МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет
імені Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних
інформаційних систем, д-р техн. наук, проф.
_______________Ю. П. Кондратенко
«____»_______2024 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

РОЗРОБКА 2D-ГРИ В ЖАНРІ ROGUELIKE З ДИНАМІЧНОЮ ГЕНЕРАЦІЄЮ РІВНІВ ЗАСОБАМИ РУШІЯ UNITY

Спеціальність 122 «Комп'ютерні науки»

122 – КРБ – 401.22010111

Виконав студент 4-го курсу, групи 401
_______________ *В. О. Кулеба*
«18» червня 2024 р.

Керівник: канд. техн. наук, доцент
_______________*В. Ю. Савінов*
«18» червня 2024 р.

Миколаїв – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет ім. Петра Могили
Факультет комп’ютерних наук
Кафедра інтелектуальних інформаційних систем

Рівень вищої освіти **бакалавр**
Спеціальність **122 «Комп’ютерні науки»**________________
(шифр і назва)
Галузь знань **12 «Інформаційні технології»**____________________
(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних
інформаційних систем, д-р техн. наук, проф.
______________ Ю. П. Кондратенко
«____» ______________2024 р.

З А В Д А Н Н Я
на виконання кваліфікаційної роботи

Видано студенту групи 401 факультету комп’ютерних наук Кулебі Владиславу Олександровичу.

1. Тема кваліфікаційної роботи «Розробка 2D-гри в жанрі roguelike з динамічною генерацією рівнів засобами рушія Unity».

Керівник роботи Савінов Володимир Юрійович, канд. техн. наук, доцент.

Затв. наказом Ректора ЧНУ ім. Петра Могили від «28» грудня 2023 р. № 271

2. Строк представлення кваліфікаційної роботи студентом «18» червня 2024 р.

3. Вхідні (початкові) дані до роботи: аналіз ринку ігор жанру roguelike, існуючі методики генерації рівнів, поточні тренди у використанні ігрових рушіїв.

Очікуваний результат: функціональний прототип 2D-гри roguelike з реалізацією динамічної генерації рівнів, розроблений з використанням Unity.

4. Перелік питань, що підлягають розробці (зміст пояснювальної записки):

– дослідження жанру roguelike, його характеристики та історія розвитку;
– огляд сучасних ігрових рушіїв та вибір Unity;
– методики процедурної генерації рівнів і їх придатність для жанру roguelike;

– розробка і тестування прототипу гри.

5. Перелік графічного матеріалу: презентація.

6. Завдання до спеціальної частини: «Загрози здоров'ю працівників IT-сфери та методи їх профілактики»

7. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис
Спеціальна частина з охорони праці	Алексєєва А. О., доцент кафедри екології	

Керівник роботи __канд. техн. наук, доцент, Савінов В. Ю.________
(наук. ступінь, вчене звання, прізвище та ініціали)

____
(підпис)

Завдання прийнято до виконання ___Кулеба В. О.______________
(прізвище та ініціали)

____
(підпис)

Дата видачі завдання «_14_» ______січня____ 2024 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: Розробка 2D-гри в жанрі roguelike з динамічною генерацією рівнів засобами рушія Unity

№	Найменування роботи	Початок	Закінчення	Примітки
1	Подання заяви на затвердження теми та керівників КРБ	10.11.2023	15.11.2023	Виконано
2	Отримання завдання на виконання КРБ	10.01.2024	15.01.2024	Виконано
3	Складання календарного плану роботи на весь період виконання КРБ	16.01.2024	30.01.2024	Виконано
4	Аналіз жанру roguelike, дослідження нових roguelike ігор	1.02.2024	15.02.2024	Виконано
5	Вибір технологій та інструментальних засобів розробки гри	16.02.2024	1.03.2024	Виконано
6	Реалізація та тестування гри	1.03.2024	15.04.2024	Виконано
7	Оформлення пояснювальної записки	15.04.2024	31.04.2024	Виконано
8	Проходження переддипломної практики, збір та аналіз матеріалів до КРБ	29.04.2024	11.05.2024	Виконано
9	Розробка звіту з переддипломної практики	12.05.2024	15.05.2024	Виконано
10	Розробка спеціальної частини з охорони праці	15.05.2024	18.05.2024	Виконано
11	Аналіз отриманих результатів, та дороблення пояснювальної записки	18.05.2024	26.06.2024	Виконано
12	Перший попередній захист КРБ на засіданні комісії кафедри	27.05.2024	27.05.2024	Виконано
13	Корегування пояснювальної записки після попереднього захисту	28.05.2024	09.06.2024	Виконано
14	Другий попередній захист КРБ на засіданні комісії кафедри	10.06.2024	10.06.2024	Виконано
15	Кінцеве оформлення слайдів та пояснювальної записки	10.06.2024	13.06.2024	Виконано
16	Подання КРБ рецензенту	13.06.2024	13.06.2024	Виконано
17	Подання КРБ, її електронної копії та інших документів (відгуку, рецензії) до захисту	17.06.2024	21.06.2024	Виконано
18	Захист КРБ перед екзаменаційною комісією (ЕК)	26.06.2024	26.06.2024	Виконано

Розробив студент __Кулеба В. О.__________________ _____________
(прізвище, ім'я, по батькові студента) (*підпис*)

Керівник роботи __канд. техн. наук, доцент. Савінов В. Ю _____________
(*посада, прізвище, ім'я, по батькові)* *(підпис)*

«_29_» _______01________ 2024 р.

АНОТАЦІЯ

кваліфікаційної роботи студента групи 401 ЧНУ ім. Петра Могили

Кулеби Владислава Олександровича

Тема: «Розробка 2D-гри в жанрі roguelike з динамічною генерацією рівнів засобами рушія Unity»

Кваліфікаційна робота присвячена розробці програмної реалізації 2D гри у жанрі roguelike на базі ігрового рушія Unity.

Об’єкт дослідження – процес розробки 2D-гри в жанрі roguelike з динамічною генерацією рівнів.

Предмет дослідження – ігрові механіки жанру roguelike та технології розробки на платформі Unity.

Метою роботи – аналіз ігрової індустрії та сучасних трендів у жанрі roguelike, а також розробка ігрового застосунку з динамічною генерацією рівнів.

Робота складається з фахової та спеціальної частини з охорони праці.

Пояснювальна записка фахової частини складається з вступу, трьох розділів, висновків та додатків.

Вступ містить огляд актуальності теми.

Перший розділ присвячений дослідженню жанру roguelike.

Другий розділ описує вибір технологічних рішень та обґрунтування вибору Unity.

Третій розділ розкриває процес проектування та реалізацію гри.

Висновки підсумовують результати дослідження та розробки

Бакалаврська кваліфікаційна робота містить 62 сторінки (без додатків), 48 рисунків, 1 таблицю, 28 джерел та 1 додаток.

Ключові слова: Roguelike, Unity, 2D гра, випадкова генерація рівнів, ігровий рушій, ігрові механіки, розробка ігор.

ABSTRACT

for bachelor's qualification work of a student of 401 group at Petro Mohyla Black Sea National University

Kuleba Vladyslav

Theme: Development of a 2D roguelike game with dynamic level generation using the Unity engine"

The qualification work is dedicated to the development of a software implementation of a 2D roguelike game based on the Unity game engine.

Object of work – the process of developing a 2D roguelike game with dynamic level generation.

Subject of work – game mechanics of the roguelike genre and development technologies on the Unity platform.

The purpose of this work – analysis of the gaming industry and current trends in the roguelike genre, as well as the development of a gaming application with dynamic level generation.

The work consists of a professional and a special part on labor protection.

The explanatory note of the professional part consists of an introduction, three chapters, conclusions and appendices.

The introduction provides an overview of the relevance of the topic.

The first chapter is devoted to the study of the roguelike genre.

The second chapter describes the choice of technological solutions and the rationale for choosing Unity.

The third section reveals the process of designing and implementing the game.

The conclusions summarise the results of the research and development.

The bachelor qualification work contains 62 pages (without appendices), 48 figures, 1 table, 28 sources and 1 appendix.

Keywords: Roguelike, Unity, 2D game, random level generation, game engine, game mechanics, game development.

ЗМІСТ

ВСТУП 3
1 ДОСЛІДЖЕННЯ ЖАНРУ ROGUELIKE 5
1.1 Історія створення ігор жанру roguelike 5
1.2 Характеристики жанру roguelike і геймплейні інновації 7
1.3 Аналіз наявних ігор та технологій 9
1.4 Вплив жанру на сучасні ігрові тренди. 13
Висновки до розділу 1 16
2 ДОСЛІДЖЕННЯ ТА ВИБІР ТЕХНОЛОГІЧНИХ РІШЕНЬ 17
2.1 Дослідження популярних ігрових рушіїв 17
2.2 Обґрунтування вибору рушія Unity для розробки гри і його переваги у порівнянні з іншими рушіями 20
2.3 Методи випадкової генерації рівнів 23
Висновок до розділу 2 28
3 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ГРИ 29
3.1 Визначення основних геймплейних механік для roguelike-гри 29
3.2 Створення системи керування для гри 31
3.3 Додавання ворогів та налаштування їх характеристик 35
3.4 Підготовка та реалізація алгоритму випадкового генерування рівня 39
3.5 Створення головного меню гри 46
3.6 Збір інформації з гри 50
3.7 Завершення створення гри 53
Висновок до розділу 3 57
ВИСНОВКИ 58
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 60
ДОДАТОК А Код генерації рівня 63

ВСТУП

Актуальність. У сучасному світі індустрія відеоігор є однією з найбільш динамічних і перспективних галузей цифрових технологій. Особливе місце серед численних жанрів відеоігор займають ігри жанру roguelike, які завдяки своїм унікальним характеристикам здобули широку популярність і любов гравців. Ігри цього жанру вирізняються процедурною генерацією рівнів, високим рівнем складності та постійною загрозою смерті персонажа, що змушує гравців розвивати стратегічне мислення та адаптуватися до непередбачуваних ситуацій.

Історія створення жанру roguelike розпочалася з випуску гри "Rogue" у 1980 році, яка стала визначальною для цього жанру. З того часу ігри типу roguelike постійно еволюціонували, інтегруючи нові технології та геймплейні інновації, що дозволило залучити широку аудиторію гравців. Сучасні ігри цього жанру, такі як "The Binding of Isaac", "Dead Cells" та "Hades", демонструють величезний потенціал для подальшого розвитку, поєднуючи в собі глибокі наративи, складні механіки та передові технології.

Мета роботи. Основною метою бакалаврської кваліфікаційної роботи є розробка 2D-гри в жанрі roguelike з динамічною генерацією рівнів засобами рушія Unity. Для досягнення цієї мети необхідно провести детальне дослідження жанру roguelike, обрати оптимальні технологічні рішення для розробки гри, а також реалізувати проект з урахуванням всіх необхідних геймплейних механік.

Завдання дослідження:

1) провести аналіз історії створення та розвитку ігор жанру roguelike;

2) визначити ключові характеристики жанру roguelike та геймплейні інновації, які зробили його популярним;

3) здійснити огляд існуючих ігор цього жанру та сучасних технологій, які використовуються для їх розробки;

4) оцінити вплив жанру roguelike на сучасні ігрові тренди;

5) провести дослідження популярних ігрових рушіїв та обґрунтувати вибір рушія Unity для розробки гри;

6) розробити метод випадкової генерації рівнів, який буде використовуватися в грі;

7) реалізувати проект гри з урахуванням визначених геймплейних механік та технологічних рішень.

Об'єкт дослідження. Процес розробки 2D-гри в жанрі roguelike з динамічною генерацією рівнів.

Предмет дослідження. Ігрові механіки жанру roguelike та технології розробки на платформі Unity.

Структура роботи. Відповідно до мети та завдань дослідження, бакалаврська кваліфікаційна робота складається з трьох розділів. Перший розділ присвячений дослідженню жанру roguelike, його історії, характеристик та впливу на сучасні ігрові тренди. У другому розділі розглядаються технологічні рішення для розробки гри, обґрунтовується вибір рушія Unity та описуються методи випадкової генерації рівнів. Третій розділ містить опис процесу проектування та реалізації гри, включаючи розробку геймплейних механік, систему керування персонажем, додавання ворогів та налаштування їх характеристик, а також реалізацію алгоритму випадкової генерації рівнів.

Висновки. У висновках підсумовуються результати дослідження та розробки гри, а також визначаються перспективи подальшого розвитку проекту.

1 ДОСЛІДЖЕННЯ ЖАНРУ ROGUELIKE

1.1 Історія створення ігор жанру roguelike

Roguelike це жанр відеоігор, що характеризується процедурно генерованими рівнями, постійною смертю персонажа та геймплеєм, що сильно залежить від дослідження підземель та тактичної боротьби. Назва жанру походить від гри "Rogue", яка вийшла у 1980 році і стала визначальною для цього типу ігор [1].

Ігри жанру roguelike здобули популярність завдяки своїм унікальним особливостям, таким як процедурно генеровані світи, постійна загроза смерті персонажа та високий рівень складності. Ці ігри часто змушують гравців адаптуватися та розвивати стратегії виживання в непередбачуваних умовах. Розглянемо ключові етапи розвитку цього жанру.

Перша roguelike-гра Rogue.

Жанр roguelike з'явився на початку 1980-х років із створенням гри "Rogue" студентами університету Майклом Той і Гленні Вічменом. Спочатку розроблена для систем UNIX, гра "Rogue" поєднувала елементи традиційних рольових ігор з комп'ютерними технологіями, ставлячи перед гравцями завдання вижити в лабіринтах, наповнених монстрами та скарбами. Відмінною рисою гри було процедурне генерування рівнів, що забезпечувало унікальний макет і нові виклики кожного разу, коли гравець починав нову гру. Це робило кожен ігровий сеанс неповторним, адже підземелля, монстри та скарби змінювалися щоразу. Гравцям належало шукати Амулет Єндора на найнижчому рівні підземелля, знищуючи ворогів і збираючи корисні предмети, такі як зброя, броня, зілля та інше [2].

На рисунку 1.1 зображено скріншот гри Rogue.

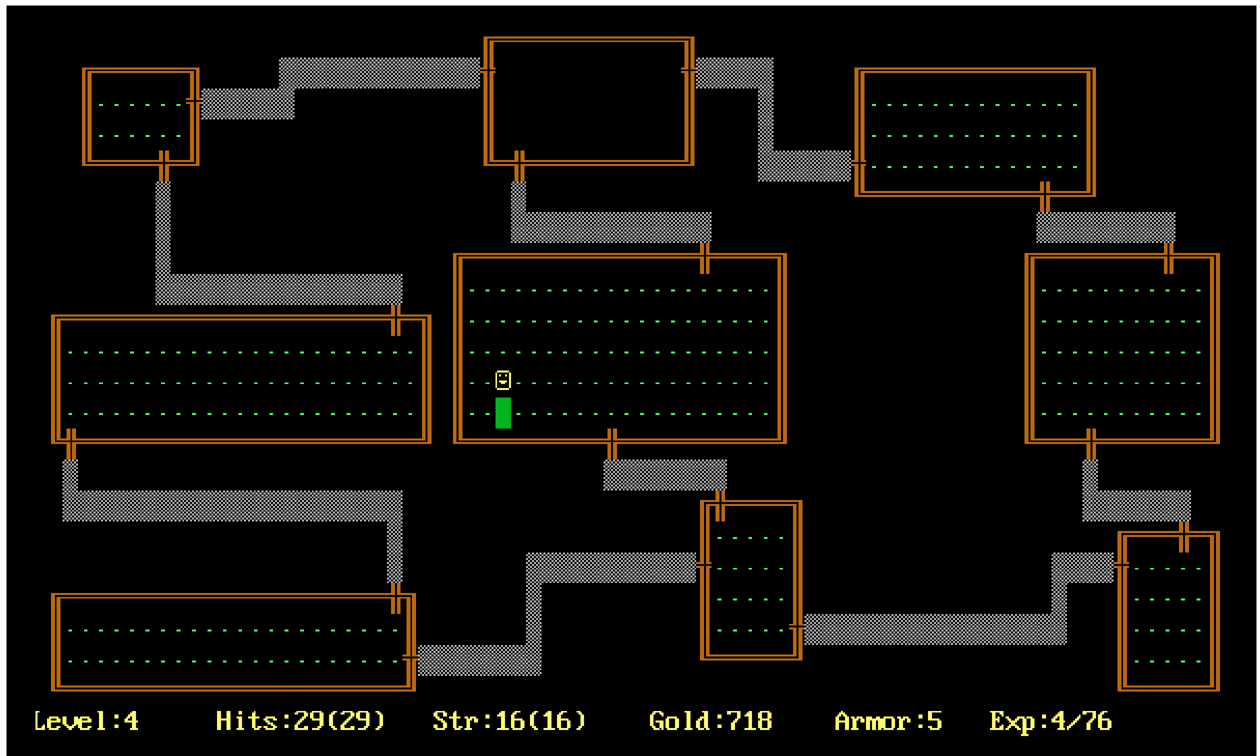

Рисунок 1.1 – Скріншот гри Rogue [2]

Успіх "Rogue" поклав початок розвитку цілої ніші ігор, що відзначалися схожими характеристиками. Відомі ігри, такі як "Hack" (1985) і "NetHack" (1987), розширили ідеї "Rogue", додаючи нові функції та більшу складність. "NetHack" відзначається своїм глибоким сюжетом та складністю, і вона продовжує бути популярною серед фанатів жанру до сьогодні.

Розвиток технологій та нові можливості.

З розвитком комп'ютерних технологій, ігри жанру roguelike стали складнішими та візуально привабливішими. Починаючи з 2000-х років, з'явилися ігри, які використовували більш сучасні графічні і аудіо можливості. "Dungeon Crawl Stone Soup", "The Binding of Isaac" (2011), та "Spelunky" (2008) змогли залучити більшу аудиторію завдяки оновленому інтерфейсу та інтеграції графічних і звукових ефектів.

Сучасні тенденції.

Сьогодні жанр roguelike продовжує еволюціонувати, інтегруючи елементи з інших жанрів та пропонуючи все нові й нові механіки. Ігри, такі як "Hades" (2020), показали, як глибоке наративне розповідання може бути інтегровано в рамках

roguelike-структури, забезпечуючи імпульс для розвитку персонажів та збереження захоплення гравців. Постійна загроза смерті та необхідність кожного разу починати заново не відштовхують, а, навпаки, додають іграм динамізму та виклику.

Жанр roguelike залишається одним із найбільш інноваційних у світі відеоігор, продовжуючи захоплювати гравців своєю непередбачуваністю та складністю. Ігри цього жанру часто вимагають від гравців глибокого занурення, стратегічного мислення та готовності до невдач, що робить їх незабутнім досвідом у світі інтерактивних розваг.

1.2 Характеристики жанру roguelike і геймплейні інновації

Жанр roguelike вирізняється своїми унікальними характеристиками, які створюють глибокий і викликаючий інтерес геймплейний досвід. Ці ігри вимагають від гравців стратегічного мислення, вміння швидко адаптуватися до змінних умов і готовності до постійних викликів. У цьому розділі розглянемо основні характеристики жанру roguelike та геймплейні інновації, які зробили його таким популярним.

Важливу роль відіграє ігровий дизайн, який є процесом створення та формування механік, систем і правил гри. Процес розробки гри зазвичай включає концептуалізацію, створення прототипів, тестування та ревізію. У випадку roguelike-ігор, особливу увагу приділяють механікам процедурної генерації та балансу складності, щоб забезпечити непередбачуваність та повторюваність геймплею. Це включає постійне тестування і вдосконалення, щоб забезпечити збалансованість ігрових рівнів та цікавий досвід для гравців [3].

Основні характеристики жанру roguelike.

Процедурно генеровані світи.

Однією з ключових особливостей roguelike-ігор є процедурна генерація світів. Кожен запуск гри створює новий унікальний світ з випадковими підземеллями, монстрами та скарбами. Це забезпечує безкінечну варіативність і реіграбельність, адже гравці ніколи не знають, що чекає на них за наступним

поворотом [4].

Перманентна смерть.

Перманентна смерть або постійна смерть означає, що після смерті персонажа гравець повинен починати гру спочатку. Це додає грі значної напруги і змушує гравців обережно підходити до кожного кроку, адже одна помилка може коштувати всього прогресу [5].

Покроковий геймплей.

Більшість класичних roguelike-ігор використовують покроковий режим, де кожна дія гравця виконується в рамках одного кроку. Це дозволяє гравцям ретельно планувати свої дії і обдумувати стратегії, що особливо важливо в складних ситуаціях [4].

Roguelike-ігри зазвичай мають складні механіки і високий рівень складності. Гравці повинні враховувати безліч факторів, таких як ресурси, здоров'я, інвентар, та тактичні маневри для того, щоб вижити в небезпечних умовах.

Геймплейні інновації.

Гібридні жанри. У сучасних roguelike-іграх часто поєднуються елементи з інших жанрів. Наприклад, "The Binding of Isaac" комбінує елементи action та roguelike, створюючи динамічний ігровий процес, що спирається на швидкі рефлекси гравця. Інший приклад, "Darkest Dungeon", поєднує roguelike з елементами стратегічних ігор та менеджменту ресурсів.

Мета-прогресія. У класичних roguelike-іграх після смерті гравця весь прогрес губиться. Проте сучасні ігри, такі як "Rogue Legacy" і "Hades", запровадили мета-прогресію, де певні досягнення чи ресурси зберігаються між спробами. Це забезпечує відчуття прогресу і мотивації навіть після невдачі [4].

Покращена графіка і звук. Хоча традиційні roguelike-ігри мали просту графіку, сучасні розробники впроваджують покращені візуальні і звукові ефекти. Це допомагає залучити ширшу аудиторію і робить ігри більш захоплюючими. Наприклад, "Hades" відзначається чудовою графікою та динамічним саундтреком, що додає глибини ігровому досвіду.

Складні наративи і персонажі. Деякі сучасні roguelike-ігри інтегрують глибокі наративи і розвиток персонажів у свій геймплей. Наприклад, у "Hades" гравці взаємодіють із різними персонажами, розкриваючи їхні історії і розвиваючи стосунки. Це додає додатковий шар занурення і мотивації для гравців.

Жанр roguelike зазнав значних змін з моменту свого створення, зберігаючи при цьому основні характеристики, що роблять його унікальним. Інновації в геймплеї, поєднання з іншими жанрами, поліпшення графіки і звуку, а також розширені наративи зробили ці ігри популярними серед різноманітної аудиторії. Незалежно від того, чи це класичні roguelike з ASCII-графікою, чи сучасні гібриди з багатим сюжетом, ці ігри продовжують захоплювати гравців своїм викликом і непередбачуваністю.

1.3 Аналіз наявних ігор та технологій

Сучасні roguelike ігри продемонстрували значний прогрес у використанні новітніх технологій та ігрових механік, особливо у таких популярних іграх як "The Binding of Isaac: Rebirth", "Dead Cells", "Hades" та "Risk of Rain 2". Ці ігри ілюструють як інновації у геймплеї, так і впровадження передових технологій можуть змінити відчуття і досвід гри в жанрі roguelike.

The Binding of Isaac: Rebirth.

The Binding of Isaac: Rebirth — це переробка оригінальної гри "The Binding of Isaac", яка вже сама по собі була культовою. Переродження пропонує значні поліпшення в графіці та процесі гри, використовуючи новий ігровий двигун, що дозволило розробникам включити більш складні механіки і значно покращити візуальні ефекти. Гра відома своєю тематикою та символізмом, а також складною системою предметів, яка значно впливає на стратегію ігрового процесу. Процедурно генеровані рівні та елементи roguelike-гри гарантують, що кожна гра буде унікальною [6].

На рисунку 1.2 зображено фрагмент ігрового процесу гри The Binding of Isaac: Rebirth.

Рисунок 1.2 – Фрагмент ігрового процесу гри The Binding of Isaac: Rebirth

Dead Cells.

Dead Cells відома своєю "roguevania" структурою, яка поєднує roguelike механіки з елементами metroidvania. Це означає, що ігровий світ має структуровану неперервність, в той час як рівні є процедурно генерованими. Гра має високу динаміку, відмінне управління персонажем та широкий арсенал зброї, що додає глибину ігровому процесу. Dead Cells також вирізняється своєю плавною анімацією та яскравою піксельною графікою, що є значним досягненням в жанрі [7].

На рисунку 1.3 зображено Фрагмент ігрового процесу гри Dead Cells.

Рисунок 1.3 – Фрагмент ігрового процесу гри Dead Cells

Hades.

Hades є яскравим прикладом сучасного підходу до жанру roguelike, в якому кожна спроба пройти гру супроводжується розгортанням глибокого сюжету. Гра, розроблена Supergiant Games, розповідає історію Загрея, безсмертного сина Аїда, який прагне втекти з підземного світу, борючись із численними ворожими душами. Відзначаючись чудовою графікою, різноманітністю геймплею та ретельно розробленими персонажами, Hades використовує сучасні технології для створення плавних анімацій та візуальних ефектів, що значно підсилює загальну привабливість ігрового процесу [8].

На рисунку 1.4 зображено фрагмент ігрового процесу гри Hades.

Рисунок 1.4 – Фрагмент ігрового процесу гри Hades

Risk of Rain 2.

Переходячи від 2D до 3D, Risk of Rain 2 змінює стандартні уявлення про roguelike ігри. 3D-середовище вносить нову глибину і комплексність у жанр, дозволяючи гравцям досліджувати більш об'ємні світи і використовувати більш складні стратегії. Гра також включає мультиплеєр, що не часто зустрічається в традиційних roguelike іграх, дозволяючи гравцям об'єднуватися та спільно долати виклики [9].

На рисунку 1.5 зображено фрагмент ігрового процесу гри Risk of Rain 2.

Рисунок 1.5 – Фрагмент ігрового процесу гри Risk of Rain 2

Сучасні технології та інноваційні підходи до геймплею в значній мірі збагатили жанр roguelike. Від переробок класичних ігор до введення нових механік і розширення жанрових меж — ігри, як "The Binding of Isaac: Rebirth", "Dead Cells", "Hades" та "Risk of Rain 2", демонструють потенціал для подальшого розвитку та інновацій. Ці ігри не тільки забезпечують захоплюючий ігровий досвід, але й спонукають до рефлексії над тим, як ігри можуть розвиватися та впливати на гравців у майбутньому.

1.4 Вплив жанру на сучасні ігрові тренди.

Жанр roguelike став одним із найбільш впливових напрямків у сучасній ігровій індустрії, вносячи вагомий внесок у розробку ігор та формування трендів. Від ігор із незначними інді-проектами до великих AAA-видань, roguelike-елементи стали ключовими у визначенні нових напрямків розвитку ігрового процесу і дизайну. Розглянемо детальніше, як саме цей жанр вплинув на ігрові тренди.

Збільшення складності ігор та вимог до стратегічного мислення.

Однією з вирішальних характеристик жанру roguelike є висока складність ігрового процесу, яка вимагає від гравців продуманих тактичних рішень і

стратегічного планування. Сучасні ігри все частіше інтегрують цей аспект, створюючи більш викликаючі та задовільні ігрові досвіди. Це спонукає гравців до глибшого занурення в ігрові механіки та довше утримує їхню увагу.

Процедурна генерація контенту.

Процедурно генеровані рівні та ігрові світи, які є ключовими для roguelike-ігор, стали популярними в інших жанрах. Вони дозволяють створювати унікальний ігровий досвід при кожному новому проходженні, збільшуючи реіграбельність і довговічність ігор. Ця техніка використовується не тільки в пригодницьких і рольових іграх, але й у стратегіях та симуляторах [10].

На рисунку 1.6 зображено генерацію рівня бінарним розбиттям простору.

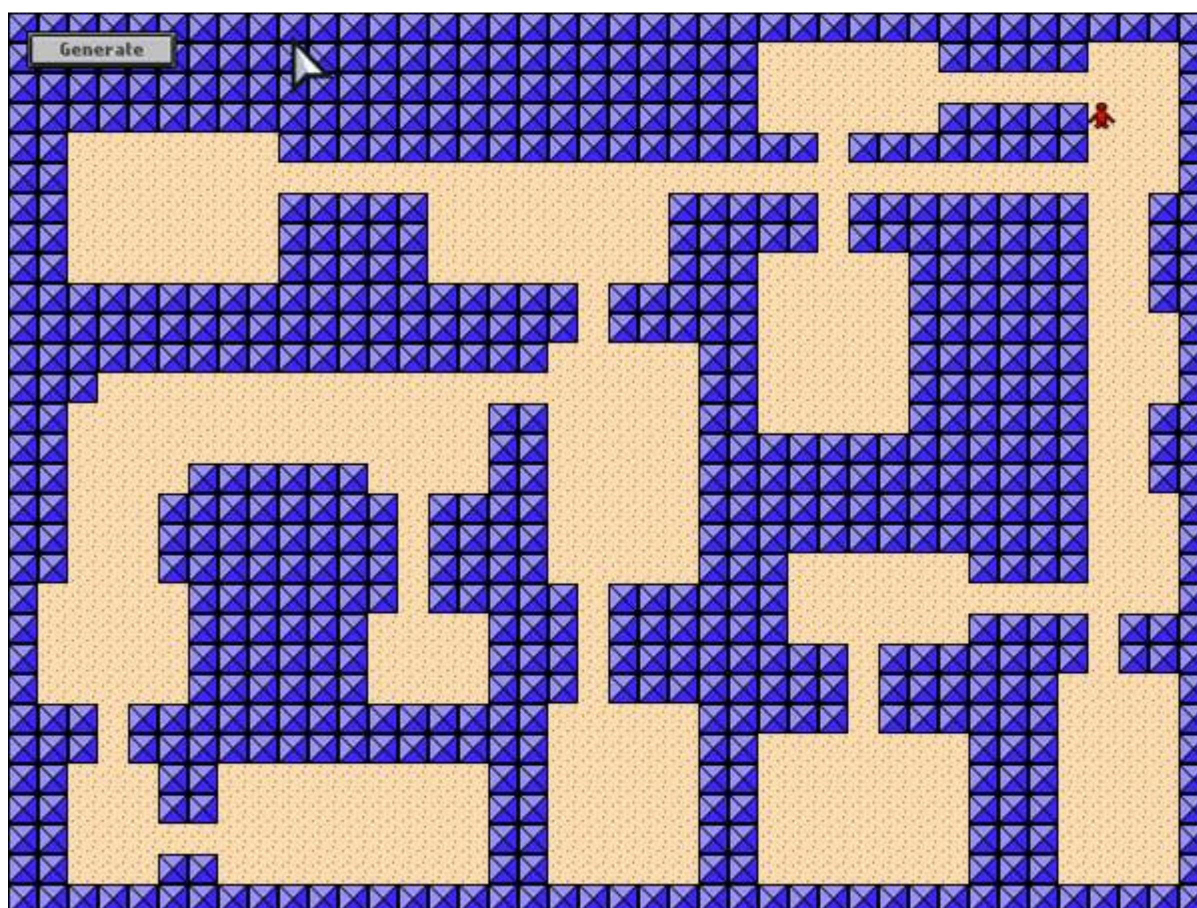

Рисунок 1.6 – Генерація рівня бінарним розбиттям простору

Популяризація безповоротної смерті.

Концепція безповоротної смерті, що означає втрату всіх досягнень при смерті персонажа, змінила підходи до дизайну ігрового процесу. Ця характеристика

стимулює гравців до вибору більш обачливих та виважених дій, роблячи кожну спробу значущою. Впровадження такого підходу в ігри інших жанрів збільшує емоційний вплив на гравців, змушуючи їх більш відповідально ставитися до своїх рішень [10].

На рисунку 1.7 зображено екран смерті в грі Hades.

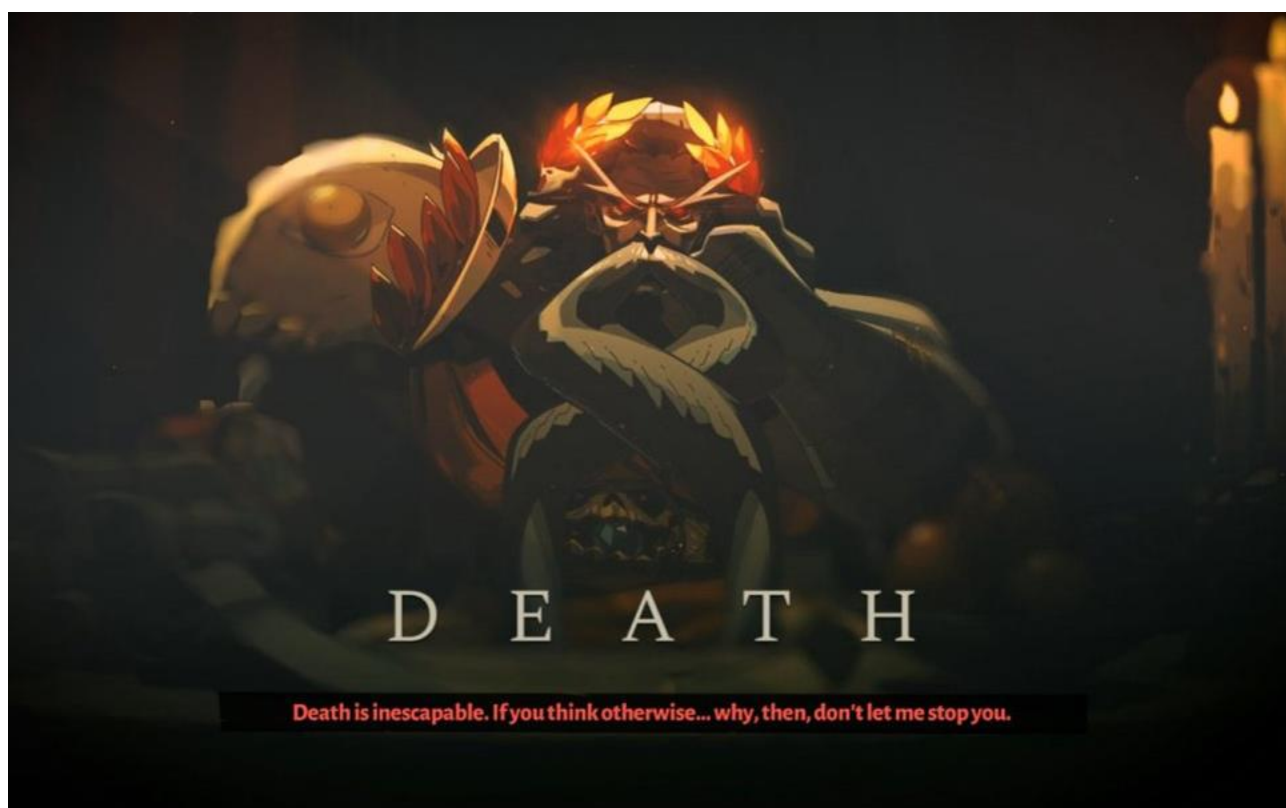

Рисунок 1.7 – Екран смерті в грі Hades

Розширення жанрових меж.

Roguelike-ігри часто поєднують елементи з різних жанрів, створюючи нові унікальні комбінації, які можуть включати RPG, стратегії, шутери, і навіть головоломки. Це сприяло виникненню гібридних жанрів, які пропонують гравцям нові виклики та способи взаємодії з ігровим світом.

Вплив на ігрову індустрію та культуру.

Зростаюча популярність жанру roguelike сприяла формуванню активних онлайн-спільнот і фан-баз, що сприяє обміну ідеями, стратегіями та модами між гравцями. Це не тільки збільшує взаємодію в рамках спільноти, але й стимулює розвиток ігрового контенту та інновацій.

Вплив жанру roguelike на сучасні ігрові тренди є значним та має далекосяжні наслідки. Інновації в геймплеї, а також філософія дизайну, яка стимулює інновації та креативність, перетворили цей жанр на важливий елемент сучасної ігрової культури. Roguelike-ігри продовжують впливати на розробку нових ігрових проектів, розширюючи можливості та визначаючи нові напрямки у розвитку ігрової індустрії.

Висновки до розділу 1

Жанр roguelike, відомий своїми процедурно генерованими світами, перманентною смертю персонажів та високою складністю, почався з гри "Rogue" у 1980 році. Ігри цього жанру, такі як "Hack" та "NetHack", розвивали ці ідеї, а сучасні приклади на кшталт "The Binding of Isaac" і "Hades" продемонстрували, як можна впроваджувати глибокі наративи та сучасну графіку. Основні характеристики жанру включають непередбачуваність і постійний виклик, що вимагають від гравців стратегічного мислення та адаптивності. Інновації, такі як гібридні жанри, мета-прогресія та покращена графіка, зберігають інтерес до roguelike ігор. Ці ігри демонструють значний прогрес у використанні новітніх технологій і впливають на сучасні ігрові тренди, підвищуючи складність і популяризуючи процедурну генерацію контенту. Roguelike-елементи інтегруються в інші жанри, створюючи нові гібридні форми, які продовжують розвивати індустрію відеоігор, роблячи цей жанр одним із найбільш інноваційних та впливових у сучасній ігровій культурі.

2 ДОСЛІДЖЕННЯ ТА ВИБІР ТЕХНОЛОГІЧНИХ РІШЕНЬ

2.1 Дослідження популярних ігрових рушіїв

У сучасному світі відеоігор, ігрові рушії відіграють вирішальну роль, оскільки вони є фундаментом для розробки ігор, надаючи розробникам необхідні інструменти та функціонал для створення ігрових світів. Деякі ігрові рушії стали особливо популярними завдяки своїй гнучкості, потужності та доступності. В цьому дослідженні розглянемо найпопулярніші ігрові рушії, такі як Unity, Unreal Engine, Godot, та GameMaker Studio, та оцінимо їх вплив на ігрову індустрію.

Unity.

Unity один з найбільш вживаних ігрових рушіїв у світі, відомий своєю універсальністю та підтримкою мультиплатформенності. Рушій дозволяє розробникам створювати ігри для майже будь-якої платформи, включаючи PC, консолі, мобільні пристрої та навіть VR-системи. Unity особливо популярний серед інді-розробників завдяки своїй легкості у використанні, обширній документації та активній спільноті, яка постійно ділиться порадами, плагінами та навчальними матеріалами. Unity також відомий своїми потужними інструментами для анімації та фізики, що робить його ідеальним вибором для створення як 2D, так і 3D ігор [11].

На рисунку 2.1 зображено логотип ігрового рушія Unity.

Рисунок 2.1 – Логотип Unity [11]

Unreal Engine.

Unreal Engine, розроблений компанією Epic Games, є одним з найпотужніших ігрових рушіїв, який використовується як великими студіями, так і інді-розробниками. Цей рушій відомий своїми передовими графічними можливостями, зокрема, підтримкою високоякісного рендерингу, реалістичної фізики та динамічного освітлення. Unreal Engine є вибором для проектів, які вимагають вражаючої візуалізації та інтенсивної інтеракції з ігровим середовищем. Нещодавні оновлення рушія, такі як введення Unreal Engine 5, засвідчують його здатність до інновацій та адаптації до сучасних технологічних вимог [12].

На рисунку 2.2 зображено логотип ігрового рушія Unreal Engine.

Рисунок 2.2 – Логотип Unreal Engine [12]

Godot.

Godot є відкритим ігровим рушієм, який швидко набуває популярності серед розробників завдяки своїй повній відкритості та гнучкості. Він підтримує розробку як 2D, так і 3D ігор та вирізняється інтуїтивно зрозумілим інтерфейсом та низьким порогом входження. Godot особливо привабливий для тих, хто шукає повний контроль над своїм ігровим двигуном без ліцензійних зборів або обмежень комерційного використання [13].

На рисунку 2.3 зображено логотип ігрового рушія Godot.

Рисунок 2.3 – Логотип Godot [13]

GameMaker Studio.

GameMaker Studio, розроблений YoYo Games, є ідеальним вибором для новачків у галузі ігрової розробки, які хочуть швидко створити ігри. Цей рушій зосереджений на розробці 2D ігор і вирізняється простотою використання, що не вимагає глибоких знань програмування завдяки візуальному програмуванню та "drag-and-drop" інтерфейсу. GameMaker широко використовується для створення незалежних ігор з невеликим бюджетом і є популярним вибором серед освітніх закладів для навчання основам ігрової розробки [14].

На рисунку 2.4 зображено логотип ігрового рушія GameMaker Studio.

Рисунок 2.4 – Логотип GameMaker Studio [14]

Кожен з цих ігрових рушіїв має свої унікальні характеристики, що робить його ідеальним для певних типів ігрових проектів та розробників. Від відкритих джерел до комерційних продуктів, від простих інструментів для новачків до комплексних систем для професіоналів — сучасні ігрові рушії пропонують розмаїття опцій, яке допомагає розробникам реалізувати свої творчі візії. Завдяки цим технологіям ігрова індустрія продовжує розвиватися, вносячи нові інновації та покращуючи ігровий досвід для гравців по всьому світу.

2.2 Обґрунтування вибору рушія Unity для розробки гри і його переваги у порівнянні з іншими рушіями

Розробка ігор — це складний процес, що потребує використання відповідних інструментів і технологій. Вибір ігрового рушія є одним з найважливіших рішень, яке впливає на всі аспекти створення гри, від розробки до випуску і підтримки. У цьому тексті розглянемо обґрунтування вибору рушія Unity для розробки 2D-гри в жанрі roguelike з динамічною генерацією рівнів та його переваги у порівнянні з іншими рушіями.

Unity широко відомий своєю універсальністю та підтримкою як 2D, так і 3D-графіки. Для розробки 2D-гри в жанрі roguelike Unity пропонує потужні інструменти, такі як Sprite Editor, Tilemap Editor та 2D Physics Engine. Sprite Editor дозволяє легко створювати та редагувати спрайти, Tilemap Editor спрощує процес створення рівнів, а 2D Physics Engine забезпечує реалістичну фізику для інтерактивних елементів гри.

Однією з ключових особливостей roguelike-ігор є процедурна генерація рівнів. Unity пропонує розробникам потужні інструменти та бібліотеки для реалізації процедурної генерації. Використовуючи скрипти на C#, розробники можуть створювати алгоритми, які генерують унікальні рівні під час кожного запуску гри, забезпечуючи гравцям новий ігровий досвід при кожній спробі.

Unity відомий своєю гнучкістю та можливістю розширення. Завдяки великій кількості плагінів та доповнень, доступних у Unity Asset Store, розробники можуть додавати нові функції та інструменти без необхідності писати код з нуля. Це значно зменшує час розробки та дозволяє зосередитися на унікальних аспектах гри [15].

Порівняння з Unreal Engine.

Unreal Engine відомий своїми передовими графічними можливостями та потужністю, однак він більш орієнтований на розробку 3D-ігор. Для розробки 2D-ігор Unity є більш підходящим вибором через спеціалізовані інструменти для роботи з 2D-графікою. Крім того, Unity має нижчий поріг входження, що важливо для інді-розробників та невеликих студій [16].

Порівняння з Godot.

Godot є відкритим рушієм з великою гнучкістю та підтримкою 2D та 3D-графіки. Однак, Unity має значно більшу спільноту та більш розвинену екосистему, що забезпечує доступ до широкого спектра ресурсів, таких як документація, форуми та навчальні матеріали. Це може бути вирішальним фактором для розробників, які потребують підтримки та швидкого вирішення проблем [16].

Порівняння з GameMaker Studio.

GameMaker Studio спеціалізується на розробці 2D-ігор та має простий інтерфейс, що робить його популярним серед новачків. Проте, Unity пропонує більшу гнучкість та розширюваність завдяки своїй підтримці C# і потужним інструментам для роботи з 2D та 3D-графікою. Unity також краще підходить для складніших проектів, які можуть вимагати інтеграції з іншими системами та платформами [16].

Переваги Unity для розробки roguelike-ігор.

Однією з ключових переваг Unity для розробки roguelike-ігор є можливість детального налаштування ігрового процесу. Рушій дозволяє створювати складні механіки, такі як перманентна смерть, випадкова генерація ігрових предметів, складні AI для ворогів та інтерактивні об'єкти. Розробники можуть швидко тестувати та впроваджувати нові ідеї завдяки швидкому циклу розробки та налагодження.

Unity підтримує широкий спектр платформ, включаючи PC, консолі, мобільні пристрої та VR-системи. Це дозволяє розробникам розширювати аудиторію своєї гри, випускаючи її на різних платформах без значних додаткових витрат. Для roguelike-ігор, які можуть мати широку базу шанувальників на різних пристроях, це є значною перевагою.

Однією з найбільших переваг Unity є його велика спільнота розробників. Це забезпечує доступ до численних ресурсів, включаючи форуми, відеоуроки, блоги та офіційну документацію. Розробники можуть швидко знаходити відповіді на свої запитання, обмінюватися досвідом та отримувати підтримку від колег.

Було сформовано таблицю 2.1 порівняння ігрових рушіїв використовуючи

такі електронні ресурси [11, 12, 13, 14, 15, 16].

Таблиця 2.1 – Порівняння основних характеристик ігрових рушіїв

Характеристика	Unity	Unreal Engine	Godot	GameMaker Studio
Платформи	ПК, мобільні, консолі, VR	ПК, мобільні, консолі, VR	ПК, мобільні, консолі	ПК, мобільні, консолі
Підтримка графіки	2D і 3D	Високоякісна 3D, базова 2D	2D і 3D	Високоякісна 2D, базова 3D
Доступність для новачків	Висока, з великою кількістю ресурсів	Середня, складніший для навчання	Висока, відкрите джерело	Висока, візуальне програмування
Мови програмування	C#	C++, Blueprints (візуальне програмування)	GDScript, C#, C++	GameMaker Language (GML)
Ресурси для навчання	Багато офіційних та спільнотних ресурсів	Багато офіційних та спільнотних ресурсів	Обмеженіше, але зростаюча спільнота	Хороші офіційні ресурси
Вартість	Безкоштовний для освіти і невеликих проектів, відсоток від доходу за великі проекти	Безкоштовний до досягнення певного доходу, потім відсоток доходу	Повністю безкоштовний	Безкоштовний базовий план, платні розширення

Рушій Unity є ідеальним вибором для розробки 2D-гри в жанрі roguelike з динамічною генерацією рівнів. Його потужні інструменти для роботи з 2D-графікою, підтримка процедурної генерації, гнучкість та велика спільнота роблять його привабливим варіантом для розробників. Порівняно з іншими рушіями, такими як Unreal Engine, Godot та GameMaker Studio, Unity пропонує оптимальний баланс між потужністю, універсальністю та зручністю використання, що забезпечує ефективний та продуктивний процес розробки гри.

2.3 Методи випадкової генерації рівнів

У розробці 2D-ігор жанру roguelike одним із ключових аспектів є динамічна генерація рівнів, що дозволяє створювати унікальний ігровий досвід при кожному новому запуску гри. Різноманітні методи генерації рівнів допомагають досягти високого рівня реіграбельності та забезпечують непередбачуваність, яка є важливою характеристикою жанру roguelike. У цьому тексті розглянемо чотири популярні методи генерації рівнів: метод розподілу кімнат, метод поділу простору (BSP), метод клітинних автоматів, та граф на основі кімнат.

Метод розподілу кімнат (Room Placement).

Метод розподілу кімнат передбачає створення кімнат у випадково обраних локаціях на мапі, з подальшим їх з'єднанням коридорами. Цей метод часто використовується у розробці класичних dungeon crawlers, де кожна кімната може мати різні розміри та форми. Unity дозволяє легко реалізувати такий тип генерації за допомогою своїх інструментів для роботи з тайлами та спрайтами, що робить можливим детальне налаштування кожної кімнати та коридору [17].

На рисунку 2.5 зображено генерацію рівня методом розподілу кімнат.

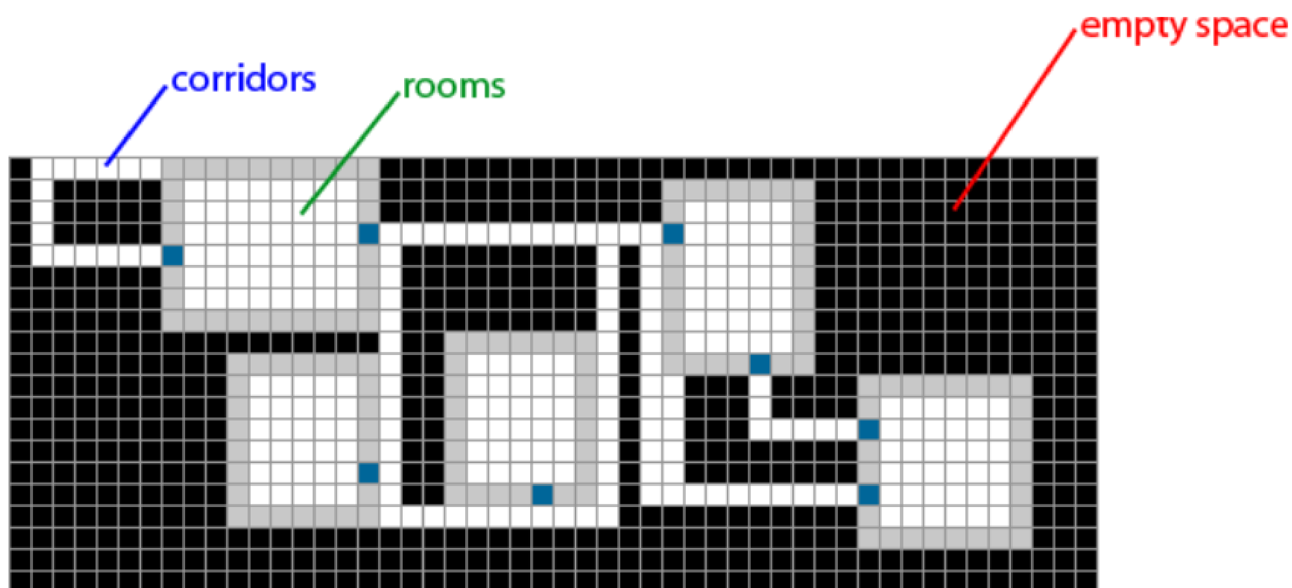

Рисунок 2.5 – Генерація рівня методом розподілу кімнат

Метод поділу простору (BSP - Binary Space Partitioning).

Метод Binary Space Partitioning (BSP) є не простим підходом, що починається з рекурсивного поділу ігрового простору на менші частини. Процес починається зі створення прямокутного підземелля, заповненого стінами, яке потім поділяється до розміру приблизної кімнати. Кожен поділ здійснюється випадковим вибором напряму (горизонтального чи вертикального) і позиції для поділу, що дозволяє уникнути поділу біля країв підземелля. Кінцеві поділи створюють "листки", де генеруються кімнати.

Після створення кімнат в кожному листку, вони з'єднуються коридорами. Якщо стіни кімнат розташовані один навпроти одного, використовується прямий коридор; інакше створюється коридор у формі z. Процес повторюється на вищих рівнях дерева, з'єднуючи батьківські під регіони між собою.

Цей підхід забезпечує високий контроль над розподілом простору, дозволяючи створювати більш структуровані та збалансовані рівні. Завдяки BSP дереву, кімнати не перекриваються, що сприяє різноманітності й інтерактивності гри, забезпечуючи шлях між усіма кімнатами [18].

На рисунку 2.6 зображено генерацію рівня методом поділу простору.

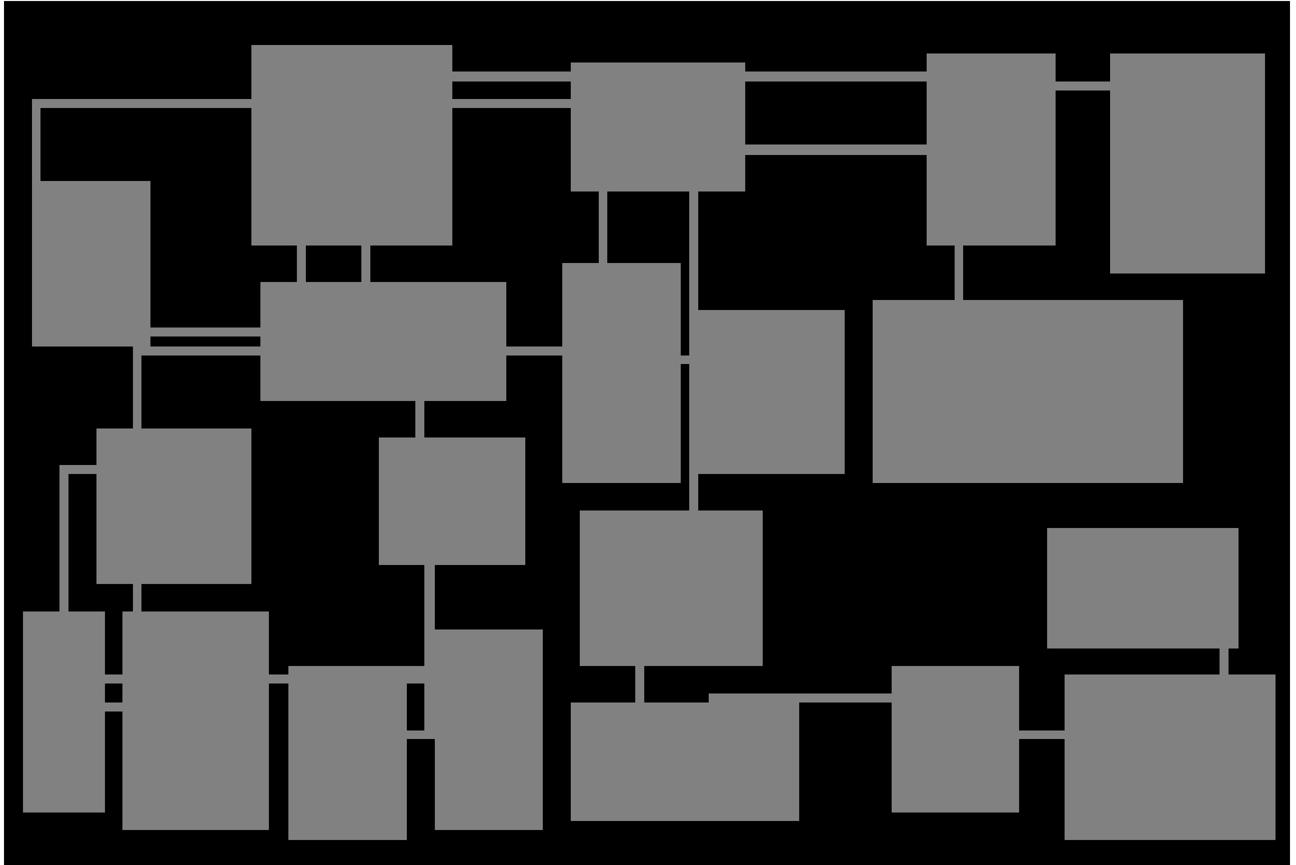

Рисунок 2.6 – Генерація рівня методом поділу простору

Метод клітинних автоматів (Cellular Automata).

Метод клітинних автоматів ґрунтується на моделі, де кожна клітина на сітці має стан, який змінюється відповідно до станів сусідніх клітин. Цей метод є ефективним для створення природних печер або складних лабіринтів. Використання клітинних автоматів у Unity дозволяє генерувати різноманітні текстури та топології рівнів, які виглядають натуралістично і мають органічні форми. Наприклад, для генерації печер можна використовувати сітку клітин, де кожна клітина може бути або стіною, або проходом. Процес генерації полягає в застосуванні правил, які визначають зміну стану клітини на основі стану її сусідів, що створює складні і неповторні структури печер [19].

На рисунку 2.7 зображено генерацію рівня методом клітинних автоматів.

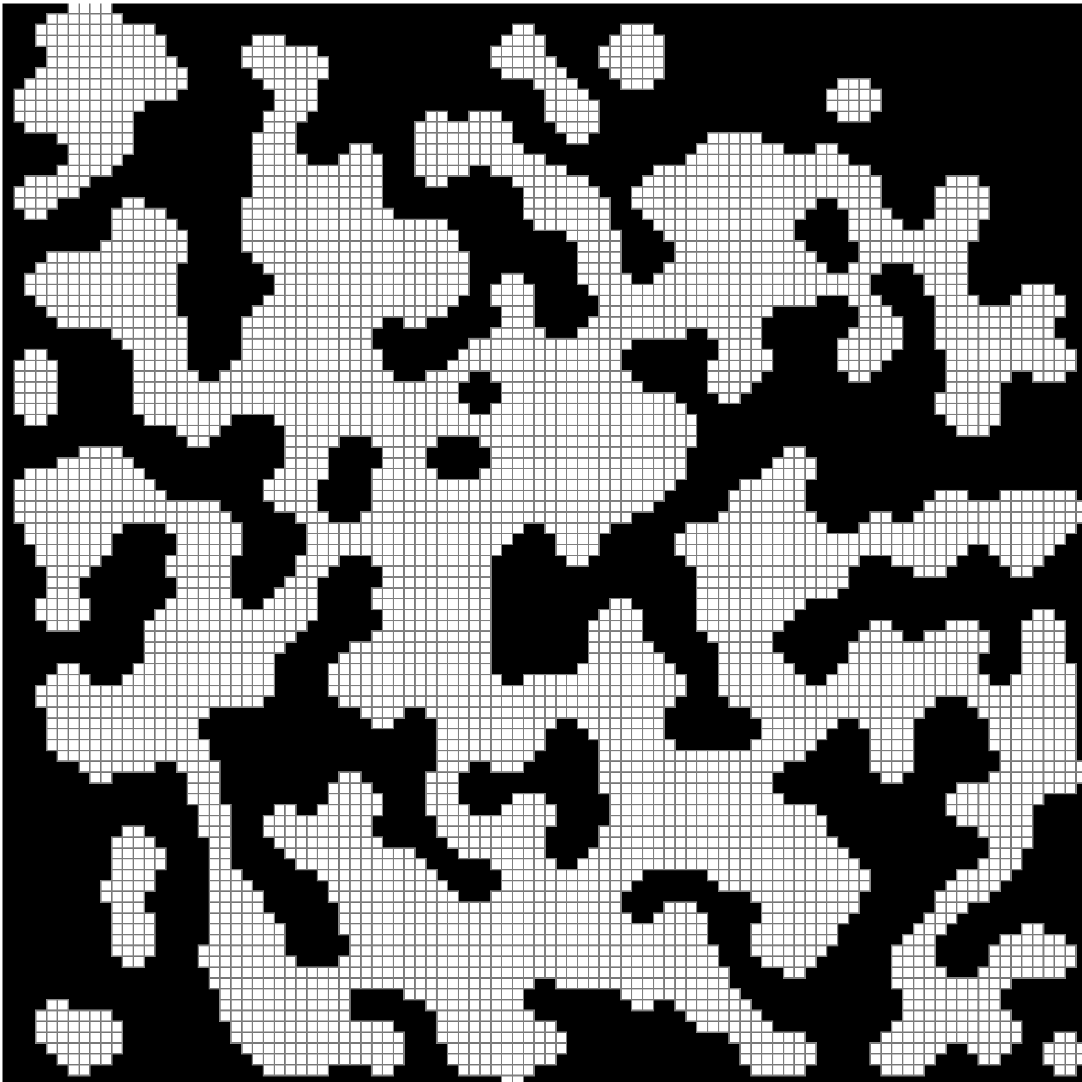

Рисунок 2.7 – Генерація рівня методом клітинних автоматів [19]

Граф на основі кімнат (Graph-based Room Generation).

Граф, побудований на основі кімнат, застосовує теорію графів для створення карт, де кімнати — це вершини, а коридори — ребра. Це дозволяє створювати карти з чіткою структурою та логічним плануванням. У Unity використання графів допомагає розробляти складні рівні з оптимізованими коридорами, підтримуючи плавний геймплей.

Алгоритм генерації рівнів використовує набір шаблонів кімнат і граф зв'язності рівня. Вершини графа представляють кімнати, а ребра визначають зв'язки між ними. Мета алгоритму — призначити форму та позицію кожній кімнаті, щоб вони не перетиналися і були з'єднані дверима. Граф зв'язності дозволяє дизайнерам впливати на потік геймплею, створюючи основні та додаткові шляхи. Алгоритм

підтримує цикли у графах рівня, що не часто зустрічається через складність реалізації. Також можна задавати різні форми кімнат для різних приміщень, як кімната з босом чи стартова кімната. Крім того, алгоритм підтримує явне задання позицій дверей, що дає дизайнерам контроль над входами в кімнати. Якщо потрібно з'єднати кімнати коридорами, потрібно додати нові вузли між сусідніми кімнатами, що ускладнює завдання, але є спеціальна версія алгоритму для обробки коридорів [20].

Цей метод забезпечує гнучкість і контроль над структурою ігрових світів, що робить його потужним інструментом для розробників.

На рисунку 2.8 зображено генерацію рівня методом графів на основі кімнат.

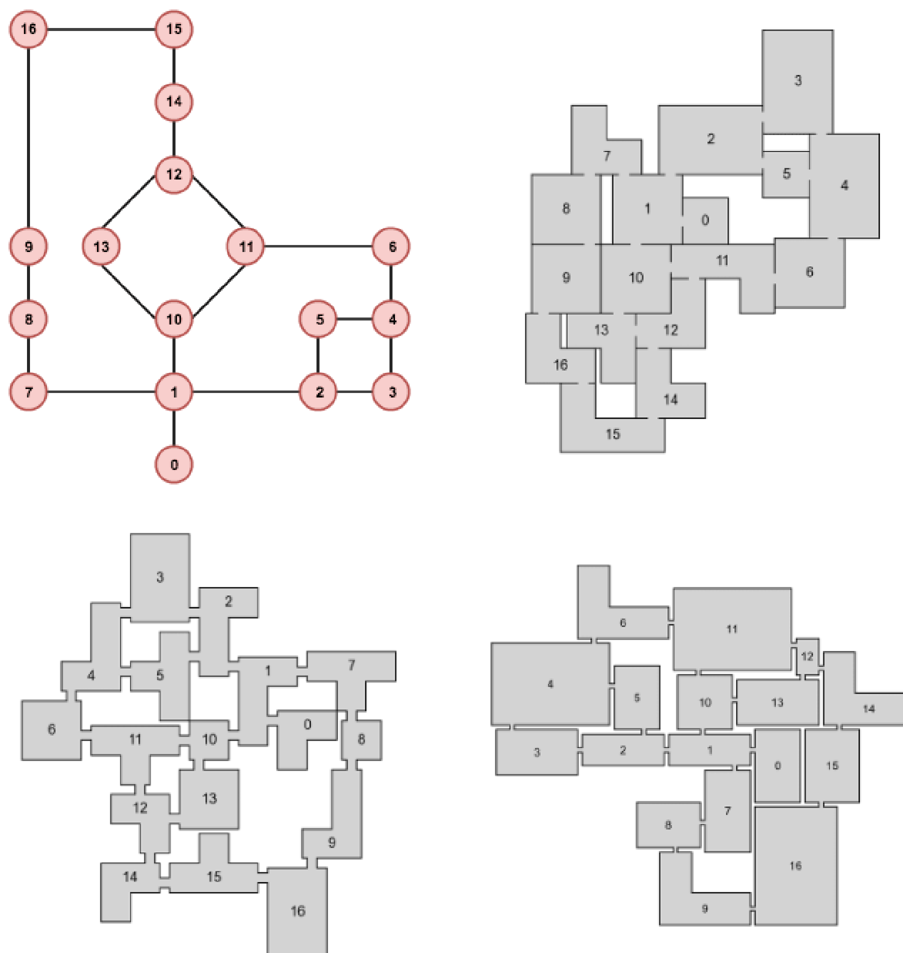

Рисунок 2.8 – Генерація рівня методом графів на основі кімнат [20]

Методи випадкової генерації рівнів в Unity відкривають широкі можливості для розробників 2D-ігор в жанрі roguelike, забезпечуючи різноманітність,

відтворюваність та непередбачуваність ігрового процесу. Кожен з цих методів має свої унікальні переваги і може бути використаний в залежності від специфіки проекту та бажаного результату. За допомогою Unity, розробники можуть ефективно імплементувати ці методи, створюючи захоплюючі та унікальні ігрові світи, які захоплюють гравців на довгі години.

Висновок до розділу 2

У розділі проаналізовано чотири основні ігрові рушії: Unity, Unreal Engine, Godot та GameMaker Studio. Кожен з них має свої переваги, такі як універсальність, потужні інструменти для анімації та фізики (Unity), передові графічні можливості (Unreal Engine), гнучкість і відкритість (Godot), та простота використання (GameMaker Studio).

Для розробки 2D-ігри в жанрі roguelike обрано Unity через його універсальність, потужні 2D-інструменти та підтримку процедурної генерації рівнів. Unity пропонує розробникам можливість створювати унікальні рівні за допомогою C# скриптів, що забезпечує новий досвід для гравців з кожним запуском гри. Велика спільнота та широкий спектр ресурсів Unity роблять його привабливим для інді-розробників.

Методи випадкової генерації рівнів, такі як розподіл кімнат, поділ простору, клітинні автомати та граф на основі кімнат, відкривають широкі можливості для створення різноманітних ігрових світів. Кожен з цих методів може бути ефективно використаний у Unity, забезпечуючи високу реіграбельність та непередбачуваність.

Вибір Unity як основного рушія для проекту виявився оптимальним завдяки його функціональності, гнучкості та підтримці спільноти, що сприятиме успішному розвитку та випуску гри.

3 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ГРИ

3.1 Визначення основних геймплейних механік для roguelike-гри

Цей розділ присвячений детальному опису ключових геймплейних механік, що складають основу розробленої гри в жанрі roguelike. Кожна механіка розглядається з точки зору її функціональності, взаємодії з іншими елементами гри та внеску у формування захопливого та реіграбельного ігрового досвіду, характерного для цього жанру.

Керування персонажем та рух.

Рух персонажа. Гравець має повний контроль над переміщенням свого персонажа у двовимірному ігровому світі. В основі механіки руху лежить використання вектора напрямку, який оновлюється в реальному часі залежно від команд, що вводить гравець. Для забезпечення плавності та природності руху використовується фізичний двигун Rigidbody2D. Це дозволяє персонажу реалістично взаємодіяти з оточенням, враховуючи зіткнення з перешкодами та інерцію.

Орієнтація в просторі. Орієнтація персонажа в просторі безпосередньо пов'язана з положенням курсора миші. Персонаж завжди повертається обличчям до курсора, що забезпечує інтуїтивне та зручне керування напрямком атаки та взаємодії з об'єктами навколишнього світу.

Бойова система.

Атака. Однією з ключових механік гри є можливість гравця атакувати ворогів. Атака здійснюється за допомогою холодної зброї (меча) та активується відповідною командою гравця. Кожна атака супроводжується унікальною анімацією, що підкреслює динаміку бою. Для запобігання спаму атак та створення більш тактичного геймплею впроваджено механізм затримки між атаками. Після кожного удару меч має "перезарядитися", перш ніж гравець зможе атакувати знову.

Зона ураження. Атака мечем має обмежену зону ураження, що визначається радіусом навколо персонажа. Тільки вороги, що знаходяться в межах цієї зони, отримують пошкодження. Це стимулює гравця до активного маневрування на полі

бою, ухилення від атак ворогів та пошуку оптимальних позицій для власних ударів.

Здоров'я та управління пошкодженнями.

Індикатор здоров'я. У відеоіграх здоров'я персонажа відображається за допомогою числових показників або так званих шкал здоров'я. Здоров'я "hit points" є кількісною оцінкою здатності персонажа витримувати ушкодження. При отриманні пошкоджень від ворогів здоров'я зменшується. Повна втрата здоров'я веде до втрати одного життя, після чого гравець продовжує гру з певного місця відродження. В деяких іграх можливе автоматичне відновлення здоров'я, якщо персонаж не отримує ушкоджень певний час. Гравці повинні уважно стежити за рівнем здоров'я та використовувати предмети для його відновлення [21].

На рисунку 3.1 зображено приклад індикатору здоров'я.

Рисунок 3.1 – Приклад індикатору здоров'я [21]

Смерть персонажа. Якщо здоров'я персонажа падає до нуля, він гине. Це призводить до завершення поточної ігрової сесії та повернення гравця на початок гри. Смерть є невід'ємною частиною жанру roguelike, що додає грі елемент ризику та напруги. Для покращення ігрового досвіду, можна створити захоплюючий екран "Game Over", який мотивуватиме гравця повернутися до гри. Наприклад, меню "Game Over" може зробити цей момент більш вражаючим, а також додавання опцій для перезапуску або виходу з гри [22].

Управління ігровими рівнями.

Меню. Гра має головне меню, яке слугує точкою входу для гравця. З меню

гравець може розпочати нову гру, продовжити попередню гру (якщо така є) або вийти з гри. Це забезпечує зручність та гнучкість у керуванні ігровим процесом.

Генерація рівнів. Однією з ключових особливостей жанру roguelike є процедурна генерація рівнів. У нашій грі кожен рівень складається з кількох кімнат, з'єднаних між собою переходами. Розташування, розміри та вміст кімнат визначаються випадковим чином на початку кожної нової гри. Це гарантує, що кожне проходження гри буде унікальним та непередбачуваним.

Ворожа поведінка.

Виявлення гравця. Виявлення гравця. Вороги в грі мають здатність виявляти гравця на певній відстані, що реалізується за допомогою рейкастинга, триггерів або інших методів виявлення. Ця відстань може змінюватися в залежності від типу ворога та його характеристик. Коли ворог виявляє гравця, він переходить у стан "тривоги" та починає переслідувати його [23].

Атака. Механізм атаки ворога активується, коли він наближається до гравця на достатню відстань. Кожен тип ворога має свій унікальний спосіб атаки, який може включати використання холодної зброї, стрільбу, магічні здібності тощо. Атака ворога супроводжується відповідною анімацією та звуковими ефектами, що допомагає гравцеві зрозуміти, що відбувається та вчасно зреагувати [23].

Управління ресурсами ігрової сцени.

Менеджер гри. Спеціальний компонент, який називається GameManager, відповідає за збереження та управління глобальними параметрами гри. Він відстежує поточний рівень, на якому знаходиться гравець, його здоров'я, кількість пройдених кімнат, кількість знищених ворогів та інші важливі дані. Цей компонент також відповідає за перехід між рівнями, збереження прогресу гравця та інші аспекти управління ігровою сесією.

3.2 Створення системи керування для гри

У даному розділі розглянуто процес створення та реалізації системи керування персонажем для гри в жанрі roguelike, використовуючи нову систему вводу Unity Input System. Нова система вводу є більш гнучкою та дозволяє

використовувати будь-який тип пристроїв для керування ігровим контентом, замінюючи класичний Input Manager [24].

На скріншоті показано налаштування, які були зроблені для керування персонажем. Введено "Action Maps" для гравця з діями, такими як рух (Move) та атака (Attack). Для руху використано клавіші W, A, S, D на клавіатурі, а для атаки ліву кнопку миші. Вікно "Player Input" містить параметри дій та вказує на використання нової системи вводу Unity.

На рисунку 3.2 зображено створення та налаштування Input System.

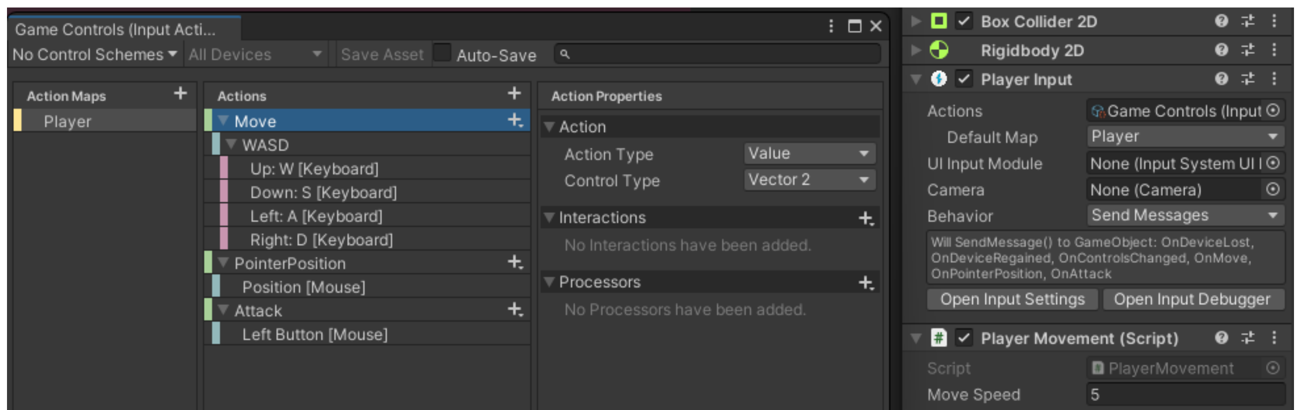

Рисунок 3.2 – Створення та налаштування Input System

Unity Input System дозволяє легко зчитувати поточний стан пристроїв вводу, таких як геймпади, клавіатури та миші, забезпечуючи гнучке налаштування керування. Наприклад, для зчитування даних з геймпада можна використовувати Gamepad.current.leftStick.ReadValue() для руху персонажа [25].

Створення Скрипта Руху Персонажа.

Скрипт PlayerMovement є ключовим компонентом для забезпечення керування рухами персонажа в ігровому світі. Використання компонента Rigidbody2D надає можливість реалізувати фізично коректний та плавний рух, враховуючи такі фактори, як інерція та зіткнення з об'єктами оточення.

На рисунку 3.3 зображено налаштування Rigidbody 2D та Box Collider 2D.

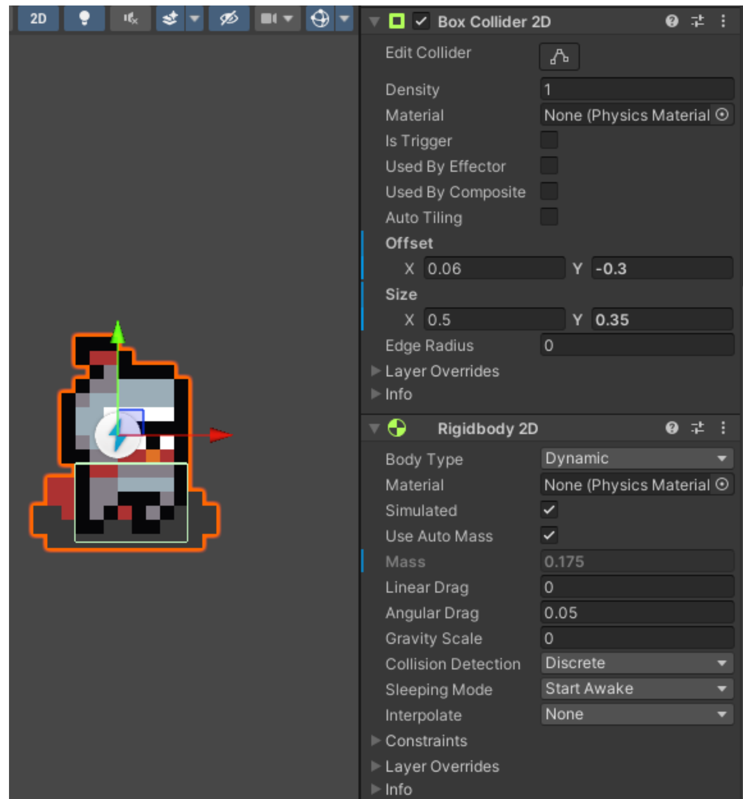

Рисунок 3.3 – Налаштування Rigidbody 2D та Box Collider 2D

Параметри скрипта, такі як moveSpeed, визначають швидкість переміщення персонажа, що дозволяє налаштувати ігровий процес відповідно до бажаної динаміки. Рух керується за допомогою методу OnMove, який отримує вектор напрямку з системи вводу. Цей вектор визначає, в якому напрямку та з якою інтенсивністю повинен рухатися персонаж.

На рисунку 3.4 зображено фрагмент коду PlayerMovement.

```
void OnMove(InputValue value)
{
    _moveDirection = value.Get<Vector2>();
}

private void FixedUpdate()
{
    Vector2 targetVelocity = _moveDirection * moveSpeed;
    _rigidbody2D.MovePosition(_rigidbody2D.position + targetVelocity * Time.fixedDeltaTime);
}
```

Рисунок 3.4 – Фрагмент коду PlayerMovement

Система вказівника та обертання зброї.

Для забезпечення інтуїтивного керування напрямком атаки персонажа

використовується скрипт WeaponRotation. Система відстежує позицію курсора миші на екрані та обчислює напрямок до цієї точки з позиції зброї. Це дозволяє гравцеві легко та точно цілитися у ворогів, просто переміщуючи курсор миші.

Скрипт Атаки.

Скрипт PlayerAttack реалізує можливість атаки ворогів, використовуючи анімацію та візуальні ефекти для створення враження удару. Використання аніматорів, таких як swordAnimator і effectAnimator, додає динаміки та реалізму процесу атаки. Контроль часу між атаками здійснюється через змінну attackCooldown, що запобігає безперервній атаці та створює необхідність тактичного підходу до бою.

На рисунку 3.5 зображено фрагмент коду PlayerAttack.

```
void OnAttack()
{
    if (Time.time - lastAttackTime > attackCooldown && !attackArea.isAttacked)
    {
        TriggerAttack();
    }
}
```

Рисунок 3.5 – Фрагмент коду PlayerAttack

Обробка зони атаки.

Компонент AttackArea використовується для визначення, чи потрапляють вороги у зону атаки персонажа. Цей скрипт взаємодіє з PlayerAttack для перевірки, чи була атака успішною і чи знаходиться ворог у межах досяжності атаки. При колізії з ворожим об'єктом, скрипт викликає метод TakeDamage на компоненті EnemyHealth ворога, завдаючи йому шкоди.

На рисунку 3.6 зображено фрагмент коду PlayerAttack.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    EnemyHealth enemyHealth = collision.GetComponent<EnemyHealth>();
    if (enemyHealth != null && playerAttack.IsAttacking && !isAttacked)
    {
        enemyHealth.TakeDamage(playerAttack.Damage);
        isAttacked = true;
    }
}
```

Рисунок 3.6 – Фрагмент коду PlayerAttack

Розробка системи керування персонажем в roguelike-іграх є важливою складовою геймплею, що вимагає уважного підходу до деталей інтерактивності. Використання Unity Input System дозволило створити інтуїтивно зрозумілу та відгукову систему керування, яка підвищує загальну залученість гравців та задоволення від гри.

3.3 Додавання ворогів та налаштування їх характеристик

У цьому розділі буде описано процес створення різноманітних ворогів для roguelike-гри: гобліна, слимака та літаючого ворога. Кожен ворог матиме унікальні характеристики та поведінку, реалізовані за допомогою скриптів EnemyMovement, EnemyHealth та EnemyAttack.

Скрипт руху ворога.

Скрипт EnemyMovement відповідає за переміщення ворога в напрямку гравця, а також за анімацію та зміну орієнтації спрайту ворога залежно від його руху.

На рисунку 3.7 зображено фрагмент коду EnemyMovement.

```
void Update()
{
    if (player.position.y > enemyTransform.position.y)
    {
        enemySpriteRenderer.sortingOrder = 9;
    }
    else
    {
        enemySpriteRenderer.sortingOrder = 6;
    }

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

    if (distanceToPlayer <= detectionRange)
    {
        if (distanceToPlayer > attackRange)
        {
            enemyAnimator.SetBool("IsRunning", true);
            MoveTowardsPlayer();
        }
        else
        {
            enemyAnimator.SetBool("IsRunning", false);
            enemyAttack.AttackPlayer();
        }
    }
    else
    {
        enemyAnimator.SetBool("IsRunning", false);
    }
```

Рисунок 3.7 – Фрагмент коду EnemyMovement

Поведінка Руху.

Виявлення гравця. Ворог починає рухатися у напрямку гравця, коли відстань між ними менша за detectionRange.

Зміна напрямку. Спрайт ворога відображається дзеркально залежно від напрямку руху, щоб він завжди був повернутий обличчям до гравця.

Переслідування. Якщо ворог знаходиться на відстані між detectionRange та attackRange, він рухається до гравця зі швидкістю moveSpeed.

Атака. Якщо відстань до гравця менша за attackRange, ворог припиняє рух та атакує, викликаючи метод AttackPlayer скрипта EnemyAttack.

Скрипт Здоров'я Ворога

Скрипт EnemyHealth відповідає за керування здоров'ям ворога, візуалізацію

стану здоров'я за допомогою слайдера healthSlider, а також за знищення ворога при досягненні нульового рівня здоров'я.

На рисунку 3.8 зображено налаштування healthSlider.

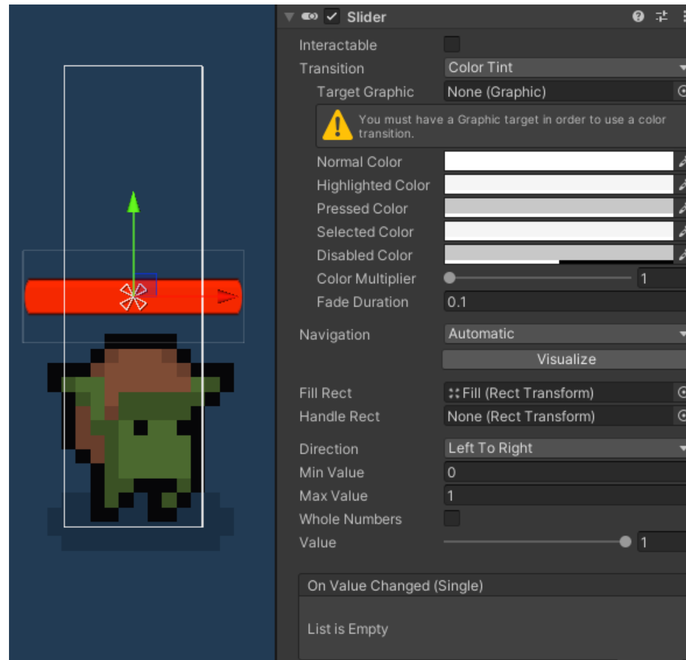

Рисунок 3.8 – Налаштування healthSlider

Обробка Здоров'я.

Отримання шкоди: Метод TakeDamage зменшує поточне здоров'я ворога на величину отриманої шкоди. Оновлення слайдера здоров'я: Після кожної зміни здоров'я викликається метод UpdateHealthSlider для візуалізації поточного стану здоров'я на слайдері. Смерть: Якщо здоров'я ворога досягає нуля, викликається метод Die, який знищує об'єкт ворога.

На рисунку 3.9 зображено фрагмент коду EnemyHealth.

```
public void TakeDamage(int damage)
{
    currentHealth -= damage;
    UpdateHealthSlider();

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

Рисунок 3.9 – Фрагмент коду EnemyHealth

Скрипт атаки ворога.

Метод AttackPlayer викликається зі скрипта EnemyMovement, коли гравець знаходиться в межах дистанції атаки. Якщо пройшов достатній час з моменту останньої атаки (attackCooldown), вороги завдають шкоди гравцеві.

На рисунку 3.10 зображено фрагмент коду EnemyAttack.

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

Рисунок 3.10 – Фрагмент коду EnemyAttack

Унікальні характеристики ворогів.

Гоблін: швидкий та дуже міцний ворог.

Слимак: повільний та слабкий ворог.

Літаючий ворог: найшвидший та маневрений ворог, але дуже слабкий.

Налаштування унікальних характеристик здійснюється через зміну параметрів скриптів (moveSpeed, damage, attackCooldown, тощо).

На рисунку 3.11 зображено приклад налаштування характеристик ворога.

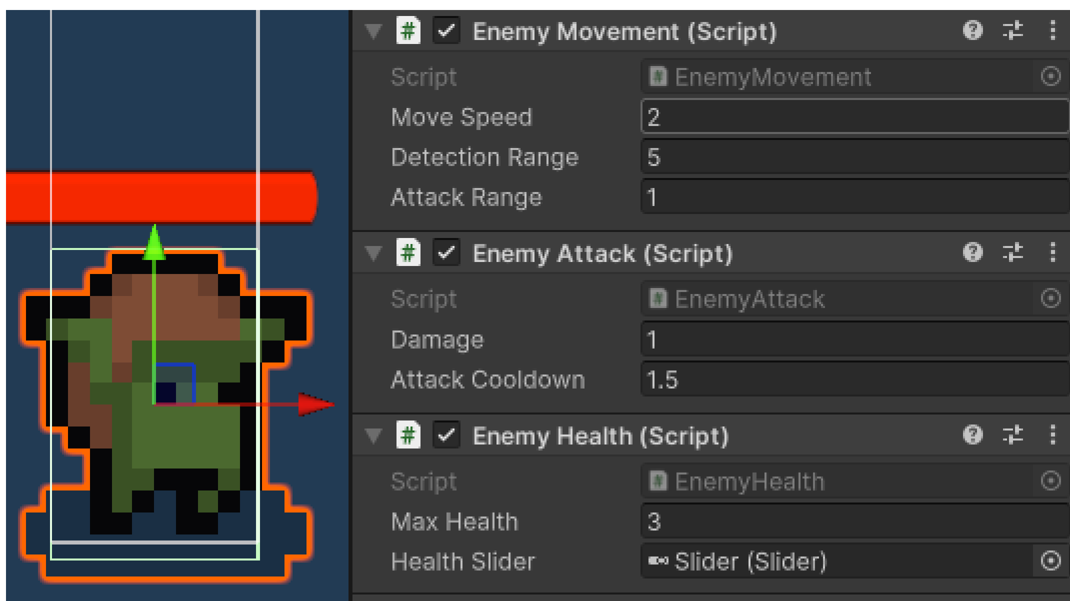

Рисунок 3.11 – Приклад налаштування характеристик ворога

Додавання різноманітних ворогів з унікальними характеристиками робить roguelike-гру більш цікавою та захоплюючою. Скрипти EnemyMovement, EnemyHealth та EnemyAttack забезпечують базову поведінку ворогів, а налаштування їх параметрів дозволяє створити унікальні типи ворогів з різними стилями бою та рівнями складності.

На рисунку 3.12 зображено створенні вороги.

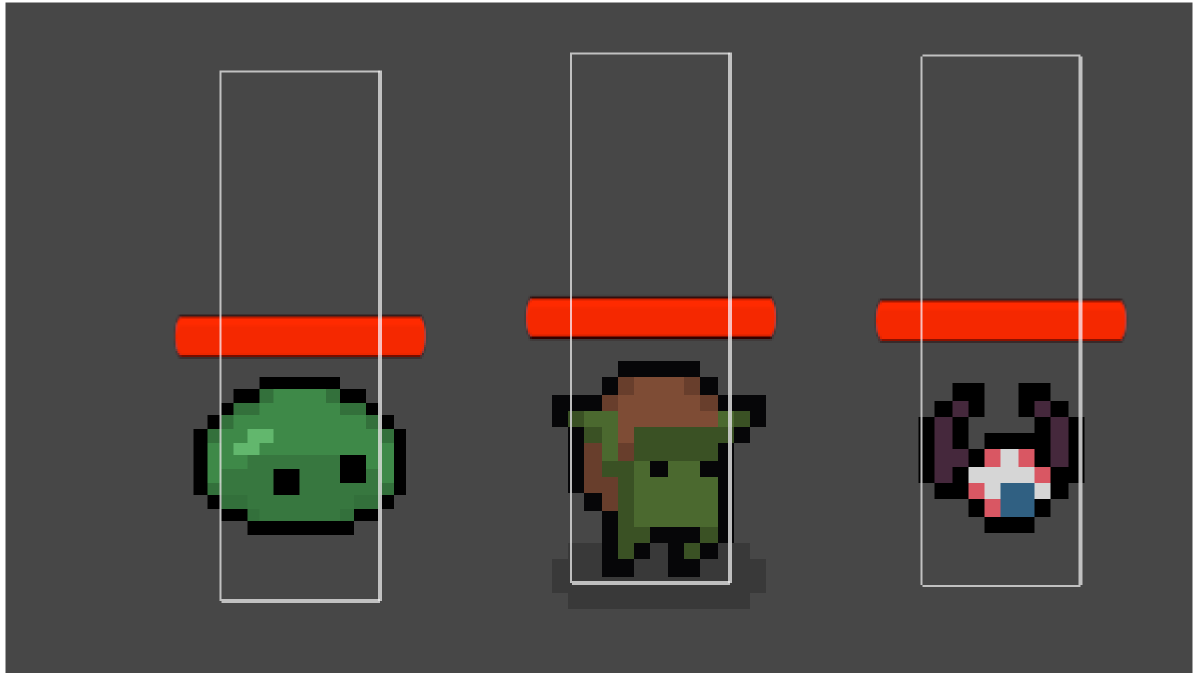

Рисунок 3.12 – Створенні вороги

3.4 Підготовка та реалізація алгоритму випадкового генерування рівня

Процедурна генерація є важливою для створення захоплюючого і непередбачуваного ігрового досвіду в рольових іграх. Вона дозволяє динамічно створювати унікальні рівні, кімнати та коридори, які логічно пов'язані між собою, забезпечуючи різноманітність геймплейних сценаріїв завдяки різним конфігураціям і розміщенню кімнат. Це дозволяє кожного разу створювати новий рівень, підтримуючи інтерес гравця до гри і збільшуючи її реіграбельність. Процедурна генерація є ключовим компонентом в іграх жанру roguelike, оскільки вона гарантує, що кожен прохід буде унікальним і вимагає адаптації стратегії, що заохочує дослідження і відкриття нових можливостей та сценаріїв гри [26].

Підготовка до генерації рівня.

Перед початком були створені префаби кімнат, які будуть використовуватися в подальшому для побудови рівня. Префаби включають в себе різні типи кімнат з різними конфігураціями виходів (верх, низ, ліво, право), а також спеціальні кімнати, такі як стартова кімната і вихідна кімната. Це дозволяє створювати різноманітні комбінації кімнат і забезпечувати неповторність кожного нового рівня.

На рисунку 3.13 зображено результат створення префабів кімнат.

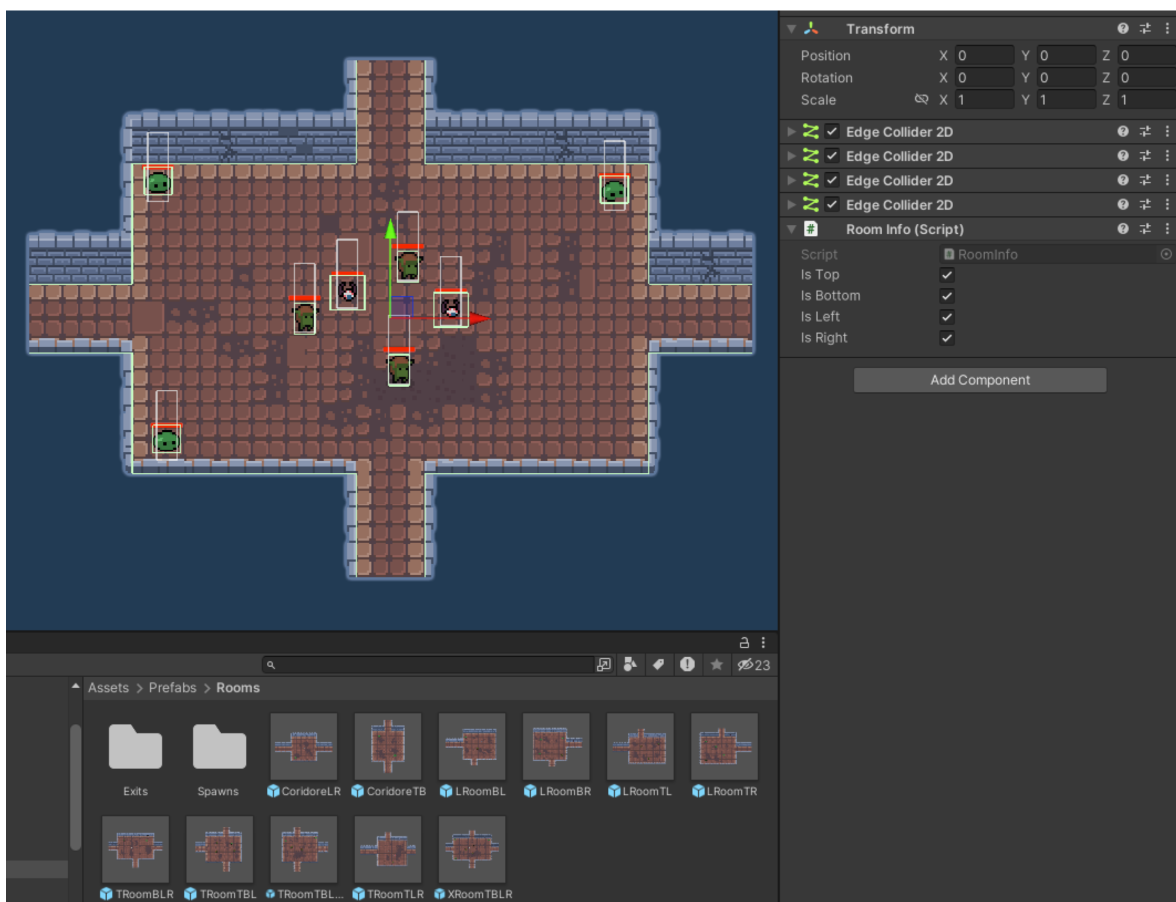

Рисунок 3.13 – Результат створення префабів кімнат

Реалізація алгоритму.

Створення мапи рівня метод CreateMap.

Для початку створення рівня потрібно ініціювати карту, яка буде представляти собою набір координат для кімнат. Цей процес починається з визначення початкової точки та поступового розширення карти шляхом додавання

нових кімнат.

Метод CreateMap є ключовим етапом в процесі генерації рівня. Він починається з додавання початкової кімнати у центр карти. Алгоритм використовує об'єкт pointObject як місце для кімнат, яке додається в список placeForRooms. Початкова позиція визначається, як правило, в центрі координатної сітки, що дозволяє рівномірно розподіляти кімнати в усіх напрямках.

На рисунку 3.14 зображено фрагмент коду додавання початкової кімнати.

```
private void CreateMap()
{
    AddRoom(pointObject.transform.position);

    GameObject currentPosition = placeForRooms[0];
    int errorCounter = 1;
    int attempts = 0;
    int maxAttempts = 20;

    Vector3[] directions = {
        new Vector3(0, roomSize.y, 0),  // Вверх
        new Vector3(0, -roomSize.y, 0), // Вниз
        new Vector3(-roomSize.x, 0, 0), // Ліворуч
        new Vector3(roomSize.x, 0, 0)   // Праворуч
    };
```

Рисунок 3.14 – Фрагмент коду додавання початкової кімнати

Далі алгоритм використовує набір напрямків (directions), який включає в себе чотири можливих варіанти: верх, низ, ліво і право. На кожній ітерації випадково вибирається один з цих напрямків, щоб додати нову кімнату. Нова позиція обчислюється шляхом додавання вектора напрямку до поточної позиції.

Якщо нова позиція не зайнята іншою кімнатою (тобто не міститься в occupiedPositions), то кімната додається до списку placeForRooms, а її координати додаються до occupiedPositions. В іншому випадку, якщо позиція вже зайнята, алгоритм намагається вибрати інший напрямок або ж переміщується на попередню позицію і пробує знову. Це забезпечує уникнення дублювання кімнат та дозволяє запобігти нескінченному циклу генерації.

Управління спробами і помилками.

Для забезпечення стабільності алгоритму використовується лічильник помилок (errorCounter) та лічильник спроб (attempts). Якщо алгоритм не може знайти потрібну позицію після певної кількості спроб (maxAttempts), він видаляє частину вже розміщених кімнат (RemoveQuarterRooms) та звільняє місце для нових спроб. Це дає можливість алгоритму адаптуватися та знаходити нові рішення навіть у складних конфігураціях карти. Метод RemoveQuarterRooms видаляє чверть кімнат, що дозволяє уникнути заповнення карти невдалими спробами та забезпечує гнучкість алгоритму в пошуку нових можливостей для розміщення кімнат.

На рисунку 3.15 зображено фрагмент коду що відповідає за перевірку на помилки.

```
if (!occupiedPositions.Contains(newPosition))
{
    AddRoom(newPosition);
    currentPosition = placeForRooms[placeForRooms.Count - 1];
    attempts = 0; // Скидання кількості помилок, оскільки успішно додано кімнату
}
else
{
    attempts++;
    if (attempts > maxAttempts)
    {
        RemoveQuarterRooms();
        if (placeForRooms.Count > 0)
        {
            currentPosition = placeForRooms[placeForRooms.Count - 1];
        }
        errorCounter++;
        attempts = 0;
    }
    if (errorCounter > 20)
    {
        Debug.LogError("Помилка генерації карти: не вдалося створити необхідну кіль
        break;
    }
}
```

Рисунок 3.15 – Фрагмент коду що відповідає за перевірку на помилки

Використання GameManager.

Алгоритм автоматично адаптується до параметрів гри, таких як кількість кімнат, задана менеджером гри (GameManager.Instance.numbersOfRooms). Це забезпечує, що генерація рівня залишається актуальною для поточного сценарію гри і може бути масштабована в залежності від рівня підземелля. Мінімальна кількість кімнат встановлюється на рівні чотирьох.

На рисунку 3.16 зображено фрагмент коду, що відповідає за встановлення значення кількості кімнат.

```
numberOfRooms = GameManager.Instance.numbersOfRooms;
```

Рисунок 3.16 – Фрагмент коду, що відповідає за встановлення значення кількості кімнат

Після створення карти, що складається з точок, настає етап розміщення фактичних кімнат. Цей етап включає в себе аналіз кожної позиції та вибір відповідних кімнатних об'єктів на основі їх взаємодії з іншими кімнатами.

Метод PlaceRooms.

Метод PlaceRooms у класі LevelGenerator відповідає за розташування кімнат на мапі після її створення. Він враховує позиції кімнат і відповідні об'єкти для розміщення таких кімнат як стартова кімната, проміжні кімнати та вихідна кімната. Метод працює з позиціями, які були визначені раніше під час генерації мапи, і на основі цих позицій розставляє відповідні об'єкти кімнат.

На рисунку 3.17 зображено фрагмент коду PlaceRooms.

```
void PlaceRooms()
{
    Vector2 firstRoomPosition = placeForRooms[0].transform.position;
    Vector2 secondPosition = placeForRooms[1].transform.position;
    Vector2 preLastPosition = placeForRooms[placeForRooms.Count - 2].transform.position;
    Vector2 lastPosition = placeForRooms[placeForRooms.Count - 1].transform.position;

    PlaceSpawnRoom(secondPosition, firstRoomPosition);
    PlaceSecondRoom(secondPosition, lastPosition);

    for (int counter = 2; counter < placeForRooms.Count - 2; counter++)
    {
        Vector2 currentPosition = placeForRooms[counter].transform.position;
        PlaceIntermediateRoom(currentPosition, firstRoomPosition, lastPosition);
    }

    PlacePreLastRoom(preLastPosition, firstRoomPosition);
    PlaceExitRoom(preLastPosition, lastPosition);
}
```

Рисунок 3.17 – Фрагмент коду PlaceRooms

Розміщення кімнат.

Стартова кімната.

Спочатку метод розміщує стартову кімнату, використовуючи метод PlaceSpawnRoom. Цей метод визначає, яка зі сторін кімнати (верх , низ, ліво або

право) буде містити стартову кімнату, залежно від позиції другої кімнати. Наприклад, якщо друга кімната знаходиться верху над стартовою кімнатою, то обирається кімната з виходом вверх. Після цього, стартова кімната створюється на відповідній позиції.

Друга кімната.

Далі розміщується друга кімната після стартової, використовуючи метод PlaceSecondRoom. Цей метод перевіряє, які сусідні позиції вже зайняті, щоб правильно визначити тип кімнати, яка буде розміщена на цій позиції. Метод використовує логіку, схожу на розміщення стартової кімнати, але з урахуванням більшого числа можливих напрямків для виходів. Після визначення відповідної кімнати, вона створюється і додається до списку кімнат.

Проміжні кімнати.

Для всіх проміжних кімнат між другою і передостанньою використовується метод PlaceIntermediateRoom. Цей метод також перевіряє сусідні позиції, щоб визначити, які стіни повинні бути присутні в кімнаті. Він забезпечує, щоб кожна проміжна кімната мала правильні виходи, які з’єднують її з сусідніми кімнатами, створюючи логічний шлях від стартової кімнати до виходу.

Передостання кімната.

Передостання кімната розміщується за допомогою методу PlacePreLastRoom. Цей метод працює аналогічно до розміщення другої кімнати, перевіряючи сусідні позиції і визначаючи правильний тип кімнати. Важливо, щоб передостання кімната мала правильні виходи для з'єднання з попередніми і наступними кімнатами, включаючи вихідну кімнату.

Вихідна кімната.

Нарешті, вихідна кімната розміщується за допомогою методу PlaceExitRoom. Цей метод визначає, де повинна бути розміщена вихідна кімната, залежно від позиції передостанньої кімнати. Як і у випадку зі стартовою кімнатою, вихідна кімната розміщується з урахуванням напрямку виходів, щоб забезпечити правильний шлях до завершення рівня.

На рисунку 3.18 зображено результат роботи алгоритму випадкової генерації рівня.

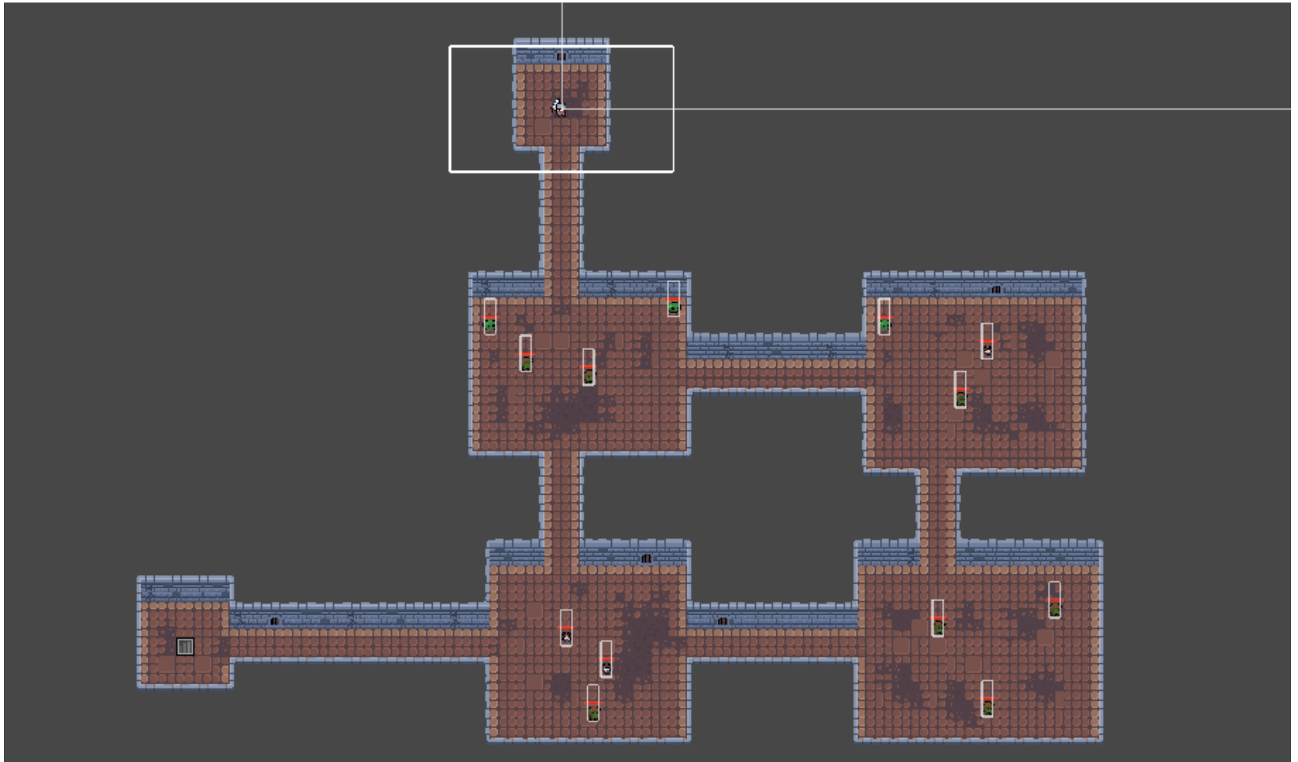

Рисунок 3.18 – Результат роботи алгоритму випадкової генерації рівня

Генерація випадкових рівнів є важливим аспектом розробки рольових ігор, який дозволяє підтримувати інтерес гравців і забезпечувати високу реіграбельність. Метод PlaceRooms у поєднанні з іншими допоміжними методами забезпечує створення унікальних ігрових карт з логічно пов'язаними кімнатами, що дозволяє створювати захоплюючі і непередбачувані ігрові досвіди. Завдяки цьому підходу, кожен новий рівень буде відчуватися свіжим і цікавим, підтримуючи інтерес гравців до гри протягом тривалого часу.

Висновок

В результаті розробки та реалізації алгоритму випадкової генерації рівнів, було створено префаби, які служать основою для побудови унікальних ігрових карт. Ці префаби містять різні типи кімнат з різними конфігураціями виходів, що забезпечує гнучкість і різноманітність при створенні нових рівнів. Метод CreateMap дозволяє ініціювати мапу рівня та розміщувати кімнати на базі

випадково вибраних напрямків, уникаючи дублювання і запобігаючи нескінченним циклам генерації.

Завершення генерації рівня методами PlaceRooms, PlaceSpawnRoom, PlaceSecondRoom, PlaceIntermediateRoom, PlacePreLastRoom та PlaceExitRoom гарантує логічне розміщення кімнат, створюючи послідовний маршрут для гравця від стартової кімнати до виходу. Цей підхід дозволяє створювати захоплюючі ігрові рівні з високим рівнем реіграбельності та унікальності.

Таким чином, реалізація алгоритму випадкової генерації рівня успішно забезпечила створення динамічних і унікальних ігрових карт, які кожного разу пропонують новий досвід для гравців, підтримуючи їхній інтерес до гри протягом тривалого часу.

3.5 Створення головного меню гри

Головне меню є важливою частиною будь-якої гри, оскільки воно надає гравцям можливість почати нову гру, налаштувати параметри, такі як звук, або вийти з гри. Добре структуроване меню покращує загальний досвід користувача, дозволяючи легко орієнтуватися та керувати різними аспектами гри.

Створення головного меню.

Головне меню є важливою складовою для створення позитивного першого враження та забезпечення зручної навігації в грі. Було створено головне меню, яке містить основні кнопки для взаємодії гравця з грою. Головне меню є першим враженням гравця про гру і відіграє важливу роль у навігації. У меню були додані кнопки "Play", "Options" та "Exit" для забезпечення зручності використання та професійного вигляду гри. Наприклад, кнопка "Play" запускає гру, а кнопка "Exit" завершує її роботу, що забезпечує легкість у використанні та покращує загальний ігровий досвід [27].

На рисунку 3.19 зображено результат побудованого меню.

Рисунок 3.19 – Результат побудованого меню

Це меню дозволяє гравцеві почати нову гру, зайти в налаштування, або вийти з гри. Ось відповідний код, який реалізує ці функції.

На рисунку 3.20 зображено фрагмент коду який відповідає за логіку головного меню.

```
public class Menu : MonoBehaviour
{
    0 references
    public void PlayNewGame()
    {
        GameManager.Instance.playerCurrentHp = GameManager.Instance.PLAYERCURRENTHP;
        GameManager.Instance.currentLevel = GameManager.Instance.CURRENTLEVEL;
        GameManager.Instance.numbersOfRooms = GameManager.Instance.NUMBERSOFROOMS;
        Time.timeScale = 1;

        GameManager.Instance.StartGameData();
        SceneManager.LoadScene("Game");
    }

    0 references
    public void ExitGame()
    {
        Debug.Log("Exit game requested");
#if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
#endif
        Application.Quit();
    }
}
```

Рисунок 3.20 – Фрагмент коду який відповідає за логіку головного меню

Додавання меню налаштування звуку.

Додавання вікна налаштування звуку в головне меню є важливим кроком для поліпшення користувацького досвіду. Це дозволяє гравцям регулювати різні параметри звуку гри, такі як загальний рівень звуку, гучність музики та звукових ефектів, що забезпечує більш індивідуалізоване та комфортне ігрове середовище. Наприклад, за допомогою спеціальних повзунків користувач може налаштувати рівень звуку відповідно до своїх вподобань, що значно покращує загальний враження від гри та її зручність [28].

На рисунку 3.21 зображено результат розробленого меню налаштування звуку.

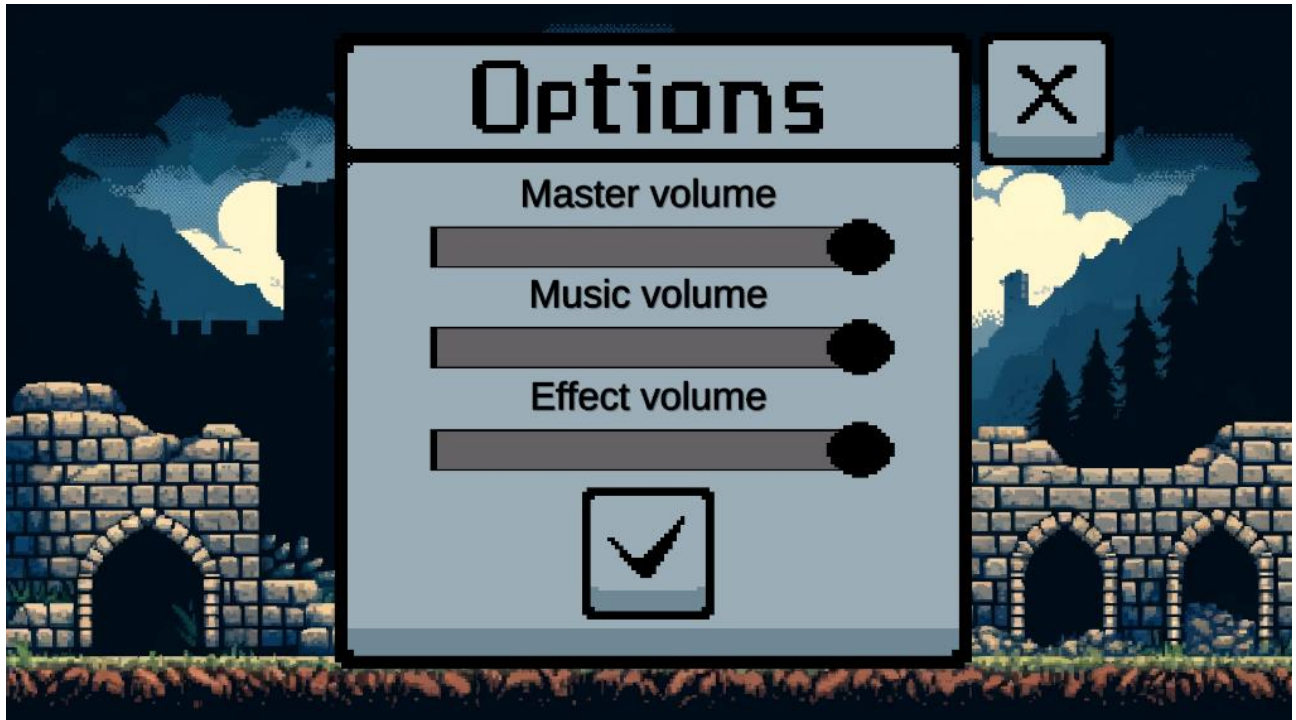

Рисунок 3.21 – Результат розробленого меню налаштування звуку

Створення екрану смерті персонажа.

На завершення було створено екран смерті персонажа, який відображається після поразки гравця. Цей екран дозволяє гравцеві вибрати між повторним запуском гри або виходом з неї. Нижче наведено малюнок екрану смерті.

На рисунку 3.22 зображено меню смерті персонажа.

Рисунок 3.22 – Меню смерті персонажа

Ці елементи разом забезпечують повноцінний ігровий інтерфейс, який робить гру більш зручною та привабливою для гравців.

3.6 Збір інформації з гри

Збір інформації з гри є важливою складовою для аналізу ігрового процесу, розуміння поведінки гравців, а також для подальшого вдосконалення ігрового контенту. Даний підрозділ детально описує механізми збору даних у грі, їх обробку та збереження, базуючись на наданому коді гри, написаному на мові програмування C# з використанням Unity.

Структура та основні функції.

GameManager є центральним компонентом для збору та обробки ігрових даних. Його основні функції включають відстеження ігрового прогресу, збирання статистики про гравця та рівні, а також збереження зібраних даних у файл для подальшого аналізу.

Ключовими елементами GameManager є:

– відстеження часу початку та завершення гри;

– записування інформації про кожен рівень. Початковий та кінцевий час, кількість завданих і отриманих пошкоджень, статус проходження;

– збереження зібраних даних у файл для подальшого аналізу.

Ініціалізація та зберігання даних про рівень.

Коли гра починається, викликається метод StartGameData, який фіксує час початку гри та ініціалізує об'єкт для зберігання даних про поточний рівень.

На рисунку 3.23 зображено фрагмент коду StartGameData.

```
public void StartGameData()
{
    levelData = new LevelData();
    gameStartTime = DateTime.Now;
    levelData.startTime = DateTime.Now;
}
```

Рисунок 3.23 – Фрагмент коду StartGameData

Метод nextLevelData використовується для переходу між рівнями і збирає інформацію про завершений рівень.

На рисунку 3.24 зображено фрагмент коду NextLevelData.

```
public void NextLevelData(bool isPassed)
{
    if (!isPassed)
    {
        currentLevel++;
    }
    levelData.currentLevel = Instance.currentLevel;
    levelData.endTime = DateTime.Now;
    levelData.isPassed = isPassed;
    levelDatas.Add(levelData);
    levelData = new LevelData();
    levelData.startTime = DateTime.Now;
}
```

Рисунок 3.24 – Фрагмент коду NextLevelData

Цей метод оновлює поточний рівень, фіксує час його завершення, статус проходження, а потім зберігає ці дані до списку levelDatas.

Збір даних про події.

GameManager також збирає інформацію про події, що відбуваються протягом гри. Це включає отримання і завдання пошкоджень.

На рисунку 3.25 зображено фрагмент коду методів PlayerTookDamage та EnemyTookDamage.

```
1 reference
public void PlayerTookDamage(int damage)
{
    if (levelData != null)
    {
        levelData.damageTaken += damage;
    }
}

1 reference
public void EnemyTookDamage(int damage)
{
    if (levelData != null)
    {
        levelData.damageDealt += damage;
    }
}
```

Рисунок 3.25 – Фрагмент коду методів PlayerTookDamage та EnemyTookDamage

Ці методи оновлюють відповідні поля в об'єкті levelData, коли гравець або ворог отримують пошкодження.

Завершення гри та збереження даних Метод PlayerDiedData викликається у випадку смерті гравця, завершуючи гру та зберігаючи всі зібрані дані.

На рисунку 3.26 зображено фрагмент коду PlayerDiedData і EndGameData.

```
1 reference
public void StartGameData()
{
    levelData = new LevelData();
    gameStartTime = DateTime.Now;
    levelData.startTime = DateTime.Now;
}

1 reference
private void EndGameData()
{
    gameEndTime = DateTime.Now;
}
```

Рисунок 3.26 – Фрагмент коду PlayerDiedData і EndGameData

Метод SaveGameData зберігає всі зібрані дані у файл у форматі тексту.

На рисунку 3.27 зображено фрагмент коду SaveGameData.

```
1 reference
private void SaveGameData()
{
    string gameData = $"Game Start Time: {gameStartTime}\nGame End Time: {gameEndTime}\n\n";

    foreach (LevelData levelData in levelDatas)
    {
        gameData += $"Level: {levelData.currentLevel}\nStart Time: {levelData.startTime}" +
            $"\nEnd Time: {levelData.endTime}\nDamage Taken: {levelData.damageTaken}" +
            $"\nDamage Dealt: {levelData.damageDealt}\nPassed: {levelData.isPassed}\n\n";
    }

    string filePath = Application.persistentDataPath + $"/GameData_{DateTime.Now:yyyy-MM-dd-HH-mm-ss}.txt";
    System.IO.File.WriteAllText(filePath, gameData);
    Debug.Log("Game data saved to " + filePath);
}
```

Рисунок 3.27 – Фрагмент коду SaveGameData

Цей метод створює детальний звіт про кожен рівень та всю ігрову сесію, який зберігається у файл в директорії Application.persistentDataPath.

Збір інформації з гри є критичним компонентом для аналізу та вдосконалення ігрового процесу. GameManager забезпечує ефективний збір, обробку та збереження даних про ігрові сесії, що дозволяє розробникам детально аналізувати поведінку гравців та вносити покращення до гри. Впровадження цього механізму дозволяє збільшити залученість гравців, підвищити якість гри та забезпечити довготривале задоволення від ігрового процесу.

3.7 Завершення створення гри

Додавання анімацій.

Анімації є важливою складовою візуального оформлення гри, оскільки вони додають динаміки та реалістичності. У грі було додано анімації для руху персонажа, атак, реакцій на пошкодження та інших дій.

Скрипт PlayerMovement відповідає за анімацію руху гравця.

На рисунку 3.28 зображено фрагмент коду який відповідає за анімацію гравця.

```
Unity Message | 0 references
private void Update()
{
    _animator.SetFloat("Speed", _moveDirection.magnitude);

    Vector3 worldPointerPosition = mainCamera.ScreenToWorldPoint(new Vector3(pointerPosition.x, pointerPosition.y, mainCamera.nearClipPlane));
    worldPointerPosition.z = 0;

    Vector2 direction = ((Vector2)worldPointerPosition - (Vector2)_transform.position).normalized;

    if (direction.x < 0)
        _transform.rotation = Quaternion.Euler(0, 180, 0);
    else if (direction.x > 0)
        _transform.rotation = Quaternion.Euler(0, 0, 0);
}
```

Рисунок 3.28 – Фрагмент коду який відповідає за анімацію гравця

Анімація ворогів реалізована у скрипті EnemyMovement.

На рисунку 3.29 зображено фрагмент коду що відповідає за анімацію ворогів.

```
if (distanceToPlayer <= detectionRange)
{
    if (distanceToPlayer > attackRange)
    {
        enemyAnimator.SetBool("IsRunning", true);
        MoveTowardsPlayer();
    }
    else
    {
        enemyAnimator.SetBool("IsRunning", false);
        enemyAttack.AttackPlayer();
    }
}
else
{
    enemyAnimator.SetBool("IsRunning", false);
}
```

Рисунок 3.29 – Фрагмент коду що відповідає за анімацію ворогів

Щоб анімації вірно працювали потрібно налаштувати аніматор.

На рисунку 3.30 зображено налаштування аніматора для гравця.

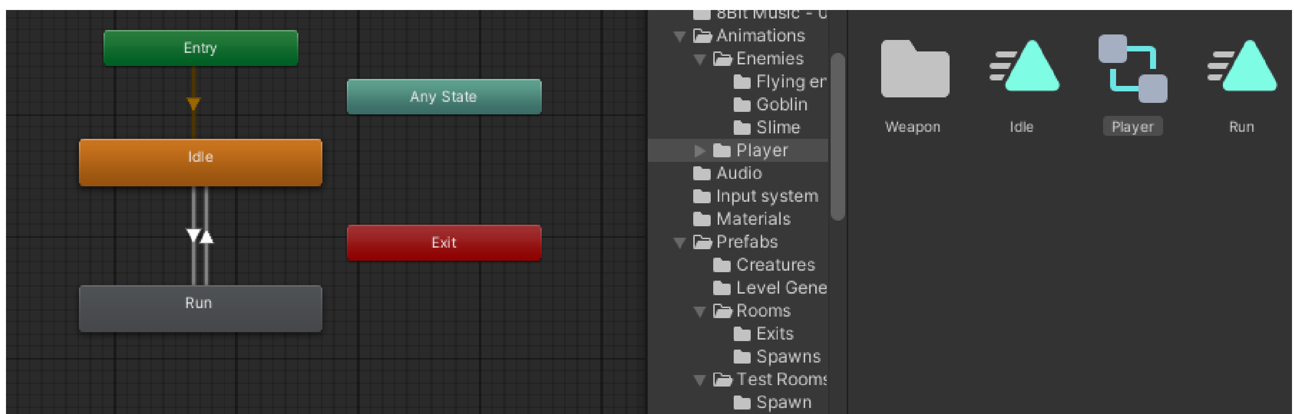

Рисунок 3.30 – Налаштування аніматора для гравця

Також було налаштовано аніматори для всіх видів ворогів, що дозволило реалізувати різні типи анімацій для кожного ворога. Це включає анімації руху, анімація простою, що робить кожного ворога унікальним і додає різноманітності до ігрового процесу. Завдяки цьому, гравець отримує більш багатий та динамічний досвід під час битв з ворогами, що сприяє більш захоплюючому ігровому процесу. Ці анімації створюють плавний та природний рух персонажів, що підвищує загальну якість візуального сприйняття гри.

Додавання звуку та музики.

Звукове оформлення значно впливає на сприйняття гри, створюючи атмосферу та допомагаючи гравцям орієнтуватися у подіях гри. У грі було додано різні звукові ефекти для атак, отримання пошкоджень та інших важливих подій, а також фонову музику, що створює відповідний настрій.

Скрипт **AudioController** відповідає за відтворення звукових ефектів.

На рисунку 3.31 зображено фрагмент коду що відповідає за звук.

```
using UnityEngine;
public class AudioController : MonoBehaviour
{
    [SerializeField] private AudioSource playerAttackAudio;
    [SerializeField] private AudioSource PlayerTakeDamage;
    [SerializeField] private AudioSource enemyTakeDamage;

    public void PlayAudioPlayerAttack()
    {
        playerAttackAudio.pitch = Random.Range(0.7f, 1f);
        playerAttackAudio.Play();
    }
    public void PlayAudioPlayerTakeDamage()
    {
        playerAttackAudio.pitch = Random.Range(0.9f, 1.1f);
        PlayerTakeDamage.Play();
    }
    public void PlayAudioEnemyTakeDamage()
    {
        playerAttackAudio.pitch = Random.Range(0.9f, 1.1f);
        enemyTakeDamage.Play();
    }
}
```

Рисунок 3.31 – Фрагмент коду що відповідає за звук

Цей скрипт забезпечує випадкове змінення тону звуку для кожного ефекту, що додає різноманітності звуковому супроводу гри.

Скрипт **SoundManager** керує загальним рівнем гучності гри.

На рисунку 3.32 зображено фрагмент коду який керує гучністю.

```
public void ChangeVolume()
{
    audioMixer.SetFloat("Master", Mathf.Lerp(-40, 0, masterVolumeSlider.value));
    audioMixer.SetFloat("Music", Mathf.Lerp(-40, 0, musicVolumeSlider.value));
    audioMixer.SetFloat("Sound", Mathf.Lerp(-40, 0, soundVolumeSlider.value));
}
```

Рисунок 3.32 – Фрагмент коду який керує гучністю

Використання аудіо мікшера дозволяє гравцям налаштовувати гучність різних аспектів звуку відповідно до їхніх уподобань.

На рисунку 3.33 зображено результат побудованого меню налаштування звуку.

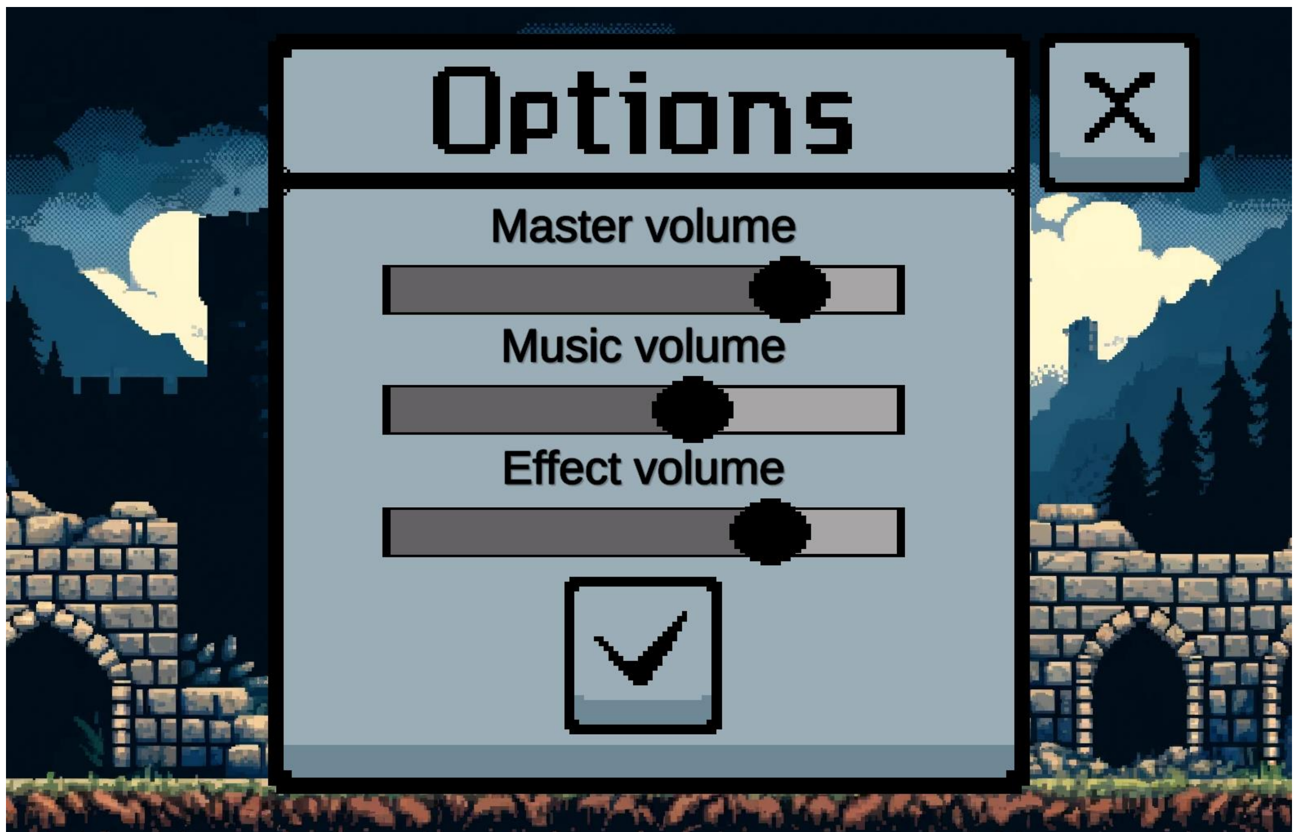

Рисунок 3.33 – Результат побудованого меню налаштування звуку

На завершення створення гри було додано анімації та звукові ефекти, що

значно покращили ігровий досвід. Анімації забезпечують більш реалістичний та захоплюючий візуальний стиль, а звуки та музика створюють атмосферу, яка занурює гравців у світ гри. Ці вдосконалення сприяють тому, що гра стає не тільки більш привабливою, але й більш інтерактивною та емоційно насиченою.

Висновок до розділу 3

У розділі 3 детально розглянуто процес проектування та реалізації гри в жанрі roguelike. Описані ключові аспекти створення ігрових механік, таких як керування персонажем, рух, бойова система, індикатор здоров'я та система управління пошкодженнями. Було впроваджено механізми, що забезпечують плавність та природність руху персонажа, зручність атаки та взаємодії з оточенням, а також додано затримку між атаками для більш тактичного геймплею.

Особлива увага приділена створенню системи керування персонажем, використовуючи нову систему вводу Unity Input System, яка дозволяє легко налаштовувати керування для різних пристроїв. Впроваджено скрипти для руху персонажа, обертання зброї та атаки, що забезпечують точне і плавне керування персонажем та його взаємодію з ворогами. Також розроблено різноманітних ворогів із унікальними характеристиками та поведінкою, що робить гру більш цікавою та захоплюючою.

Процедурна генерація рівнів стала ключовим компонентом для створення унікальних ігрових досвідів з високою реіграбельністю. Впроваджено алгоритм створення мапи рівня, який гарантує неповторність кожного проходження гри. Завершено створення гри додаванням анімацій, звукових ефектів та музики, що значно покращило загальний ігровий досвід, роблячи його більш динамічним та емоційно насиченим.

ВИСНОВКИ

У ході цієї бакалаврської роботи вдалося досягти головної мети, а саме створення 2D гри в жанрі roguelike з динамічною генерацією рівнів за допомогою рушія Unity.

Перш за все, було досліджено жанр roguelike, його історію, особливості та вплив на сучасну ігрову індустрію. Ігри цього жанру відзначаються випадковим створенням рівнів, високим рівнем складності та постійною загрозою смерті персонажа, що робить їх привабливими для гравців завдяки непередбачуваності ігрового процесу.

Далі, було здійснено вибір ігрового рушія. Після аналізу різних рушіїв, таких як Unreal Engine, Godot та GameMaker Studio, було обрано Unity. Цей рушій виявився найоптимальнішим завдяки універсальності, потужним інструментам для роботи з 2D-графікою та підтримці процедурної генерації рівнів.

Розробка гри включала визначення основних геймплейних механік: управління персонажем, бойову систему, систему здоров'я та пошкоджень, а також алгоритм випадкової генерації рівнів. Вороги з унікальними характеристиками та поведінкою додали грі різноманітності та динаміки. Використання нової системи вводу Unity Input System дозволило створити інтуїтивне та точне керування персонажем.

Алгоритм випадкової генерації рівнів став ключовим елементом, забезпечивши унікальні ігрові карти з кожним новим запуском гри, що значно підвищило реіграбельність. Використані методи генерації рівнів, такі як розподіл кімнат, поділ простору, клітинні автомати та графи на основі кімнат, забезпечили різноманітність ігрових світів.

Головне меню та екран смерті персонажа покращили загальний користувацький досвід. Доданий інтерфейс для налаштування звуку дозволяє гравцям регулювати різні параметри звукових ефектів та музики. Анімації та звукові ефекти зробили гру більш реалістичною та захоплюючою, створюючи емоційно насичену атмосферу.

На завершення, впроваджено механізм збору інформації з гри для аналізу ігрового процесу та поведінки гравців, що дозволяє зібрати дані про кожну ігрову сесію для подальшого вдосконалення гри.

Таким чином, розроблена гра демонструє високий рівень реіграбельності, складності та залученості гравців, що робить її конкурентоспроможною на сучасному ринку відеоігор. Перспективи подальшого розвитку проекту включають вдосконалення геймплейних механік, розширення функціоналу гри та інтеграцію нових технологій для покращення ігрового досвіду.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Вікіпедія Roguelike [Електронний ресурс] URL: https://uk.wikipedia.org/wiki/Roguelike (дата звернення: 15.05.2024).
2. Wikipedia Rogue (video game) [Електронний ресурс] URL: https://en.wikipedia.org/wiki/Rogue_(video_game) (дата звернення: 15.05.2024).
3. Game design Wikipedia [Електронний ресурс] URL: https://en.wikipedia.org/wiki/Game_design (дата звернення: 15.05.2024).
4. Механіки "рогаликів" на службі ігрового розробника [Електронний ресурс] URL: https://core-rpg.net/articles/analytics/genre/the-key-design-elements-of-roguelikes (дата звернення: 16.05.2024).
5. Живи, помри, і знову: занурюємося у світ рогаликів [Електронний ресурс] URL: https://habr.com/ru/companies/ruvds/articles/574252/ (дата звернення: 17.05.2024).
6. The Binding of Isaac: Rebirth Wikipedia [Електронний ресурс] URL: https://en.wikipedia.org/wiki/The_Binding_of_Isaac:_Rebirth (дата звернення: 18.05.2024).
7. Dead Cells Wikipedia [Електронний ресурс] URL: https://en.wikipedia.org/wiki/Dead_Cells (дата звернення: 18.05.2024).
8. Hades (game) fandom [Електронний ресурс] URL: https://hades.fandom.com/wiki/Hades_(game) (дата звернення: 18.05.2024).
9. Risk of Rain 2 Wikipedia [Електронний ресурс] URL: https://en.wikipedia.org/wiki/Risk_of_Rain_2 (дата звернення: 18.05.2024).
10. Roguelike манія. Чому рогалики стали такими популярними? [Електронний ресурс] URL: https://dtf.ru/games/2231360-roguelike-maniya-pochemu-rogaliki-stali-tak-populyarny (дата звернення: 20.05.2024).
11. Unity (game engine) Wikipedia [Електронний ресурс] URL: https://en.wikipedia.org/wiki/Unity_(game_engine) (дата звернення: 20.05.2024).
12. Unreal Engine Wikipedia [Електронний ресурс] URL: https://en.wikipedia.org/wiki/Unreal_Engine (дата звернення: 20.05.2024).

13. Godot (game engine) Wikipedia [Електронний ресурс] URL: https://en.wikipedia.org/wiki/Godot_(game_engine) (дата звернення: 20.05.2024).

14. GameMaker Wikipedia [Електронний ресурс] URL: https://en.wikipedia.org/wiki/GameMaker (дата звернення: 20.05.2024).

15. Що таке Unity? [Електронний ресурс] URL: https://lemon.school/blog/shho-take-unity (дата звернення: 21.05.2024).

16. Best Game Development Frameworks: A Comparison Of Unity, Unreal Engine, Cocos2D-X, Godot, & Gamemaker Studio [Електронний ресурс] URL: https://medium.com/@amandubey_6607/best-game-development-frameworks-a-comparison-of-unity-unreal-engine-cocos2d-x-godot-16e5b5226f70 (дата звернення: 21.05.2024).

17. Procedural... house with rooms generator [Електронний ресурс] URL: https://gamedev.stackexchange.com/questions/47917/procedural-house-with-rooms-generator (дата звернення: 24.05.2024).

18. Basic BSP Dungeon generation [Електронний ресурс] URL: https://www.roguebasin.com/index.php/Basic_BSP_Dungeon_generation (дата звернення: 24.05.2024).

19. Cave-like Level Generation Using Cellular Automata Linkedin [Електронний ресурс] URL: https://www.linkedin.com/pulse/cave-like-level-generation-using-cellular-automata-martin-celusniak/ (дата звернення: 24.05.2024).

20. GRAPH-BASED DUNGEON GENERATOR [Електронний ресурс] URL: https://ondra.nepozitek.cz/blog/graph-based-dungeon-generator-basics-1/ (дата звернення: 24.05.2024).

21. Здоров'я (відеоігри) Вікіпедія [Електронний ресурс] URL: https://uk.wikipedia.org/wiki/%D0%97%D0%B4%D0%BE%D1%80%D0%BE%D0%B2%27%D1%8F_(%D0%B2%D1%96%D0%B4%D0%B5%D0%BE%D1%96%D0%B3%D1%80%D0%B8) (дата звернення: 1.06.2024).

22. Game Over, Man: Creating A Game Over Screen In Unity by Antonio Delgado [Електронний ресурс] URL: https://gt3000.medium.com/game-over-man-creating-a-game-over-screen-in-unity-90e1be71cd85 (дата звернення: 1.06.2024).

23. Реалізація ШІ ворога в Unity [Електронний ресурс] URL: https://ru.sharpcoderblog.com/blog/implementing-enemy-ai-in-unity (дата звернення: 1.06.2024).

24. Input System [Електронний ресурс] URL: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.8/manual/index.html (дата звернення: 4.06.2024).

25. Quick start guide [Електронний ресурс] URL: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/QuickStartGuide.html (дата звернення: 4.06.2024).

26. What Is A Roguelike? The Beginner's Guide [Електронний ресурс] URL: https://citizenside.com/technology/what-is-a-roguelike-the-beginners-guide/ (дата звернення: 4.06.2024).

27. How to Create a Start Menu in Unity [Електронний ресурс] URL: https://gamedevacademy.org/unity-start-menu-tutorial/ (дата звернення: 8.06.2024).

28. Full Unity 2D Game Tutorial 2019 – Main Menu [Електронний ресурс] URL: https://www.gamedevelopment.blog/unity-2d-game-tutorial-2019-main-menu/ (дата звернення: 8.06.2024).

ДОДАТОК А

Код генерації рівня

Вміст файлу LevelGenerator.cs:

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
public class LevelGenerator : MonoBehaviour
{
    [SerializeField] private int numberOfRooms = 4;
    [SerializeField] private Vector2 roomSize = new Vector2(3, 3);
    [SerializeField] private GameObject pointObject;

    [SerializeField] private GameObject spawnT;
    [SerializeField] private GameObject spawnB;
    [SerializeField] private GameObject spawnL;
    [SerializeField] private GameObject spawnR;

    [SerializeField] private GameObject exitT;
    [SerializeField] private GameObject exitB;
    [SerializeField] private GameObject exitL;
    [SerializeField] private GameObject exitR;


    [SerializeField] private GameObject[] rooms;


    private List<GameObject> placeForRooms;
    private HashSet<Vector2> occupiedPositions;

    private List<GameObject> roomList;
    private void Start()
    {
        placeForRooms = new List<GameObject>();
        occupiedPositions = new HashSet<Vector2>();

        roomList = new List<GameObject>();

        GenerateLevel();
    }

    private void GenerateLevel()
    {
        numberOfRooms = GameManager.Instance.numbersOfRooms;
        if (numberOfRooms < 4)
        {
            numberOfRooms = 4;
            Debug.Log("Мінімальне значення кімнат 4!");
        }
        ClearMap();
        CreateMap();
        PlaceRooms();
        ClearPoints();
    }
    private void ClearMap()
    {
        ClearGameObjectList(placeForRooms);
        ClearGameObjectList(roomList);
        occupiedPositions.Clear();
    }
```

```
    private void ClearGameObjectList(List<GameObject> list)
    {
        foreach (GameObject obj in list)
        {
            Destroy(obj);
        }
        list.Clear();
    }
    private void ClearPoints()
    {
        ClearGameObjectList(placeForRooms);
    }

    private void CreateMap()
    {
        AddRoom(pointObject.transform.position);

        GameObject currentPosition = placeForRooms[0];
        int errorCounter = 1;
        int attempts = 0;
        int maxAttempts = 20;

        Vector3[] directions = {
            new Vector3(0, roomSize.y, 0),  // Вверх
            new Vector3(0, -roomSize.y, 0), // Низ
            new Vector3(-roomSize.x, 0, 0), // Ліворуч
            new Vector3(roomSize.x, 0, 0)   // Праворуч
        };

        while (placeForRooms.Count < numberOfRooms)
        {
            int randomIndex = Random.Range(0, directions.Length);
            Vector3 direction = directions[randomIndex];
            Vector3 newPosition = currentPosition.transform.position + direction;

            if (!occupiedPositions.Contains(newPosition))
            {
                AddRoom(newPosition);
                currentPosition = placeForRooms[placeForRooms.Count - 1];
                attempts = 0; // Скидання кількості помилок, оскільки успішно додано кімнату
            }
            else
            {
                attempts++;
                if (attempts > maxAttempts)
                {
                    RemoveQuarterRooms();
                    if (placeForRooms.Count > 0)
                    {
                        currentPosition = placeForRooms[placeForRooms.Count - 1];
                    }
                    errorCounter++;
                    attempts = 0;
                }
                if (errorCounter > 20)
                {
                    Debug.LogError("Помилка генерації карти: не вдалося створити необхідну кількість кімнат, хоча ліміт
спроб не досягнуто.");
                    break;
                }
            }
```

```
    }
}

private void AddRoom(Vector3 position)
{
    GameObject newRoom = Instantiate(pointObject);
    newRoom.transform.position = position;
    placeForRooms.Add(newRoom);
    occupiedPositions.Add(position);
}

private void RemoveQuarterRooms()
{
    int roomsToRemove = placeForRooms.Count / 4;
    for (int i = 0; i < roomsToRemove; i++)
    {
        int lastIndex = placeForRooms.Count - 1;
        Destroy(placeForRooms[lastIndex]);
        placeForRooms.RemoveAt(lastIndex);
    }

    occupiedPositions.Clear();
    foreach (GameObject room in placeForRooms)
    {
        occupiedPositions.Add(room.transform.position);
    }
}
void PlaceRooms()
{
    Vector2 firstRoomPosition = placeForRooms[0].transform.position;
    Vector2 secondPosition = placeForRooms[1].transform.position;
    Vector2 preLastPosition = placeForRooms[placeForRooms.Count - 2].transform.position;
    Vector2 lastPosition = placeForRooms[placeForRooms.Count - 1].transform.position;

    PlaceSpawnRoom(secondPosition, firstRoomPosition);
    PlaceSecondRoom(secondPosition, lastPosition);

    for (int counter = 2; counter < placeForRooms.Count - 2; counter++)
    {
        Vector2 currentPosition = placeForRooms[counter].transform.position;
        PlaceIntermediateRoom(currentPosition, firstRoomPosition, lastPosition);
    }

    PlacePreLastRoom(preLastPosition, firstRoomPosition);
    PlaceExitRoom(preLastPosition, lastPosition);
}

void PlaceSpawnRoom(Vector2 secondPosition, Vector2 spawnPosition)
{
    GameObject spawnRoom = null;

    if (secondPosition == spawnPosition + new Vector2(0, roomSize.y)) // Верх
    {
        spawnRoom = Instantiate(spawnT, spawnPosition, Quaternion.identity);
    }
    else if (secondPosition == spawnPosition + new Vector2(0, -roomSize.y)) // Низ
    {
        spawnRoom = Instantiate(spawnB, spawnPosition, Quaternion.identity);
    }
    else if (secondPosition == spawnPosition + new Vector2(-roomSize.x, 0)) // Ліво
    {
        spawnRoom = Instantiate(spawnL, spawnPosition, Quaternion.identity);
```

```
        }
        else if (secondPosition == spawnPosition + new Vector2(roomSize.x, 0)) // Право
        {
            spawnRoom = Instantiate(spawnR, spawnPosition, Quaternion.identity);
        }

        if (spawnRoom != null)
        {
            roomList.Add(spawnRoom);
            occupiedPositions.Add(spawnPosition);
        }
        else
        {
            Debug.LogError("Помилка розміщення кімнати спавну. Не вдалося знайти напрямок.");
        }
    }
    void PlaceSecondRoom(Vector2 secondPosition, Vector2 lastPosition)
    {
        bool isTop = occupiedPositions.Contains(secondPosition + new Vector2(0, roomSize.y)) && secondPosition + new
Vector2(0, roomSize.y) != lastPosition;
        bool isBottom = occupiedPositions.Contains(secondPosition + new Vector2(0, -roomSize.y)) && secondPosition + new
Vector2(0, -roomSize.y) != lastPosition;
        bool isLeft = occupiedPositions.Contains(secondPosition + new Vector2(-roomSize.x, 0)) && secondPosition + new
Vector2(-roomSize.x, 0) != lastPosition;
        bool isRight = occupiedPositions.Contains(secondPosition + new Vector2(roomSize.x, 0)) && secondPosition + new
Vector2(roomSize.x, 0) != lastPosition;

        GameObject matchingRoom = rooms.FirstOrDefault(room =>
            room.GetComponentInChildren<RoomInfo>().IsTop == isTop &&
            room.GetComponentInChildren<RoomInfo>().IsBottom == isBottom &&
            room.GetComponentInChildren<RoomInfo>().IsLeft == isLeft &&
            room.GetComponentInChildren<RoomInfo>().IsRight == isRight
        );

        if (matchingRoom != null)
        {
            matchingRoom.transform.position = secondPosition;
            roomList.Add(Instantiate(matchingRoom));
            occupiedPositions.Add(secondPosition);
        }
        else
        {
            Debug.LogError($"Помилка розміщення кімнати. Не вдалося знайти потрібну кімнату для позиції:
{secondPosition}");
        }
    }

    void PlacePreLastRoom(Vector2 preLastPosition, Vector2 firstRoomPosition)
    {
        bool isTop = occupiedPositions.Contains(preLastPosition + new Vector2(0, roomSize.y)) && preLastPosition + new
Vector2(0, roomSize.y) != firstRoomPosition;
        bool isBottom = occupiedPositions.Contains(preLastPosition + new Vector2(0, -roomSize.y)) && preLastPosition +
new Vector2(0, -roomSize.y) != firstRoomPosition;
        bool isLeft = occupiedPositions.Contains(preLastPosition + new Vector2(-roomSize.x, 0)) && preLastPosition + new
Vector2(-roomSize.x, 0) != firstRoomPosition;
        bool isRight = occupiedPositions.Contains(preLastPosition + new Vector2(roomSize.x, 0)) && preLastPosition + new
Vector2(roomSize.x, 0) != firstRoomPosition;

        GameObject matchingRoom = rooms.FirstOrDefault(room =>
            room.GetComponentInChildren<RoomInfo>().IsTop == isTop &&
            room.GetComponentInChildren<RoomInfo>().IsBottom == isBottom &&
            room.GetComponentInChildren<RoomInfo>().IsLeft == isLeft &&
```

```
        room.GetComponentInChildren<RoomInfo>().IsRight == isRight
    );

    if (matchingRoom != null)
    {
        matchingRoom.transform.position = preLastPosition;
        roomList.Add(Instantiate(matchingRoom));
        occupiedPositions.Add(preLastPosition);
    }
    else
    {
        Debug.LogError($"Помилка розміщення кімнати. Не вдалося знайти потрібну кімнату для позиції: {preLastPosition}");
    }
}

void PlaceIntermediateRoom(Vector2 currentPosition, Vector2 firstRoomPosition, Vector2 lastPosition)
{
    bool isTop = occupiedPositions.Contains(currentPosition + new Vector2(0, roomSize.y)) && firstRoomPosition != currentPosition + new Vector2(0, roomSize.y) && lastPosition != currentPosition + new Vector2(0, roomSize.y);
    bool isBottom = occupiedPositions.Contains(currentPosition + new Vector2(0, -roomSize.y)) && firstRoomPosition != currentPosition + new Vector2(0, -roomSize.y) && lastPosition != currentPosition + new Vector2(0, -roomSize.y);
    bool isLeft = occupiedPositions.Contains(currentPosition + new Vector2(-roomSize.x, 0)) && firstRoomPosition != currentPosition + new Vector2(-roomSize.x, 0) && lastPosition != currentPosition + new Vector2(-roomSize.x, 0);
    bool isRight = occupiedPositions.Contains(currentPosition + new Vector2(roomSize.x, 0)) && firstRoomPosition != currentPosition + new Vector2(roomSize.x, 0) && lastPosition != currentPosition + new Vector2(roomSize.x, 0);

    GameObject matchingRoom = rooms.FirstOrDefault(room =>
        room.GetComponentInChildren<RoomInfo>().IsTop == isTop &&
        room.GetComponentInChildren<RoomInfo>().IsBottom == isBottom &&
        room.GetComponentInChildren<RoomInfo>().IsLeft == isLeft &&
        room.GetComponentInChildren<RoomInfo>().IsRight == isRight
    );

    if (matchingRoom != null)
    {
        matchingRoom.transform.position = currentPosition;
        roomList.Add(Instantiate(matchingRoom));
        occupiedPositions.Add(currentPosition);
    }
    else
    {
        Debug.LogError($"Помилка розміщення кімнати. Не вдалося знайти потрібну кімнату для позиції: {currentPosition}");
    }
}

void PlaceExitRoom(Vector2 preLastPosition, Vector2 exitPosition)
{
    GameObject exitRoom = null;

    if (preLastPosition == exitPosition + new Vector2(0, roomSize.y)) // Верх
    {
        exitRoom = Instantiate(exitT, exitPosition, Quaternion.identity);
    }
    else if (preLastPosition == exitPosition + new Vector2(0, -roomSize.y)) // Низ
    {
        exitRoom = Instantiate(exitB, exitPosition, Quaternion.identity);
    }
    else if (preLastPosition == exitPosition + new Vector2(-roomSize.x, 0)) // Ліво
    {
        exitRoom = Instantiate(exitL, exitPosition, Quaternion.identity);
```

```
        }
        else if (preLastPosition == exitPosition + new Vector2(roomSize.x, 0)) // Право
        {
            exitRoom = Instantiate(exitR, exitPosition, Quaternion.identity);
        }

        if (exitRoom != null)
        {
            roomList.Add(exitRoom);
            occupiedPositions.Add(exitPosition);
        }
        else
        {
            Debug.LogError("Помилка розміщення кімнати виходу. Не вдалося знайти потрібний напрямок.");
        }
    }


#if UNITY_EDITOR
    [UnityEditor.CustomEditor(typeof(LevelGenerator))]
    public class LevelGeneratorEditor : UnityEditor.Editor
    {
        public override void OnInspectorGUI()
        {
            base.OnInspectorGUI();

            LevelGenerator generator = (LevelGenerator)target;
            if (GUILayout.Button("Generate Level"))
            {
                generator.GenerateLevel();
            }
        }
    }
#endif
}
```

Вміст фалу RoomInfo.cs:

```
using UnityEngine;
public class RoomInfo : MonoBehaviour
{
    [SerializeField] private bool isTop;
    [SerializeField] private bool isBottom;
    [SerializeField] private bool isLeft;
    [SerializeField] private bool isRight;
    public bool IsTop { get { return isTop; } }
    public bool IsBottom { get { return isBottom; } }
    public bool IsLeft { get { return isLeft; } }
    public bool IsRight { get { return isRight; }
```