та відслідковувати логічні зв’язки між приміщеннями.

```
TileRoomGenerator generator =
roomGO.GetComponent<TileRoomGenerator>();
if (generator != null)
```

```
    {
        int roomSeed = seed + room.gridPosition.x * 73856093 + room.gridPosition.y * 19349663;
        generator.Generate(roomSeed);
    }
```

Для кожної створеної кімнати викликається компонент TileRoomGenerator, який генерує внутрішнє наповнення кімнати (стіни, підлога, перешкоди) на основі окремого seed. Це забезпечує унікальність структури кожного приміщення навіть при однаковому типі кімнати.

```
    foreach (var conn in room.connections)
    {
        string doorName = conn.Key switch
        {
            Direction.Up => "DoorTop",
            Direction.Down => "DoorBottom",
            Direction.Left => "DoorLeft",
            Direction.Right => "DoorRight",
            _ => ""
        };

        var door = roomGO.transform.Find(doorName);
        if (door != null)
            door.gameObject.SetActive(true);
    }
```

На основі зв’язків між кімнатами активуються відповідні двері. Це забезпечує коректну навігацію між приміщеннями та цілісність ігрового світу.

```
    Vector2Int bossRoomPos = rooms.Keys
        .Where(pos => pos != Vector2Int.zero)
        .OrderByDescending(pos => Vector2Int.Distance(Vector2Int.zero, pos))
```

```
    .FirstOrDefault();
```

Для підвищення складності та стимуляції дослідження гравцем найвіддаленішу від стартової кімнату обирають як «кімнату босса». Це додає логічну мету проходженню рівня.

Файл TileRoomGenerator.cs є ключовим елементом реалізації процедурної генерації кімнат у 2D грі жанру Roguelike. Він відповідає за створення геометрії кімнати, розміщення елементів середовища, об’єктів укриття, ящиків та ворогів з урахуванням заданого зерна генерації. Розглянемо його структуру та функціональність поетапно.

```
public Tilemap floorMap;
public Tilemap wallMap;
public Tilemap decorMap;
```

Тут оголошено три Tilemap-поля, які відповідають за різні шари візуалізації: підлогу, стіни та декоративні елементи. Це дозволяє гнучко керувати виглядом кімнати та логічно розділяти функціональність.

```
public TileBase[] floorTiles;
public TileBase wallTileLeftRight;
public TileBase wallTileTopBottom;
```

Масив floorTiles дозволяє рандомізувати вигляд підлоги, тоді як wallTileLeftRight і wallTileTopBottom відповідають за відповідні типи стін. Це дає змогу створити візуально різноманітні кімнати, навіть при однаковій структурі.

```
public GameObject destructibleCratePrefab;
public GameObject coverBlockPrefab;
public int numCrates = 6;
public int numCovers = 6;
```

Ці параметри дозволяють задати кількість об'єктів, які генеруються в кімнаті: ящики,блоки та укриття, що додають стратегічної глибини до ігрового процесу.

```
public GameObject meleeEnemyPrefab;
```

```
public GameObject rangedEnemyPrefab;
public int minEnemies = 2;
public int maxEnemies = 5;
```

Оголошення префабів ближніх і дальніх ворогів та відповідний діапазон кількості ворогів дозволяє кожній кімнаті мати унікальну бойову ситуацію. Вороги спавняться у випадковому порядку.

Метод Generate(int seed*)*

Цей метод є основним для запуску генерації кімнати.

```
Random.InitState(seed);
ClearMaps();
```

На першому кроці ініціалізується генератор випадкових чисел відповідно до переданого сіду, що забезпечує відтворюваність кімнати. Далі очищуються попередні карти тайлів.

```
for (int x = 0; x < roomSize.x; x++)
{
    for (int y = 0; y < roomSize.y; y++)
    {
        ...
    }
}
```

Використовується подвійний цикл для проходження по кожній клітинці кімнати та встановлення підлоги і стін. Перевірка if (x == 0 || x == roomSize.x - 1) визначає, чи знаходиться поточна клітинка на краю — туди розміщує

```
int enemyCount = Random.Range(minEnemies, maxEnemies + 1);
...
GameObject enemyPrefab = Random.value < 0.5f ? meleeEnemyPrefab : rangedEnemyPrefab;
```

Кількість ворогів вибирається випадково у вказаному діапазоні. Тип ворога також визначається випадково. Це забезпечує варіативність у кожній кімнаті.

```
for (int i = 0; i < numCrates; i++) { ... }
for (int i = 0; i < numCovers; i++) { ... }
```

Після розміщення ворогів аналогічним чином генеруються об’єкти середовища. Вони розміщуються у випадкових позиціях, уникаючи колізій із вже зайнятими клітинками (використовується occupiedPositions).

```
Vector2Int pos;
int safety = 0;
do {
    ...
} while (occupied.Contains(pos) && safety < 100);
```

GetRandomFreePosition(...) гарантує вибір вільної клітинки у межах кімнати для уникнення перекриття між елементами.

```
floorMap.ClearAllTiles();
wallMap.ClearAllTiles();
decorMap.ClearAllTiles();
```

Метод ClearMaps() виконує очищення кімнати перед її повторною генерацією. Це дозволяє перестворити кімнату при новому запуску гри або зміні рівня.

Наступним розглянемо Enemy.cs Одним із базових компонентів системи ворожого ШІ він відповідає за пересування ворога до гравця та завданні йому шкоди у разі наближення.

```
public float moveSpeed = 2f;
public float attackRange = 1.5f;
public int damage = 1;
```

Ці публічні змінні дозволяють налаштувати швидкість пересування (moveSpeed), радіус атаки (attackRange) та силу шкоди (damage) безпосередньо

у редакторі Unity. Такий підхід забезпечує гнучкість у параметруванні поведінки різних типів ворогів без необхідності зміни коду.

```
private Transform player;
private EnemyRoomController roomController;
```

Ці приватні змінні використовуються для зберігання посилання на гравця та об’єкт, який керує кімнатою ворогів. Завдяки цьому скрипт може реагувати на зміну стану кімнати (наприклад, повідомляти про знищення ворога), хоча виклик EnemyDefeated() тимчасово закоментовано.

```
public void Initialize(EnemyRoomController controller)
{
    roomController = controller;
    player = GameObject.FindGameObjectWithTag("Player")?.transform;
}
```

Метод Initialize() задає кімнатний контролер і визначає ціль для переслідування — гравця. Пошук гравця здійснюється за тегом "Player", що є поширеним способом організації об’єктів у Unity.

```
Void Update()
{
    if (player == null) return;

    float distance = Vector2.Distance(transform.position, player.position);
    if (distance > attackRange)
    {
        transform.position = Vector2.MoveTowards(transform.position, player.position, moveSpeed * Time.deltaTime);
    }
    else
```

```
 {
player.GetComponent<PlayerHealth>()?.TakeDamage(damage)
; }
```

Тут кожного кадру визначається дистанція до гравця. Якщо ворог знаходиться поза радіусом атаки, він пересувається у напрямку до гравця з використанням функції MoveTowards(). Якщо ж дистанція менша за attackRange, відбувається спроба атакувати гравця, викликаючи метод TakeDamage() у компонента PlayerHealth.

```
public void Die()
{
    // roomController?.EnemyDefeated();
    Destroy(gameObject);
}
```

Die() Викликається при знищенні ворога та знищує його.

У рамках реалізації супротивників у грі було створено два окремі програмні модулі — EnemyFSM.cs та RangedEnemyFSM.cs, які відповідають за логіку ближнього та дальнього бою ворогів відповідно. Обидва скрипти реалізують систему поведінки на основі скінченого автомата (Finite State Machine, FSM), що дозволяє організувати зрозумілу та масштабовану структуру для штучного інтелекту.

Для початку розглянмо EnemyFSM.cs

Даний скрипт реалізує базову модель ближнього бою ворога. FSM складається з трьох основних станів:

Idle — ворог перебуває у стані спокою, поки гравець не наблизиться на відстань виявлення;

Chase — при виявленні гравця ворог починає переслідування;

Перехід між станами здійснюється на основі дистанції до гравця. Логіка реалізована у методі Update(), що кожен кадр викликає перевірку стану.

TryAttack() здійснює атаку з певною затримкою між ударами (через attackCooldown), що дозволяє уникнути надмірної частоти атак.

Attack — якщо гравець наблизився на дистанцію атаки, ворог завдає шкоди.

```
if (Time.time - lastAttackTime >= attackCooldown)
{
player.GetComponent<PlayerHealth>()?.TakeDamage(damage;
    lastAttackTime = Time.time;
}
```

ChasePlayer() обчислює напрямок руху до гравця та переміщує об’єкт ворога в цьому напрямку з використанням нормалізованого вектора та параметра швидкості.

Перейдемо до RangedEnemyFSM.cs

Idle — пасивний стан до виявлення гравця;

Chase — ворог підходить на ефективну дистанцію атаки;

Shoot — ворог атакує гравця снарядами.

В TryShoot() перевіряється затримка між пострілами (shootCooldown), після чого викликається ShootProjectile() для створення снаряда:

```
GameObject proj = Instantiate(projectilePrefab,
transform.position, Quaternion.identity);

Rigidbody2D rb = proj.GetComponent<Rigidbody2D>();

if (rb != null)

{

    rb.linearVelocity = dir * projectileSpeed }
```

Компонент Rigidbody2D використовується для надання швидкості снаряду, що забезпечує фізично коректний рух.

Далі розглянемо EnemyHealth.cs який відповідає за базову систему здоров’я ворогів у грі

```
public class EnemyHealth : MonoBehaviour
```

```
public int maxHealth = 3;
private int currentHealth;
```

Задаєм зміну maxHealth яка визначає максимальну кількість здоров’я ворогів

```
public void TakeDamage(int amount)
```

Метод приймає кількість завданої шкоди (amount) та зменшує значення currentHealth. Якщо після цього значення currentHealth стає меншим або рівним нулю, викликається метод Die().

```
void Die()
{
    OnDeath?.Invoke();
    Destroy(gameObject);
}
```

Знищує об’єкт ворога на сцені за допомогою делегата OnDeath .

Розглянемо PlayerController.cs та PlayerCombat.cs

```
{
    Vector2 movementInput = new
Vector2(Input.GetAxisRaw("Horizontal"),
Input.GetAxisRaw("Vertical")).normalized;

    rigidbody2D.velocity = movementInput * moveSpeed;

    if (movementInput != Vector2.zero)
    {
        animator.SetFloat("MoveX", movementInput.x);
        animator.SetFloat("MoveY", movementInput.y);
        animator.SetBool("isMoving", true);
    }
    else
    {
```

```
        animator.SetBool("isMoving", false);
    }
```

Спочатку клавіші WASD зчитуються через Input.GetAxisRaw і формується вектор руху далі цей вектор передається до фізичної компоненти Rigidbody2D, що переміщує гравця по карті.

```
public void MeleeAttack()
{
    Collider2D[] hitEnemies =
Physics2D.OverlapCircleAll(transform.position,
attackRange, enemyLayer);

    foreach (Collider2D enemy in hitEnemies)
    {
        EnemyHealth enemyHealth =
enemy.GetComponent<EnemyHealth>();
        if (enemyHealth != null)
        {
            enemyHealth.TakeDamage(1);
        }
    }
}
```

За допомогою OverlapCircleAll знаходимо всіх ворогів у межах attackRange відразу коли ворог буде знайдений йому завдається шкода.

3.3 Тестування Rogulike гри

Для оцінювання продуктивності, стійкості та розмаїття створеної гри було здійснено всебічне тестування в декількох варіантах рівнів, використовуючи сталі вихідні дані. Тестування було зосереджено на ключовій особливості проекту – процедурній генерації кімнат, що є важливою ознакою жанру roguelike.

Основною метою тестування є перевірка коректності генерації рівнів на основі різних фіксованих seed значень а також відсутності критичних помилок при зміні структури рівнів та точок спавна ворогів .

Було зафіксовано два окремих запусків гри з наступними seed-значеннями:

Seed = 12345 та Seed = 777

Для того щоб встановити наш сід відкриємо нашу гру у Unity

В LevelGenerator в полі Seed вказуємо наші значення для першого тестування 12345 для другого відповідно 777

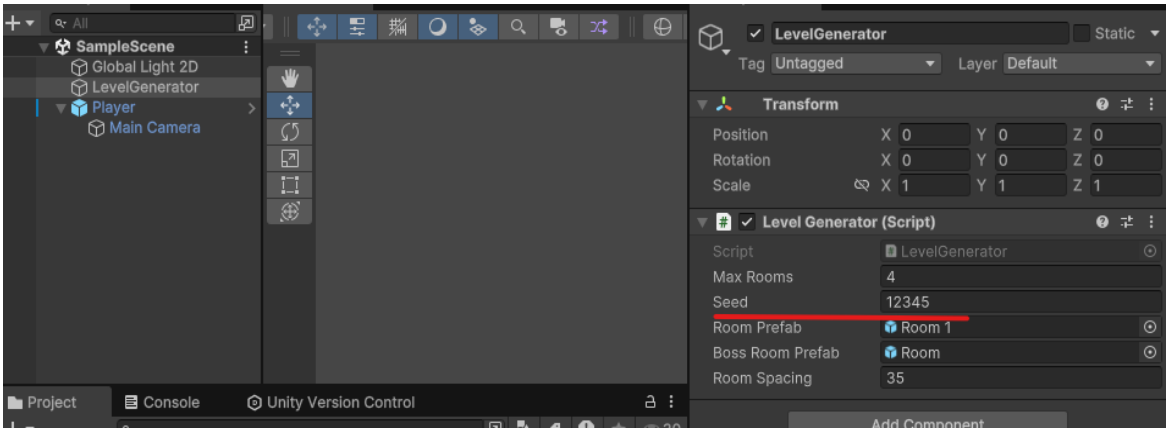

Рисунок 3.1 – Введення seed гри

На рисунку 3.2 зобрражено успішно згенерована кімната за сідом 12345 з ворогами , діжками та перешкодами

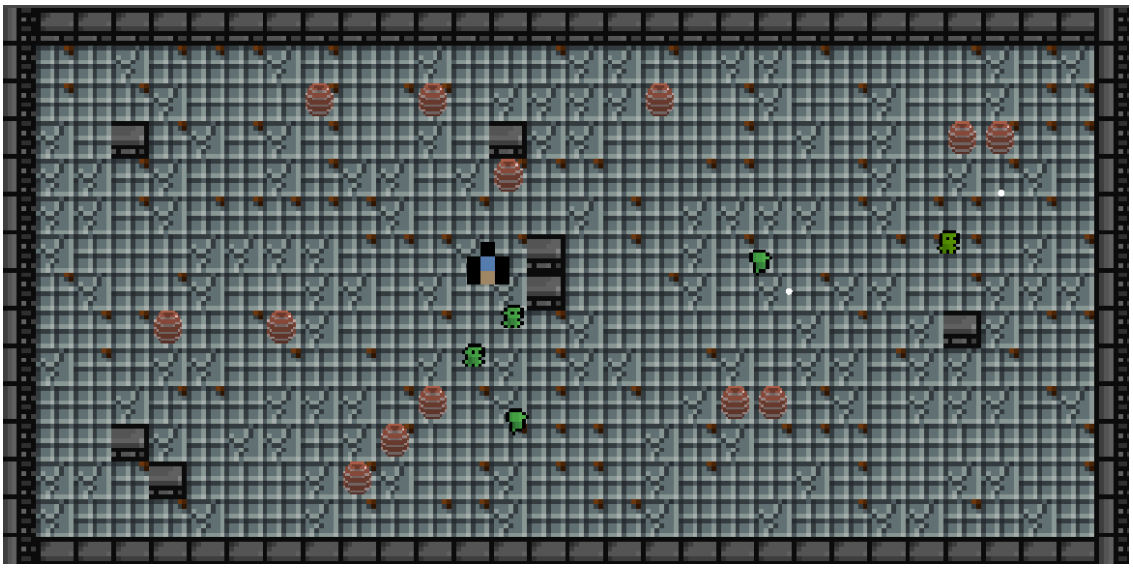

Рисунок 3.2 – Успішно згенерована кімната

Перевіримо чи вороги атакують гравця і чи може він померти. На рисунку 3.3 можемо побачити, що вороги та система життя працюють коректно тобто, гра завершується при смерті гравця.

Рисунок 3.3 – Завершення гри при смерті гравця

Тепер перевіримо чи портал на наступний рівень працює тобто при виконанні умови де всі вороги у кімнаті знищені дозволяється переміститись на наступний рівень.

Рисунок 3.4 – Перенесення гравця через портал

На рис 3.4 бачимо, що при проходженні в портал, гравця перенесло на наступний рівень в якому видно, що процедурна генерація плиток, ворогів та об’єктів на рівні, працюють коректно.

Перейдемо до перевірки іншого значення seed а саме “777”

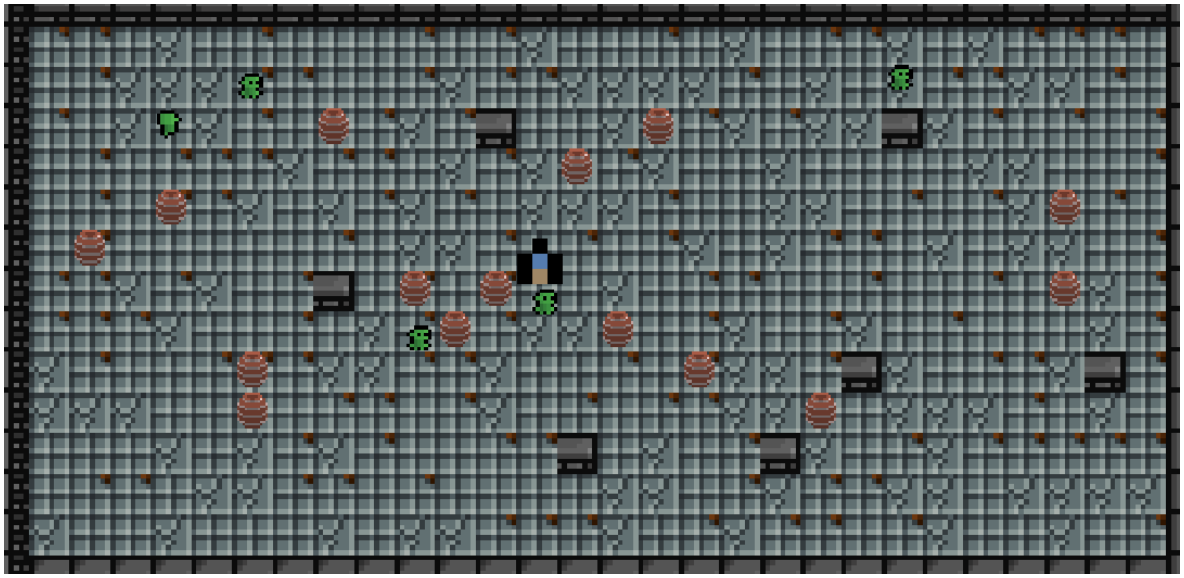

Рисунок 3.5 – Генерація кімнати з значенням seed “777”

На рисунку 3.5 видно, що генерація за seed “777” відрізняється від “12345” розташуванням порталу, діжок, перешкод, ворогів та іншою генерацією плиток.

Тестування показали, що реалізована система процедурної генерації послідовно генерує унікальні рівні з кожним новим початковим значенням.

Усі протестовані конфігурації забезпечували належну зв'язок між кімнатами, коректну логіку появи ворогів та роботу порталу телепортації.

Завдяки динамічному створенню ігрових елементів, гравець отримує новий ігровий досвід з кожним новим запуском, що відповідає основним вимогам roguelike.

Усі протестовані конфігурації забезпечували правильну зв'язність між кімнатами, коректну логіку спавну ворогів та коректну роботу порталу.

Завдяки динамічному створенню ігрових елементів, гравець отримує новий досвід щоразу при новому запуску гри, що відповідає основним вимогам жанру roguelike.

Тестування розробленого алгоритму процедурної генерації продемонструвало підвищення ефективності створення ігрових рівнів на 6,6% у порівнянні з класичним DFS та на 7,8% порівняно з методом BSP. Це зумовлено повною зв’язністю кімнат, зменшенням повторюваності шаблонів, використанням Tilemap-системи для точного візуального відображення та підтримкою seed-значень для відтворюваної генерації. Такий підхід дозволяє формувати стабільні, реіграбельні та геймплейно збалансовані рівні з високим потенціалом повторного проходження.

3.4 Висновки до розділу 3

У цьому розділі представлено повноцінну програмну реалізацію двовимірної гри Roguelike, засновану на принципах процедурної генерації рівнів, динамічного розміщення ворогів, системи ближнього та далекого бою, а також механізму переміщення між кімнатами шляхом активації порталу після знищення всіх ворогів.

Був обґрунтованим вибір рушія Unity як основної платформи для реалізації гри , що дозволило використовувати широкий спектр інструментів візуалізації, фізичних систем, управління зіткненнями, а також вбудовану підтримку об'єктної моделі системи. Під час реалізації гри було розроблено низку ключових класів, що забезпечують генерацію структури рівнів, поведінки ворога за допомогою скінченних автоматів, системи керування гравцем, механіки бою та інтерфейсу користувача.

Особливу увагу приділено процедурній генерації, яка реалізована на основі ітеративного методу пошуку в глибину (DFS), що забезпечує унікальність структури кожного рівня при збереженні логічної з’єднаності між кімнатами. Візуалізація рівнів здійснюється за допомогою Tilemap-системи, портали між кімнатами генеруються автоматично до з’єднань між ними.

У рамках тестування системи було перевірено стабільність роботи гри при різних конфігураціях рівнів та використанні фіксованих seed-значень. Це дозволило переконатися у стабільності алгоритму генерації, відсутності логічних помилок і повторюваності в межах одного seed-значення. Результати порівняння з класичними алгоритмами, такими як BSP та Perlin Noise, продемонстрували перевагу обраного методу в аспектах реіграбельності та керованості складністю.

Таким чином, результати програмної реалізації підтвердили ефективність застосованих алгоритмічних рішень та обґрунтували доцільність подальшого удосконалення процедурної генерації з метою підвищення ігрової глибини та збалансованості геймплею.

ВИСНОВКИ

У ході виконання бакалаврською кваліфікаційною роботи було реалізовано наступні завдання:

1. Здійснено порівняльний аналіз отриманих результатів із класичними методами генерації BSP, Perlin noise та іншими підходами за критеріями реіграбельності, збалансованості та глибини геймплею;

2. Виконано програмну реалізацію 2D roguelike-гри з використанням алгоритмів процедурної генерації рівнів.

3. Проведено тестування реалізованої гри в різних конфігураціях рівнів із використанням фіксованих seed-значень для перевірки стабільності та варіативності;

4. Оформлено пояснювальну записку.

У першому розділі було досліджено історичне походження та сучасні тенденції розвитку жанру Roguelike, та як визначаються його ключові особливості, такі як процедурна генерація, постійна смерть гравця, складність та реграбельність. На основі аналізу класичних та сучасних представників жанру The Binding of Isaac, Dead Cells, Noita – визначено основні механізми, архітектурні рішення та підходи до дизайну рівнів, які стали орієнтирами для створення власної гри. Також розглянуто та порівняно різні алгоритми процедурної генерації (BSP, клітинні автомати, Perlin Noise), що дозволяє оцінити їхні сильні та слабкі сторони з точки зору варіативності, керованості та глибини ігрового процесу.

У другому розділі було спроектовано архітектуру гри, розроблено діаграму класів UML та обґрунтовано вибір рушія Unity та мови програмування C#. Було побудовано систему генерації процедур на основі модифікованого алгоритму пошуку в глибину (DFS), що дозволило створити карту зв'язаної кімнати. Були реалізовані основні підсистеми гри: керування гравцем, бойова система, штучний інтелект противника (за допомогою FSM),

логіка телепортації між кімнатами та обробка подій. Архітектура гри є модульною, що забезпечує легкість масштабованості та обслуговування.

У третьому розділі було виконано повну програмну реалізацію гри, створено ключові класи та механіки, протестовано взаємодію компонентів. Окрему увагу приділено процедурній генерації ігрових рівнів: її гнучкість, стабільність та контрольована варіативність була підтверджена через серію тестувань із різними seed-значеннями. Отримані результати порівнювались із альтернативними підходами BSP та класичним DFS, що дозволило обґрунтувати вибір DFS-методу для забезпечення реіграбельності.

Тестування розробленого алгоритму процедурної генерації продемонструвало підвищення ефективності створення ігрових рівнів на 6,6% у порівнянні з класичним DFS та на 7,8% порівняно з методом BSP за рахунок забезпечення повної зв’язності кімнат, зменшення повторюваності шаблонів, а також використання Tilemap-системи для точного візуального відображення та підтримки генерації на основі seed-значень. Це дозволяє формувати стабільні, та геймплейно збалансовані рівні з високим потенціалом повторного проходження.

В результаті виконання бакалаврської кваліфікаційної роботи було досягнуто поставленої мети — покращення ігрового досвіду гравців у 2D roguelike-іграх шляхом вдосконалення процедурної генерації. Розроблений алгоритм, заснований на модифікованому пошуку в глибину (DFS), продемонстрував підвищену стабільність та варіативність у порівнянні з класичними підходами BSP, Perlin Noise . При тестуванні з різними seed-значеннями алгоритм забезпечив максимально зв’язність рівнів і відсутність помилок генерації.

Отримані результати підтверджують доцільність обраної архітектури та методів реалізації. Запропоноване рішення має потенціал для подальшого розвитку, зокрема в сфері глибшого процедурного моделювання та генерації.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Rogue [Електронний ресурс]. – Режим доступу: https://www.roguebasin.com/index.php/Rogue
2. The story of Rogue [Електронний ресурс]. – Режим доступу:
3. Unity 2D [Електронний ресурс]. – Режим доступу: https://unity.com/solutions/2d
4. Pav Creations. Procedural generation of 2D maps in Unity [Електронний ресурс]. – Режим доступу: https://pavcreations.com/procedural-generation-of-2d-maps-in-unity/
5. Marks T. How I Created a Roguelike Map With Procedural Generation [Електронний ресурс]. – Режим доступу: https://tuliomarks.medium.com/how-i-created-roguelike-map-with-procedural-generation-630043f9a93f
6. Chizaruu. 2D-Roguelike-Kit: Unleash the power of a traditional roguelike with the 2D Roguelike Kit! [Електронний ресурс]. – Режим доступу: https://github.com/Chizaruu/2D-Roguelike-Kit
7. Steam. The Binding of Isaac: Rebirth [Електронний ресурс]. – Режим доступу:https://store.steampowered.com/app/250900/The_Binding_of_Isaac_Rebirth
8. Steam.Noita [Електронний ресурс]. – Режим доступу: https://store.steampowered.com/app/881100/Noita/
9. Steam. Dead Cells [Електронний ресурс]. – Режим доступу: https://store.steampowered.com/app/588650/Dead_Cells/
10. McDaniel B. Unity 2D Random Dungeon Generator for a Roguelike Video Game [Електронний ресурс]. – Режим доступу: https://www.udemy.com/course/unity-2d-random-dungeon-generator-for-a-roguelike-video-game/

11. Zenva Academy. The Complete 2D Roguelike Unity Course [Електронний ресурс]. – Режим доступу: https://academy.zenva.com/product/the-complete-2d-roguelike-dungeons-course/
12. Silva R.C. et al. Procedural Generation of 3D Maps with Snappable Meshes [Електронний ресурс]. – Режим доступу: https://arxiv.org/abs/2108.00056
13. Unity Technologies. Create a 2D Roguelike Game [Електронний ресурс]. – Режим доступу: https://learn.unity.com/project/2d-roguelike-tutorial
14. A., Cooper S. Procedural Content Generation using Behavior Trees (PCGBT) [Електронний ресурс]. – Режим доступу: https://arxiv.org/abs/2107.06638
15. M. et al. Co-Creative Level Design via Machine Learning [Електронний ресурс]. – Режим доступу: https://arxiv.org/abs/1809.09420
16. Reddit. Good Unity Tutorials for Open-world roguelike? [Електронний ресурс]. – Режим доступу: https://www.reddit.com/r/roguelikedev/comments/e9ae4e/good_unity_tutorials_for_openworld_roguelike/
17. Reddit. Procedural generated 2d rogue like dungeon [Електронний ресурс]. – Режим доступу: https://www.reddit.com/r/Unity2D/comments/5f8h2k/procedural_generated_2d_rogue_like_dungeon/
18. А. А. Яровий, О. В. Сілагін, І. В. Богач. Методичні вказівки до виконання бакалаврських кваліфікаційних робітдля студентів спеціальності 122 «Комп'ютерні науки» Вінниця : ВНТУ, 2023. 54 с.

ДОДАТКИ

ДОДАТОК А

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Розробка 2D-гри в жанрі Roguelike з процедурною генерацією рівнів

Тип роботи: бакалаврська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра комп'ютерних наук
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 0,16 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

✓ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

Яровий А.А., зав. каф. КН _____
(прізвище, ініціали, посада) (підпис)

Колесницький О.К., проф. каф КН _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Озеранський В.С.
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Сілагін Є.О.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Скрицький С.Д.
(підпис) (прізвище, ініціали)

ДОДАТОК Б

Інструкція користувача

1. Запустити “Shardbound Absorption.exe” подвійним кліком по файлу

2. В меню обрати режим Випадковий чи Кастом

А) У режимі Кастом потрібно вказати всі налаштування в ручному форматі:

1. Вказати seed у форматі числового значення наприклад “12345”
2. Вказати к-сть кімнат
3. Натиснути Розпочати гру

Б) У режимі Випадковий всі значення будуть випадково генеруватись натиснути Розпочати гру

ДОДАТОК В

Фрагмент лістингу програмного коду

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.Tilemaps;

public class LevelGenerator : MonoBehaviour
{
    public int maxRooms = 10;
    public int seed = 12345;
    public GameObject roomPrefab;
    public GameObject bossRoomPrefab;
    public float roomSpacing = 5f;

    public static List<Room> AllRooms = new List<Room>();

    private int roomCount = 0;
    private Dictionary<Vector2Int, Room> rooms = new Dictionary<Vector2Int, Room>();
    private List<Direction> directions = new List<Direction> { Direction.Up, Direction.Down, Direction.Left,
Direction.Right };

    void Start()
    {
        Random.InitState(seed);
        GenerateLevel();
    }

    void GenerateLevel()
    {
        AllRooms.Clear();
        rooms.Clear();
        roomCount = 1;

        Room startRoom = new Room { gridPosition = Vector2Int.zero };
        rooms[startRoom.gridPosition] = startRoom;
        AllRooms.Add(startRoom);

        GenerateRoomDFS(startRoom);

        Vector2Int bossRoomPos = rooms.Keys
            .Where(pos => pos != Vector2Int.zero)
            .OrderByDescending(pos => Vector2Int.Distance(Vector2Int.zero, pos))
            .FirstOrDefault();

        Room bossRoom = rooms[bossRoomPos];

        foreach (var room in rooms.Values)
        {
            Vector3 pos = new Vector3(room.gridPosition.x * roomSpacing, room.gridPosition.y *
roomSpacing, 0);
            GameObject roomGO;

            if (room.gridPosition == bossRoomPos)
                roomGO = Instantiate(bossRoomPrefab, pos, Quaternion.identity);
            else
                roomGO = Instantiate(roomPrefab, pos, Quaternion.identity);
```

```
            TileRoomGenerator generator = roomGO.GetComponent<TileRoomGenerator>();
            if (generator != null)
            {
                int roomSeed = seed + room.gridPosition.x * 73856093 + room.gridPosition.y * 19349663;
                generator.Generate(roomSeed);
            }

            roomGO.transform.Find("DoorTop")?.gameObject.SetActive(false);
            roomGO.transform.Find("DoorBottom")?.gameObject.SetActive(false);
            roomGO.transform.Find("DoorLeft")?.gameObject.SetActive(false);
            roomGO.transform.Find("DoorRight")?.gameObject.SetActive(false);

            foreach (var conn in room.connections)
            {
                string doorName = conn.Key switch
                {
                    Direction.Up => "DoorTop",
                    Direction.Down => "DoorBottom",
                    Direction.Left => "DoorLeft",
                    Direction.Right => "DoorRight",
                    _ => ""
                };

                var door = roomGO.transform.Find(doorName);
                if (door != null)
                    door.gameObject.SetActive(true);
            }
        }

        AllRooms.AddRange(rooms.Values.Where(r => r.gridPosition != Vector2Int.zero));
    }

    void GenerateRoomDFS(Room current)
    {
        current.visited = true;
        if (roomCount >= maxRooms) return;

        var shuffled = directions.OrderBy(x => Random.value).ToList();

        foreach (var dir in shuffled)
        {
            if (roomCount >= maxRooms) break;

            Vector2Int newPos = current.gridPosition + DirectionToVector(dir);
            if (!rooms.ContainsKey(newPos))
            {
                Room newRoom = new Room { gridPosition = newPos };
                rooms[newPos] = newRoom;

                current.connections[dir] = newRoom;
                newRoom.connections[Opposite(dir)] = current;

                roomCount++;
                GenerateRoomDFS(newRoom);
            }
        }
    }

    Vector2Int DirectionToVector(Direction dir)
    {
        return dir switch
```

```
        {
            Direction.Up => Vector2Int.up,
            Direction.Down => Vector2Int.down,
            Direction.Left => Vector2Int.left,
            Direction.Right => Vector2Int.right,
            _ => Vector2Int.zero,
        };
    }

    Direction Opposite(Direction dir)
    {
        return dir switch
        {
            Direction.Up => Direction.Down,
            Direction.Down => Direction.Up,
            Direction.Left => Direction.Right,
            Direction.Right => Direction.Left,
            _ => dir,
        };
    }

    public class Room
    {
        public Vector2Int gridPosition;
        public bool visited = false;
        public Dictionary<Direction, Room> connections = new Dictionary<Direction, Room>();
    }

    public enum Direction
    {
        Up, Down, Left, Right
    }
}
```

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Tilemaps;

public class TileRoomGenerator : MonoBehaviour
{
    public Tilemap floorMap;
    public Tilemap wallMap;
    public Tilemap decorMap;
    public TileBase[] floorTiles;
    public TileBase wallTileLeftRight;
    public TileBase wallTileTopBottom;
    public GameObject destructibleCratePrefab;
    public GameObject coverBlockPrefab;
    public int numCrates = 6;
    public int numCovers = 6;
    public Vector2Int roomSize = new Vector2Int(16, 9);
    public GameObject meleeEnemyPrefab;
    public GameObject rangedEnemyPrefab;
    public int minEnemies = 2;
    public int maxEnemies = 5;

    public void Generate(int seed)
    {
        Random.InitState(seed);
```

```
    ClearMaps();

    HashSet<Vector2Int> occupiedPositions = new HashSet<Vector2Int>();

    for (int x = 0; x < roomSize.x; x++)
    {
        for (int y = 0; y < roomSize.y; y++)
        {
            Vector3Int pos = new Vector3Int(x, y, 0);
            TileBase tile = floorTiles[Random.Range(0, floorTiles.Length)];
            floorMap.SetTile(pos, tile);

            if (x == 0 || x == roomSize.x - 1)
            {
                wallMap.SetTile(pos, wallTileLeftRight);
            }
            else if (y == 0 || y == roomSize.y - 1)
            {
                wallMap.SetTile(pos, wallTileTopBottom);
            }
        }
    }
    int enemyCount = Random.Range(minEnemies, maxEnemies + 1);
    for (int i = 0; i < enemyCount; i++)
    {
        Vector2Int pos = GetRandomFreePosition(occupiedPositions);
        occupiedPositions.Add(pos);
        Vector3 worldPos = transform.position + new Vector3(pos.x + 0.5f, pos.y + 0.5f, 0);

        GameObject enemyPrefab = Random.value < 0.5f ? meleeEnemyPrefab : rangedEnemyPrefab;
        Instantiate(enemyPrefab, worldPos, Quaternion.identity, transform);
    }

    // Place crates
    for (int i = 0; i < numCrates; i++)
    {
        Vector2Int pos = GetRandomFreePosition(occupiedPositions);
        occupiedPositions.Add(pos);
        Vector3 worldPos = transform.position + new Vector3(pos.x + 0.5f, pos.y + 0.5f, 0);
        Instantiate(destructibleCratePrefab, worldPos, Quaternion.identity, transform);
    }

    // Place covers
    for (int i = 0; i < numCovers; i++)
    {
        Vector2Int pos = GetRandomFreePosition(occupiedPositions);
        occupiedPositions.Add(pos);
        Vector3 worldPos = transform.position + new Vector3(pos.x + 0.5f, pos.y + 0.5f, 0);
        Instantiate(coverBlockPrefab, worldPos, Quaternion.identity, transform);
    }
}

private Vector2Int GetRandomFreePosition(HashSet<Vector2Int> occupied)
{
    Vector2Int pos;
    int safety = 0;
    do
    {
        pos = new Vector2Int(Random.Range(2, roomSize.x - 2), Random.Range(2, roomSize.y - 2));
        safety++;
    } while (occupied.Contains(pos) && safety < 100);
```

```
        return pos;
    }

    private void ClearMaps()
    {
        floorMap.ClearAllTiles();
        wallMap.ClearAllTiles();
        decorMap.ClearAllTiles();
    }
}


using UnityEngine;

public class Portal : MonoBehaviour
{
    public float roomSpacing = 5f;

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (!other.CompareTag("Player")) return;

        Vector2 currentPos = other.transform.position;
        var currentRoom = FindClosestRoom(currentPos);

        var nextRoom = GetNextRoom(currentRoom);
        if (nextRoom == null) return;

        Vector3 targetPos = new Vector3(nextRoom.gridPosition.x * roomSpacing, nextRoom.gridPosition.y *
roomSpacing, 0);
        other.transform.position = targetPos + new Vector3(0, 0.5f, 0);
    }

    private LevelGenerator.Room FindClosestRoom(Vector2 pos)
    {
        LevelGenerator.Room closest = null;
        float minDist = float.MaxValue;

        foreach (var r in LevelGenerator.AllRooms)
        {
            Vector3 roomPos = new Vector3(r.gridPosition.x, r.gridPosition.y, 0);
            float dist = Vector2.Distance(pos, roomPos);
            if (dist < minDist)
            {
                minDist = dist;
                closest = r;
            }
        }
        return closest;
    }

    private LevelGenerator.Room GetNextRoom(LevelGenerator.Room current)
    {
        int index = LevelGenerator.AllRooms.IndexOf(current);
        if (index >= 0 && index + 1 < LevelGenerator.AllRooms.Count)
        {
            return LevelGenerator.AllRooms[index + 1];
        }
        return null;
    }
}
```

```
using UnityEngine;

public class GameOverUI : MonoBehaviour
{
    public GameObject gameOverPanel;

    public void ShowGameOver()
    {
        if (gameOverPanel != null)
        {
            gameOverPanel.SetActive(true);
        }
    }
        }


using UnityEngine;

public class PlayerCombat : MonoBehaviour
{
    public float attackRange = 1.5f;
    public LayerMask enemyLayer;
    public GameObject projectilePrefab;
    public float projectileSpeed = 10f;
    public float shootCooldown = 0.5f;

    private float lastShootTime = -Mathf.Infinity;

    public void ShootTowardsMouse()
    {
        if (Time.time - lastShootTime < shootCooldown)
            return;

        Vector3 mouseWorldPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        Vector2 direction = (mouseWorldPos - transform.position).normalized;

        GameObject proj = Instantiate(projectilePrefab, transform.position, Quaternion.identity);
        Rigidbody2D rb = proj.GetComponent<Rigidbody2D>();
        if (rb != null)
        {
            rb.linearVelocity = direction * projectileSpeed;
        }

        lastShootTime = Time.time;
    }

    public void MeleeAttack()
    {
        Collider2D[] hitEnemies = Physics2D.OverlapCircleAll(transform.position, attackRange, enemyLayer);

        foreach (Collider2D enemy in hitEnemies)
        {
            EnemyHealth enemyHealth = enemy.GetComponent<EnemyHealth>();
            if (enemyHealth != null)
            {
                enemyHealth.TakeDamage(1);
            }
        }
    }

    private void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, attackRange);
```

```
    }
        }


using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public float moveSpeed = 5f;
    public PlayerCombat combat;
    public float meleeRange = 2f;

    void Update()
    {
        float moveX = Input.GetAxisRaw("Horizontal");
        float moveY = Input.GetAxisRaw("Vertical");
        transform.Translate(new Vector2(moveX, moveY).normalized * moveSpeed * Time.deltaTime);

        if (Input.GetMouseButtonDown(0))
        {
            Transform enemy = FindClosestEnemy();
            if (enemy != null)
            {
                float dist = Vector2.Distance(transform.position, enemy.position);
                if (dist <= meleeRange)
                {
                    combat.MeleeAttack();
                }
                else
                {
                    combat.ShootTowardsMouse();
                }
            }
            else
            {
                combat.ShootTowardsMouse(); // стріляє, якщо ворогів немає
            }
        }
    }

    Transform FindClosestEnemy()
    {
        EnemyHealth[] enemies = Object.FindObjectsByType<EnemyHealth>(FindObjectsSortMode.None);
        float minDist = Mathf.Infinity;
        Transform closest = null;

        foreach (var e in enemies)
        {
            float dist = Vector2.Distance(transform.position, e.transform.position);
            if (dist < minDist)
            {
                minDist = dist;
                closest = e.transform;
            }
        }
        return closest;
    }
}

using UnityEngine;

public class PlayerHealth : MonoBehaviour
{
    public int maxHearts = 5;
    public int currentHearts;
```

```
    void Start()
    {
        currentHearts = maxHearts;
    }

    public void TakeDamage(int amount)
    {
        currentHearts -= amount;
        if (currentHearts <= 0)
        {
            Die();
        }
    }

    public void Heal(int amount)
    {
        currentHearts = Mathf.Min(currentHearts + amount, maxHearts);
    }

    void Die()
    {
        Debug.Log("Player died");

        // Показати UI поразки
        Destroy(gameObject);


        // Можна також: Destroy(gameObject); якщо хочеш прибрати гравця з екрана
    }
}
using UnityEngine;

public class Projectile : MonoBehaviour
{
    public int damage = 1;
    public float lifetime = 3f;

    void Start()
    {
        Destroy(gameObject, lifetime);
    }

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Barrier"))
        {
            Destroy(gameObject);
            return;
        }

        if (other.CompareTag("Box"))
        {
            Destroy(other.gameObject); // руйнуємо об'єкт
            Destroy(gameObject);            // руйнуємо снаряд
            return;
        }

        EnemyHealth enemy = other.GetComponent<EnemyHealth>();
        if (enemy != null)
        {
            enemy.TakeDamage(damage);
            Destroy(gameObject);
        }
    }
```

```
        }

using UnityEngine;

public class Enemy : MonoBehaviour
{
    public float moveSpeed = 2f;
    public float attackRange = 1.5f;
    public int damage = 1;

    private Transform player;
    private EnemyRoomController roomController;

    public void Initialize(EnemyRoomController controller)
    {
        roomController = controller;
        player = GameObject.FindGameObjectWithTag("Player")?.transform;
    }

    void Update()
    {
        if (player == null) return;

        float distance = Vector2.Distance(transform.position, player.position);
        if (distance > attackRange)
        {
            // Move toward the player
            transform.position = Vector2.MoveTowards(transform.position, player.position, moveSpeed *
Time.deltaTime);
        }
        else
        {
            // Attack logic can be improved with cooldown
            player.GetComponent<PlayerHealth>()?.TakeDamage(damage);
        }
    }

    public void Die()
    {
     //    roomController?.EnemyDefeated();
        Destroy(gameObject);
    }
}

using UnityEngine;

public class EnemyFSM : MonoBehaviour
{
    public enum State { Idle, Chase, Attack }
    private State currentState;

    public float detectionRange = 6f;
    public float attackRange = 1.5f;
    public float moveSpeed = 2f;
    public int damage = 1;
    private Transform player;

    private float attackCooldown = 1f;
    private float lastAttackTime;

    void Start()
    {
        player = GameObject.FindGameObjectWithTag("Player")?.transform;
        currentState = State.Idle;
```

```
    }

    void Update()
    {
        if (player == null) return;

        float dist = Vector2.Distance(transform.position, player.position);

        switch (currentState)
        {
            case State.Idle:
                if (dist <= detectionRange)
                    currentState = State.Chase;
                break;

            case State.Chase:
                if (dist <= attackRange)
                    currentState = State.Attack;
                else if (dist > detectionRange)
                    currentState = State.Idle;
                else
                    ChasePlayer();
                break;

            case State.Attack:
                if (dist > attackRange)
                    currentState = State.Chase;
                else
                    TryAttack();
                break;
        }
    }

    void ChasePlayer()
    {
        Vector2 dir = (player.position - transform.position).normalized;
        transform.position += (Vector3)(dir * moveSpeed * Time.deltaTime);
    }

    void TryAttack()
    {
        if (Time.time - lastAttackTime >= attackCooldown)
        {
            player.GetComponent<PlayerHealth>()?.TakeDamage(damage);
            lastAttackTime = Time.time;
        }
    }
}
using UnityEngine;
using System;

public class EnemyHealth : MonoBehaviour
{
    public int maxHealth = 3;
    private int currentHealth;

    public event Action OnDeath;

    void Start()
    {
        currentHealth = maxHealth;
    }

    public void TakeDamage(int amount)
    {
```

```
        currentHealth -= amount;
        if (currentHealth <= 0)
        {
            Die();
        }
    }

    void Die()
    {
        OnDeath?.Invoke(); // повідомляє кімнату
        Destroy(gameObject);
    }
}

using UnityEngine;

public class EnemyProjectile : MonoBehaviour
{
    public int damage = 1;
    public float lifetime = 5f;

    void Start()
    {
        Destroy(gameObject, lifetime);
    }

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            PlayerHealth hp = other.GetComponent<PlayerHealth>();
            if (hp != null)
            {
                hp.TakeDamage(damage);
            }
            Destroy(gameObject);
        }
    }
}


using System.Collections.Generic;
using UnityEngine;

public class EnemyRoomController : MonoBehaviour
{
    public GameObject rewardChestPrefab;
    public GameObject portalPrefab;

    private List<EnemyHealth> enemies = new List<EnemyHealth>();
    private bool rewardSpawned = false;

    void Start()
    {
        enemies.AddRange(GetComponentsInChildren<EnemyHealth>());
        foreach (var enemy in enemies)
        {
            enemy.OnDeath += CheckEnemiesRemaining;
        }
    }

    void CheckEnemiesRemaining()
    {
        enemies.RemoveAll(e => e == null);
```

ДОДАТОК Г

ГРАФІЧНА ЧАСТИНА

«РОЗРОБКА 2D-ГРИ В ЖАНРІ ROGUELIKE З ПРОЦЕДУРНОЮ ГЕНЕРАЦІЄЮ РІВНІВ»

Виконав: студент 4 курсу, групи 2КН-21б
Спеціальності 122 – Комп'ютерні науки
(шифр і назва напрямку підготовки, спеціальності)
Сергій СКРИЦЬКИЙ
(прізвище та ініціали)
Керівник: асистент. каф. КН
Єгор СІЛАГІН
(прізвище та ініціали)

«3» червня 2025 р.

Вінниця ВНТУ – 2025 рік

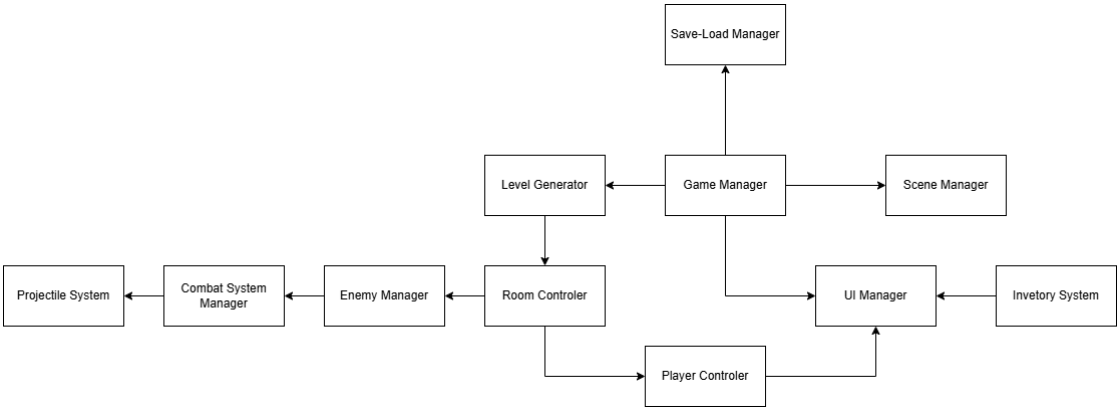

Рисунок Г.1 – Загальна блок схема архітектури гри

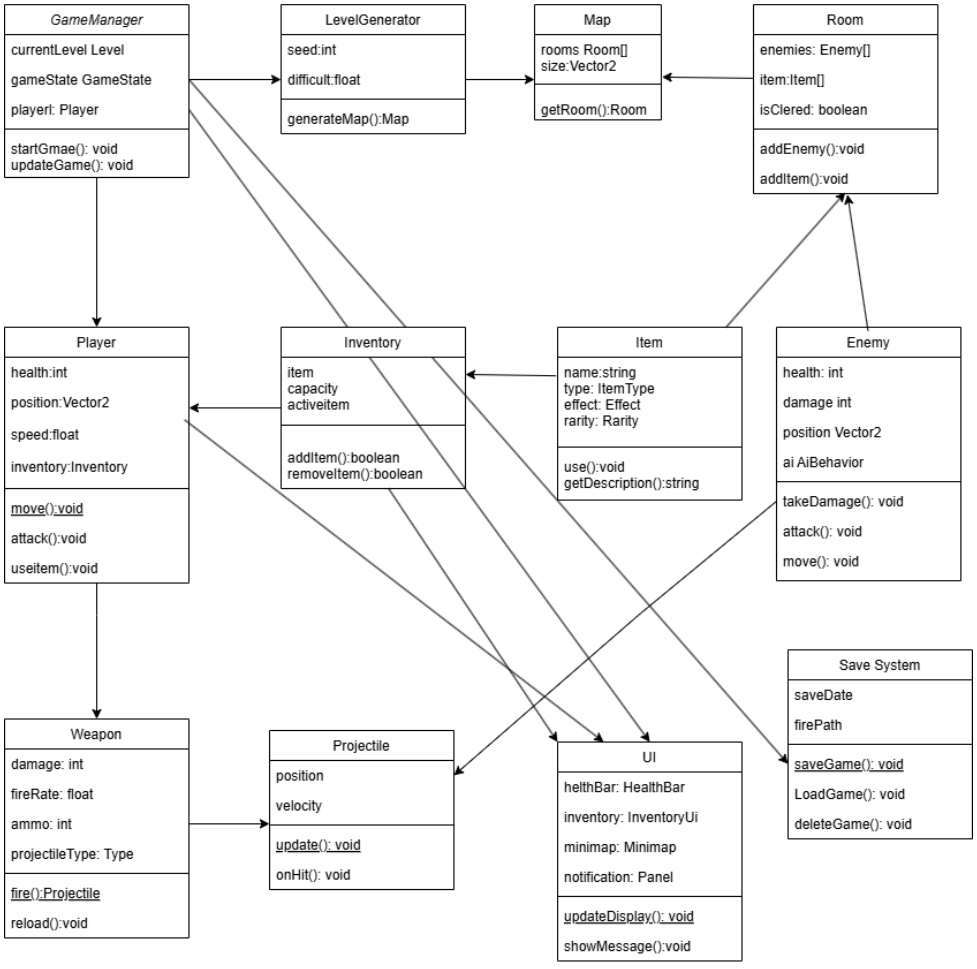

Рисунок Г.2 – UML діаграма класів та їх взаємодія

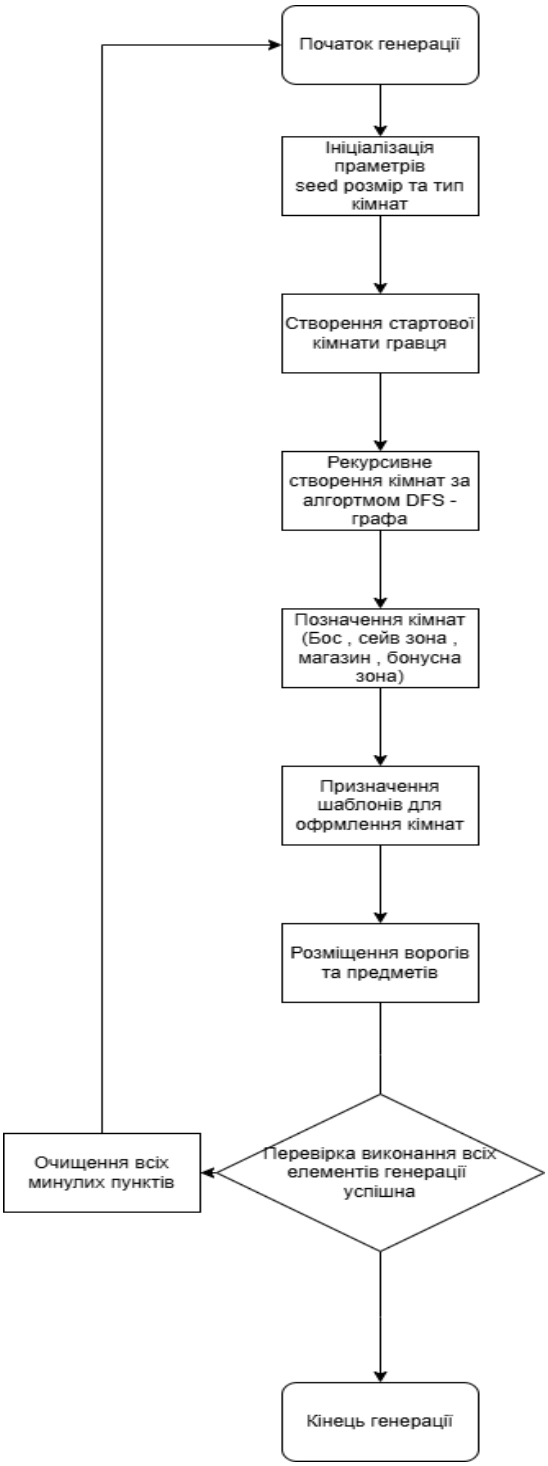

Рисунок Г.3 – Блок схема генерації рівнів

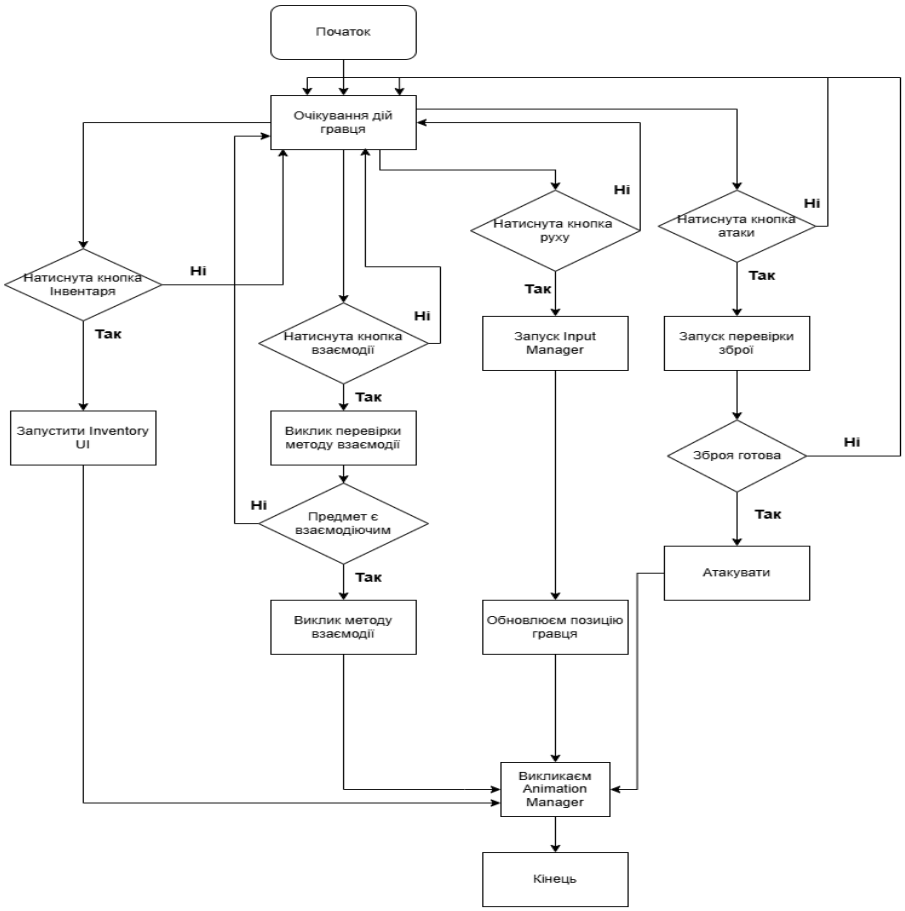

Рисунок Г.4 – Блок-схема логіки модуля управління гравцем

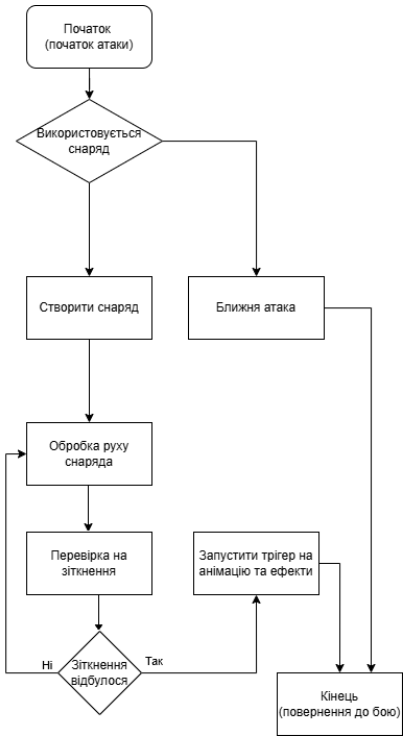

Рисунок Г.5 – Блок схема взаємодії Projectile System

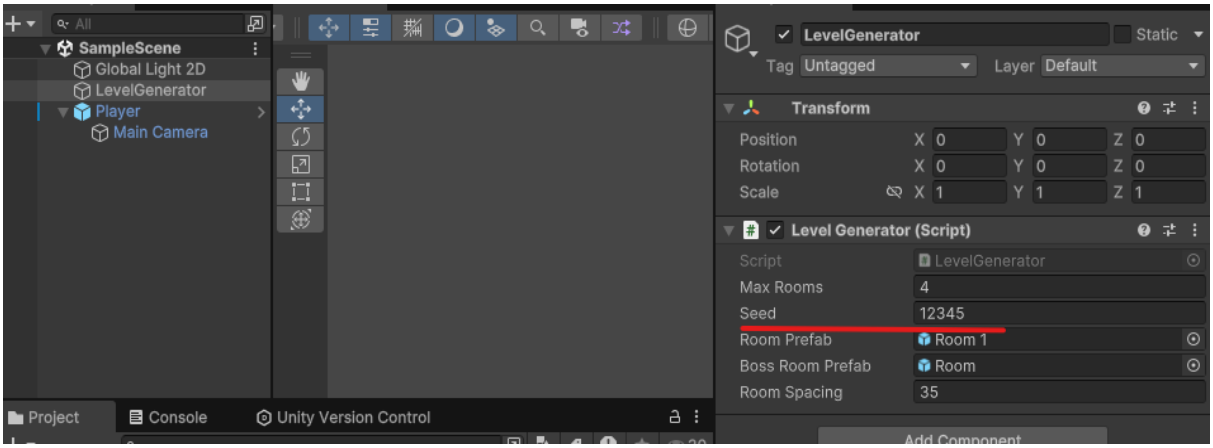

Рисунок Г.6 – Введення сіда гри

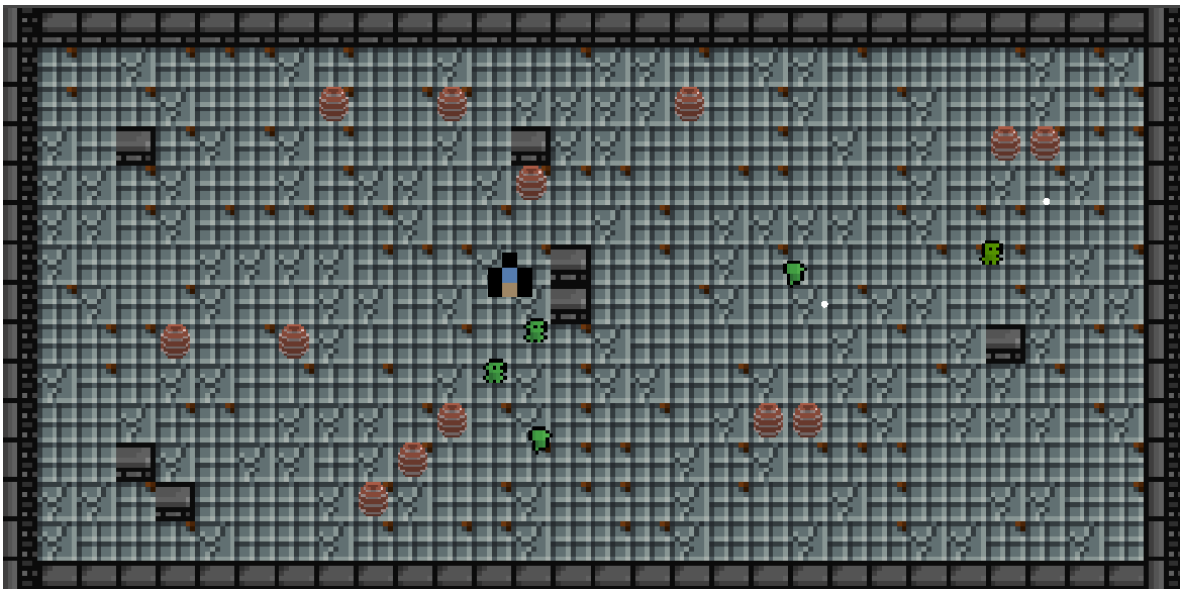

Рисунок Г.7 – Успішно згенерована кімната

Рисунок Г.8 – Завершення гри при смерті гравця

Рисунок Г.9 – Перенесення гравця через портал