МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Покращення ігрової механіки комп'ютерних ігор жанру FPS Roguelike Game**

Виконав: студент 2 курсу, групи 8.1214-іпз-2
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
І.О. Красножон
(ініціали та прізвище)
Керівник доцент, к.ф.-м.н., доцент І.А. Скрипник
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент Директор ТОВ *«Дісітел»* П.О.Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра __електроніки, інформаційних систем та програмного__ забезпечення
Рівень вищої освіти _____другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри _____Т.В. Критська_
"_01_" __квітня_____2025_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Красножону Івану Олеговичу
(прізвище, ім'я, по батькові)

1.Тема роботи __Покращення ігрової механіки комп'ютерних ігор жанру FPS Roguelike Game

керівник роботи ___к.ф.-м.н., доцент Скрипник Ірина Анатоліївна
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від _28.03.2025 р. № 361-е__
2. Строк подання студентом кваліфікаційної роботи_____01.12.2025 р.___
3. Вихідні дані магістерської роботи

- комплект нормативних документів ;
- технічне завдання до роботи.

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

- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- створення програмного продукту та його опис;
- дослідження проблеми процедурної генерації рівнів
- перелік вимог для роботи програми;
- дослідження поставленої проблеми та розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
_____слайдів презентації

6. Консультанти розділів магістерської роботи

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

7. Дата видачі завдання 01.04.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником.	15.04 – 20.04.25	виконано
2	Літературний огляд. Перший розділ. Жанр гри FPS Roguelike.	21.04 – 30.04.25	виконано
3	Літературний огляд. Другий розділ. Огляд аналогічних рішень, аналіз.	01.05 – 10.05.25	виконано
4	Третій розділ. Вибір технологій проектування та особливості архітектури програмного забезпечення.	11.05 – 20.05.25	виконано
6	Розробка застосунка на мові C# для дослідження створення відеогри в жанрі FPS Roguelike на Unity.	21.05 – 06.08.25	виконано
7	Створення системи генерації ігрових рівнів.	01.09 – 30.09.25	виконано
8	Створення контролера гравця, нанесення шкоди, модифікатори, система ворогів.	01.10 – 15.11.25	виконано
9	Тестування, виправлення багів, створення збірки гри.	15.11 – 20.11.25	виконано
10	Оформлення звіту.	20.11 – 27.11.25	виконано

Студент _____ Красножон І. О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Скрипник І. А.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник І.А,__
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок: 100

Рисунків: 8

Джерел: 17

Красножон І. О. Покращення ігрової механіки комп’ютерних ігор жанру FPS Roguelike Game: кваліфікаційна робота магістра спеціальності 121 Інженерія програмного забезпечення / наук. керівник І. А. Скрипник. Запоріжжя : ЗНУ, 2025. 102 с.

Мета і завдання дослідження полягають у вивченні сучасних підходів до розробки ігор в жанрі FPS Roguelike, а також у розробці систем для генерації рівнів для створення унікальних рівнів для кожного забігу в грі.

У процесі дослідження була розглянута проблема створення гри в жанрі FPS Roguelike з процедурною генерацією рівнів. Як результат було розроблено систему для генерації ігрових рівнів з різними типами кімнат, проходами між ними, різні типи ворогів, модифікатори для атак ворогів, атак зброєю гравця.

Дана гра повинна також запускатись на широкому спектрі персональних комп’ютерів, що дозволить охопити більшу кількість гравців для розширення аудиторії даного проєкту.

Ключові слова: гра, генерація рівнів, штучний інтелект ворогів, ігрові системи, ігрові події, ігровий рушій.

SUMMARY

Pages: 100

Figures: 8

Sources: 17

Krasnozhon I. O. Improving the game mechanics of computer games in the FPS Roguelike Game genre: Master's Qualification Thesis in Specialty 121 Software Engineering / scientific supervisor I. A. Skrypnyk. Zaporizhzhia : ZNU, 2025. 102 p.

The goal and objectives of the research are to study modern approaches to the development of games in the FPS Roguelike genre, as well as to develop systems for level generation to create unique levels for each run in the game.

In the course of the research, the problem of creating an FPS Roguelike game with procedural level generation was considered. As a result, a system was developed for generating game levels with different room types, passages between them, various enemy types, modifiers for enemy attacks, and player weapon attacks.

This game should also be runnable on a wide range of personal computers, which will allow for a greater number of players to be reached, expanding the audience of this project.

Key words: *game, level generation, enemy artificial intelligence, game systems, game events, game engine*.

ЗМІСТ

ВСТУП 8
РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ СТВОРЕННЯ ГРИ В ЖАНРІ FPS ROGUELIKE 13
1.1 Огляд проблеми створення гри в жанрі FPS Roguelike 13
1.2 Процедурна генерація рівнів як ключова проблема проєкту 13
1.3 Роль та відображення гравця в жанрі Roguelike 14
1.4 Функції та еволюція ворогів в Roguelike іграх 14
1.5 Модифікатори як засіб урізноманітнення геймплею 15
1.6 Деталізація ігрових механік, пов’язаних з FPS елементом 16
1.6.1 Системи зброї та процедурна варіативність 16
1.6.2 Динаміка руху та маневреність гравця 17
1.6.3 Відчуття стрільби та взаємодія з оточенням 17
1.6.4 Інтерфейс та надання інформації гравцю 18
1.7 Огляд існуючих відеоігор в жанрі FPS Roguelike 19
1.7.1 Gunfire Reborn 19
1.7.2 Roboquest 20
1.7.3 Deadlink 21
1.7.4 BPM: Bullets Per Minute 22
1.7.5 Immortal Redneck 23
1.8 Постановка завдання кваліфікаційної роботи 24
1.9 Висновки до першого розділу 25
2 МЕТОДИ ТА ЗАСОБИ, ВИКОРИСТАНІ ПРИ ВИКОНАННІ РОБОТИ 26
2.1 Огляд літературних джерел 26
2.2 Ігровий рушій Unity 28
2.3 Visual Studio 35
2.4 Висновки до розділу 2 35
3 ОПИС СТВОРЕННЯ ВІДЕОГРИ ЖАНРУ FPS ROGUELIKE 36
3.1 Опис предметної області роботи та її мета 36

3.2 Опис систем, котрі необхідно реалізувати 37
3.2.1 Система модифікаторів гравця 37
3.2.2 Система переміщення гравця 38
3.2.3 Система ворогів 38
3.2.4 Система кімнат 39
3.2.5 Система генерації рівнів 39
3.2.6 Система нанесення шкоди об’єктам 40
3.3 Вимоги до апаратного забезпечення системи 40
3.4 Вимоги до програмного забезпечення системи 41
3.5 Вимоги до користувачів відеогри 41
3.6 Висновки до розділу 3 42
4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ ВИКОНАНОЇ РОБОТИ 44
4.1 Реалізація відеогри в жанрі FPS Roguelike на ігровому рушії Unity 44
4.1.1 Система переміщення гравця 44
4.1.2 Система модифікаторів 53
4.1.3 Система отримання шкоди 66
4.1.4 Системи ворогів 68
4.1.5 Система генерації рівня 78
4.1.6 Ігровий менеджмент 94
4.2 Висновки до розділу 4 95
4.3 Що можна покращити в виконаній роботі 96
ВИСНОВКИ 97
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 99

ВСТУП

Актуальність теми

У сучасному світі індустрія відеоігор є однією з найдинамічніших та найприбутковіших галузей розваг. Жанр FPS (First-Person Shooter) Roguelike поєднує в собі інтенсивність шутерів від першої особи з елементами процедурної генерації, перманентної смерті та нелінійного прогресу, що забезпечує гравцям унікальний досвід кожного проходження. Популярність цього жанру зростає, проте інтеграція різноманітних ігрових механік у єдиний збалансований продукт залишається складною задачею. Відсутність систематизованих підходів до поєднання динамічного бою, адаптивного штучного інтелекту (ШІ), процедурної генерації рівнів та інших елементів обмежує потенціал створення глибокого та варіативного ігрового процесу.

Саме тому дослідження взаємодії цих механік у межах одного жанру є актуальним як для теоретиків ігрового дизайну, так і для практиків-розробників.

Історичний контекст та еволюція жанру Roguelike

Жанр Roguelike зародився у 1980-х роках із грою Rogue, яка впровадила ключові механіки: процедурну генерацію, перманентну смерть та нелінійний прогрес. З часом жанр еволюціонував, інтегруючи елементи інших жанрів, таких як платформери (Spelunky), рольові ігри (Darkest Dungeon) та шутери (Risk of Rain 2). Особливу увагу привернули гібридні форми, зокрема FPS Roguelike, де поєднуються динамічний екшен від першої особи з випадковістю та стратегічним плануванням. Наприклад, у грі Gunfire Reborn вдало інтегровані процедурні рівні, різноманітні бонуси та адаптивні вороги, що робить її важливим дослідницьким кейсом для аналізу балансу між випадковістю та складністю.

Ця еволюція свідчить про те, що сучасні Roguelike-ігри не лише зберігають класичні механіки, але й адаптують їх до нових жанрів, що підкреслює

актуальність дослідження інтеграції різноманітних систем у межах FPS Roguelike.

Аналіз існуючих ігор жанру FPS Roguelike

Для глибшого розуміння проблематики доцільно звернутися до аналізу конкретних ігор, які вже реалізують елементи FPS Roguelike:

1. Risk of Rain 2 — демонструє ефективне поєднання процедурної генерації рівнів з кооперативним геймплеєм, де випадковість предметів створює унікальні стратегії кожного забігу. Недоліком є відсутність глибокого адаптивного ШІ, що обмежує тактичну різноманітність супротивників.
2. Gunfire Reborn — акцент на комбінації різних типів зброї та модифікацій, що впливають на стиль гри. Однак система постійного прогресу тут лінійна, що зменшує реіграбельність.
3. Immortal Redneck — використовує класичні механіки Roguelike у 3D-середовищі, але обмежений у варіативності оточень та інтерактивності рівнів.

Ці приклади підкреслюють, що навіть успішні ігри стикаються з проблемами балансу між випадковістю, складністю та глибиною прогресу. Це підтверджує необхідність системного підходу до інтеграції механік, запропонованого у даній роботі.

Мета та завдання дослідження

Метою роботи є комплексний аналіз взаємодії ключових ігрових механік у жанрі FPS Roguelike та розробка рекомендацій щодо їх гармонійної інтеграції. Для досягнення цієї мети визначено такі завдання:

1. Дослідити теоретичні основи процедурної генерації рівнів, динамічного бою, систем постійного прогресу та перманентної смерті.
2. Проаналізувати вплив штучного інтелекту ворогів, варіативності зброї та інтерактивного оточення на ігровий досвід.
3. Розробити модель взаємодії механік, що забезпечує баланс між випадковістю та стратегічним плануванням.

4. Реалізувати прототип гри, що демонструє ефективність запропонованих рішень.
5. Оцінити результати експериментальної перевірки та сформулювати рекомендації для подальших досліджень.

Об’єкт і предмет дослідження

- Об’єкт дослідження: ігровий процес у жанрі FPS Roguelike.
- Предмет дослідження: механіки процедурної генерації рівнів, динамічного бою, системи постійного прогресу, перманентної смерті та їх взаємозв’язок у формуванні ігрового досвіду.

Методологія дослідження

Дослідження ґрунтується на поєднанні таких методів:

- Системний аналіз — вивчення взаємозв’язків між механіками та їх впливу на баланс гри.
- Процедурне моделювання — створення алгоритмів генерації рівнів та поведінки ШІ.
- Експериментальне тестування — оцінка варіативності та глибини ігрового процесу.
- Порівняльний аналіз — визначення переваг та недоліків існуючих аналогів.

Наукова новизна

Наукова новизна дослідження полягає в:

1. Удосконаленні алгоритмів процедурної генерації рівнів з урахуванням тактичних можливостей гравця.
2. Реалізації адаптивного ШІ, здатного реагувати на зміни в стратегії гравця.
3. Розробці системи модифікацій зброї, що поєднує випадковість зі стратегічним вибором.

Технологічні виклики в розробці FPS Roguelike

Створення FPS Roguelike пов’язане з такими технічними складнощами:

- Оптимізація процедурної генерації рівнів.
- Адаптація ШІ до динамічного середовища.
- Балансування системи прогресу.

Досвід розробників Dead Cells показує, що ключем до успіху є ітеративне тестування механік із залученням спільноти.

Соціокультурний вплив жанру Roguelike-ігри:

- Популяризують ідею "навчання через помилки".
- Стимулюють обмін знаннями через спільноти (Hades, Slay the Spire).
- Стають платформою для експериментів (Returnal).

Практична значимість

Результати дослідження можуть бути використані:

- Розробниками для створення інноваційних ігор.
- Теоретиками для аналізу механік.
- Викладачами як навчальний матеріал.

Глосарій

FPS (First-Person Shooter) – жанр відеоігор, де гравець бачить ігровий світ від першої особи, а основним елементом геймплею є стрільба.

Roguelike – жанр відеоігор, що характеризується процедурною генерацією рівнів, перманентною смертю та нелінійним прогресом.

Процедурна генерація – метод створення ігрового контенту (рівнів, предметів, ворогів) за допомогою алгоритмів, що забезпечує унікальність кожного проходження.

Перманентна смерть – механіка, за якої після смерті персонажа гравець втрачає частину або весь прогрес і починає гру з початку.

Нелінійний прогрес – система розвитку гравця, де вибір навичок, предметів або маршрутів не є строго визначеним і дозволяє різні стратегії проходження.

Штучний інтелект (ШІ) – алгоритми, що керують поведінкою неігрових персонажів (ворогів, союзників), імітуючи розумні дії.

Реіграбельність – характеристика гри, що показує, наскільки вона залишається цікавою після багаторазового проходження.

Баланс гри – система налаштувань, що забезпечує справедливість і рівновагу між різними механіками (наприклад, складністю, силою зброї, поведінкою ворогів).

Ігрові механіки – основні правила та системи, що визначають взаємодію гравця з ігровим світом (рух, бій, прогрес тощо).

Адаптивність – здатність гри або її елементів (наприклад, ШІ) змінюватися у відповідь на дії гравця.

Імерсивність – рівень занурення гравця в ігровий світ, досягнутий через візуальні, звукові та геймплейні елементи.

Прототип гри – рання версія гри, створена для тестування ключових механік та ідей.

Мета (у контексті ігор) – оптимальна стратегія або побудова персонажа, що вважається найефективнішою для досягнення цілей гри.

Структура роботи

Робота містить:

1. Вступ.
2. Теоретичний аналіз жанру FPS Roguelike.
3. Проектування ключових механік.
4. Практична реалізація прототипу.
5. Результати тестування та рекомендації.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ СТВОРЕННЯ ГРИ В ЖАНРІ FPS ROGUELIKE

1.1 Огляд проблеми створення гри в жанрі FPS Roguelike

В сучасному світі створення відеоігор є досить складною задачею і для створення гри практичного будь’якого жанру може знадобитись дуже багато часу. Звичайно, деяким проєктам для досягнення стану MVP може знадобитись декілька тижнів, а деякі ігрові проєкти можуть розроблятись по 5 або навіть більше років. Жанр Roguelike також не є виключенням. Даний жанр являється досить варіативним і через це в ньому можуть поєднуватись десятки, якщо не сотні механік.

1.2 Процедурна генерація рівнів як ключова проблема проєкту

В даному випадку основною проблемою створення даного проєкту є дослідження способів процедурної генерації рівнів для гри, щоб додати більше варіативності для гри, так-як це допоможе в майбутньому додавати нові варіації рівнів для гри з різними механіками, візуальним стилем і так далі без необхідності витрачати десятки годин для створення нових рівнів. Сама задача генерації ігрових рівнів уже вирішується в ігровій індустрії досить давно. Для прикладу можна взяти гру Dwarf Fortress, котра в своїй основі полягається на процедурну генерацію рівня в багатьох аспектах гри. Наприклад за допомогою процедурної генерації рівнів в ній створюється не тільки ігровий світ з різноманітними географічними особливостями, рельєфом і геологією, котрі впливають на ігровий процес, а ще й створюються випадкові події, сутності, артефакти і так далі. Також в Roguelike іграх окрім рівнів існує гравець, вороги та інші ігрові сутності котрі намагаються робити ігровий процес більш різноманітним, та спонукати гравця проходити гру знову і знову.

1.3 Роль та відображення гравця в жанрі Roguelike

Гравець в іграх описуваного жанру може бути максимально різним. Якщо взяти в приклад гру Risk of Rain 2, то в ній гравець відображається від третьої особи, є можливість вільно оглядати навколишній світ та шукати свої цілі. Таким же чином відбувається керування гравцем в грі Crab Champions, але в ній більш динамічне керування гравцем та швидші атаки ворогів, що надає незабутні емоції від даної гри, в особливості при її перепроходженню на максимальній складності з великою кількістю ввімкнених модифікаторів, таких як мультиплікатор шкоди по гравцю, менша кількість шкоди по ворогам, або наприклад те, що гравець може померти від одного удару ворога, що дозволяє напружити гравця ще більше. А якщо взяти в приклад гру The Bind of Isaac, то в ній гравець відображається видом зверху і він відразу може бачити всю ситуацію на рівні та ефективно знищувати ворогів одночасно ухиляючись від різноманітних атак від вищезгаданих сутностей. Також якщо взяти першу частину серії Risk of Rain, то в ній гравець керується з видом з боку і тому в ній ти не бачиш що буде далі відразу і в ній це працює як туман війни в деяких стратегіях, що додає відчуття незвіданого до гри.

1.4 Функції та еволюція ворогів в Roguelike іграх

Вороги в даних іграх також грають одну із основних ролей, так-як від них буде залежати те як буде діяти гравець в тих чи інших ситуаціях. Вороги в таких іграх бувають як і гравці дуже різноманітними. Починаючи від ворогів в покрокових Roguelike іграх, закінчуючи комплексними босами в Risk Of Rain 2, котрий може мати по декілька фаз, кожна з яких може мати різноманітні атаки та уміння. Загалом вороги в таких іграх поступово посилюються. Найчастіше на початку дають досить повільних ворогів, котрі атакують гравця в ближньому бою, а також часто додають в пару до ворогів ближнього бою якихось простих ворогів середньої дальності, котрі мають меншу кількість

здоров’я, але так-як це лише початок гри, то їх снаряди рухаються досить повільно і не наносять багато шкоди гравцю. Далі вороги поступово посилюються в залежності наприклад від ігрового часу який гравець уже провів у поточному забігу. В такому випадку чим довше він грає тим сільніші вороги стають, вони мають більшу швидкість атак, швидше переміщаються, та наносять в колосальнії різниці більше шкоди гравцю. Через це наприклад Risk of Rain необхідно проходити як можна швидше, так-як уже наприклад на годині часу проходження забігу вороги фінального рівня майже знищують гравця за один удар, через це необхідно для балансу та різноманітності гри внести в неї ще одну змінну - модифікатори.

1.5 Модифікатори як засіб урізноманітнення геймплею

Модифікатори можуть дуже сильно урізноманітнити гру. Вони можуть активуватися майже від будь-яких подій в грі, таких як переміщення гравця, стрибок, атака гравця, приземлення гравця, попадання гравцем по ворогу або по блокам рівня, а також дані модифікатори можуть просто додавати щось нове в гру. Наприклад часто можна побачити різні часточки котрі літають навколо гравця, або наприклад ворогів і вони просто наносять шкоду ворогам.

Дані модифікатори після своєї активації можуть робити дуже різні речі. Наприклад регенерувати здоров’я гравця, збільшувати шкоду по ворогам в 2 рази, але зменшити максимальне здоров’я гравця в 2 рази. Також часто роблять так, що окрім основного снаряду який може вилітати зі зброї гравця можуть вилітати додаткові, котрі можуть наприклад накладати різноманітні ефекти на ворогів або гравця.

Деякі модифікатори роблять досить приземлені зміни. Наприклад змінюють швидкість переміщення, атаки гравця або наприклад впливають на його стрибки, наприклад - зменшують гравітацію гравця, щоб він міг стрибати більш далеко і знаходитись в повітрі довше, що може допомогти частіше активувати модифікатори котрі активуються лише коли гравець знаходиться в

повітрі. Це лише капля в морі опису того як і які модифікатори можуть впливати на ігровий процес та уповільнювати або пришвидшувати геймплей гри одночасно підсилюючи або послаблюючи деякі властивості гравця.

1.6 Деталізація ігрових механік, пов'язаних з FPS елементом

1.6.1 Системи зброї та процедурна варіативність

В іграх в жанрі "шутер від першої особи" далі *FPS* також грає одну з основних ролей. Дана зброя частіше всього являється зброю дальної дії. Наприклад вогнепальна зброя, луки, магічна зброя, але в деяких випадках бувають комбіновані види зброї. Наприклад меч, як ближня зброя і магія як зброя дальньої дії і все це може варіюватися в залежності від задумки того як повинна відчуватися гра по задумці геймдизайнера щоб найголовніше в це було весело грати.

В Roguelike FPS іграх дял варіативності зброя також може бути унікальною. Якщо розглянути це базово то в них можуть варіюватися базові параметри зброї, такі як шкода, скорострільність, дальність, розмір обойми, час перезарядки. В багатьох випадках вони можуть навіть бути рандомізованими.

Також на зброю можуть впливати вищезгадані модифікатори, котрі можуть змінити відчуття від поточної зброї до невзнаваності. Наприклад додати кулям ефект заморозки, пришвидшити перезарядку або наприклад зброя може отримуваи випадкові або комбіновані ефекти. Наприклад проникаючі кулі, ланцюгові блискавки, сповільнення ворогів. Якщо взяти реальний приклад то в грі Gunfire Reborn зброя має випадкові написи (inscriptions), які додають унікальні бонуси для зброї.

Часто також зброя буває різних класів, таких як пістолети, дробовики, гвинтівки, гранатомети, магічні палиці та посохи, просто використання рук головного героя як магічної зброї. Все це може дуже сильно впливати на їх застосування в умовах динамічного Roguelike-геймплею.

1.6.2 Динаміка руху та маневреність гравця

У FPS Roguelike іграх рух часто є таким же важливим як і стрільба, так-як гравець повинен постійно ухилятись від атак ворогів, котрих з кожним рівнем може ставати більше і кожен з них може постійно посилюватись.

Якщо подивитися на базовий рівень переміщення FPS гравця, то воно включає в себе рух та швидкість руху, стрибок та іноді присідання гравця. Вони являються найосновнішими елементами переміщення гравця і можуть дуже сильно відрізнятися в різних жанрах, але в Roguelike іграх його частіше роблять швидким да досить передбачуваним для забезпечення необхідного контроля над гравцем під час забігів по динамічним рівням гри.

Для додання нових відчуттів від гри в такому жанрі механіки переміщення гравця часто розширяють додаючи в нього нові механіки. Найчастіше в FPS іграх можна зустріти такі механіки переміщення котрі доповнюють його:

- Ривки(Dashes): використання швидких коротких переміщень для ухилення від снарядів або для більш швидкого переміщення по рівню.
- Подвійні або потрійні стрибки (Double / Triple Jumps): додавання можливості гравцю для вертикальної мобільності, котра часто може допомогти в навігації в процедурно згенерованих рівнях і не тільки.
- Ковзання(Sliding): механіки, що дозволяють зберігати інерцію та швидкість під час переміщення в бою.

Наприклад у грі Crab Champions динамічне переміщення гравця і часте використання ривків є ключовими механіками для виживання через високу динамічність бою з ворогами.

1.6.3 Відчуття стрільби та взаємодія з оточенням

Так-як гравець постійно користується механікою стрільби в іграх описуваного жанру то вони мають відчуватися як атаки. Це дозволить посилити емоційне напруження від гри. Дане відчуття можна посилити найрізноманітнішими ігровими механіками.

По-перше, критично важливими для відчуття зброї передати сам постріл зброї. Простими словами повинен бути візуальний та звуковий відгук від дій гравця пов’язаними зі зброєю. Спалахи, трасери, звуки пострілів та ефектів влучання, котрі навіть можуть передавати потужність зброї. Все це намагається посилити емоційний вплив від використання тієї чи іншої зброї. Але одними ефектами гравець ніколи не буде ситим, так-як якщо зброя занадто легка в використанні то більшості гравцям може досить швидко набристи така гра, так-як у зброї повинен бути свій виклик. Тут на допомогу приходять ефекти віддачі та розкиду. Дані механіки призвані повпливати на точність і баланс зброї, особливо під час швидкої стрільби.

Також у деяких іграх доцільне використання критичних влучань по слабким місцям ворога. Найчастіше це описується попаданням в голову ворога або в зону котра так чи інакше виділяється на фоні всього іншого тіла ворога. З наявністю такої механіки доцільно зменшити кількість шкоди по звичайним частинам тіла і збільшити шкоду в декілька разів при влученні по слабким місцям ворога.

В іграх де використовується зброя також можна часто зустріти “червоні бочки” від котрих майже кожен гравець знає що очікувати, а саме те що вона може вибухнути. Тут уже приходить на допомогу фізика. Те як зброя в багатьох іграх може зруйновувати ігрові об’єкти та навіть впливати тактично на гру можна обговорювати досить довго, але перш за все фізична віддача від зброї вводиться саме для підвищення отримуваних емоцій від використання зброї.

1.6.4 Інтерфейс та надання інформації гравцю

Так-як в багатьох Roguelike іграх часто необхідна швидка реакція то інтерфейс повинен інтуїтивно відображати поточні статуси гравця та його зброї. Як мінімум в нашому випадку необхідно просто та зрозуміло відображати здоров’я гравця, так-як це може сильно впливати на пасивність та активність геймплею гравця.

Також згадуючи модифікатори котрі гравець може отримувати впродовж гри гравець повинен мати можливість передивлятись які модифікатори були підібрані та якщо якісь модифікатори складуються то скільки однакових модифікаторів знаходиться в кожній купі. Тут на допомогу приходить класична система інвентаря. Інвентар можна зробити багатьма способами починаючи з простого відображення списку предметів в худі текстом закінчуючи вбудовуванням інвентаря прямо в світ в якому знаходиться гравець. Наприклад як інвентар розгортається в вигляді рюкзака на підлозі на фоні лісу в грі The Forest. Але для більшості ігор де фокус іде саме на ігрові механіки часто достатньо простого відображення сітки інвентаря на екрані з іконками ігрових предметів.

При високій динамічності гри та особливо при великій кількості ворогів часто просто необхідно показувати підказки гравцю про те що зараз відбувається навколо нього. Тут на допомогу приходить відображення зон ураження та траєкторії снарядів, що дозволить гравцеві чистіше читати ситуацію на полі бою та швидко реагувати на неї відповідно.

1.7 Огляд існуючих відеоігор в жанрі FPS Roguelike

1.7.1 Gunfire Reborn

Gunfire Reborn одна з найпопулярніших ігор жанру FPS Roguelike. В даному проєкті ви граєте за одного з чотирьох звіроподібних Героїв, котрі збираються дослідити підземелля для зупинення зла. Фрагмент геймплею даної гри зображений на рисунку 1.



Рисунок *1 — Gunfire Reborn*

В даній грі є декілька ключових механік, а саме система “натхнення” та таланти героїв. Геймплей побудований на створенні надпотужних синергій між предметами котрі гравець здобуває впродовж гри та навичками котрі мають кожен з чотирьох Героїв.

Відчуття прогресу та потужності в ній передається від гріндексу та “зламу” гри за допомогою комбінацій навичок та предметів гравця в поєднанні з хорошим контролем зброї

1.7.2 Roboquest

В даній грі ви - робот-охоронець, який прокидається в футуристичному світі, котрий захопили ворожі роботи і ви маєте знайти з напарником шлях до порятунку. Фрагмент геймплею зображений на рисунку 2.



Рисунок *2 — Roboquest*

Із ключових механік котрі сильно впливають на геймплей я особливо хочу виділити надзвичайно високу швидкість руху та плавний паркур. Також з наявністю даних механік гравцю необхідно постійно увертатись від швидких атак ворогів для успішного проходження гри.

В даному проєкті особливо підсилюються відчуття через адреналін та відчуття стану потоку через те що стрільба в грі моментальна та досить точна разом з досить швидким паркуром в поєднанні з швидкою зачисткою кімнат без зупинки дають гравцю незабутні відчуття в яких він одночасно бореться з ворогами та швидко проходить рівень за рівень з ризиком почати гру з початку в будь який момент.

1.7.3 Deadlink

В даній грі ви - оперативник, який дистанційно керує бойовими кібернетичними корпусами (Deadlink), щоб запобігти глобальному корпоративному тероризму. Фрагмент геймплею даної гри зображено на рисунку 3.



Рисунок *3 — Deadlink*

Із ключових механік особливо хочеться виділити мобільність, орієнтовану на бій та система відновлення здоров’я та щита за рахунок агресивних дій(для прикладу - добивання).

Відчуття від гри підсилюються за допомогою агресивності та тактичної люті в поєднанні з відчуттям винагороди за ризикований та швидкий наступальний стиль гри. До того ж стрільба є інтенсивною і вимагає постійної уваги до позиціювання на полі бою.

1.7.4 BPM: Bullets Per Minute

Даний проект надає можливість гравцю побувати в ролі валькірії, яка захищає свій дім від сил підземелля.

Усі дії, такі як стрільба, перезарядка, ухилення повинні виконуватись в такт музиці, так-як вона задає темп бою. За рахунок цієї механіки ритму в боях створюється свій унікальний ритмічний виклик, а також підвищується задоволення від точності. Стрільба дає найбільшу звукову та візуальну віддачу, коли гравець ідеально синхронізує дії з бітом, перетворюючи бій на танець. Фрагмент геймплею Bullets Per Minute зображено на рисунку 4.



Рисунок *4 — BPM: Bullets Per Minute*

1.7.5 Immortal Redneck

Ви - реднек, що потрапив у Єгипет, де муміфікувався і воскрес із магічною силою, щоб досліджувати піраміди. Фрагмент геймплею Immortal Redneck зображено на рисунку 5.

В цій грі чудово вписався швидкий, олдскульний FPS-геймплей та система талісманів богів, що дають тимчасові бонуси. Також в даній грі присутня система перманентних апгрейдів між забігами, що ще більше показує гравцю його прогрес і підвищує відчуття винагороди навіть якщо гравець зумів пройти гру уже не один раз.

В даному шутері-рогалику присутнє класичне відчуття “Boomer Shooter”. Задоволення від чистої, швидкої стрільби, а також від відчутного прогресу персонажа між смертями залишають незабутній слід в пам’яті гравця, через що він захоче ще раз повернутися в дану гру і пройти ще один забіг.



Рисунок *5 — Immortal Redneck*

1.8 Постановка завдання кваліфікаційної роботи

Постановка завдання для виконання даної кваліфікаційної роботи включає в себе такі пункти:

1. Аналіз потреб і вимог: Визначення основних потреб і вимог, котрі повинна задовольняти створена відео-гра. Це включає визначення основних механік, котрі необхідно реалізувати, для того, щоб в поєднанні визначених механік вийшла гра в жанрі FPS Roguelike, визначення апаратних вимог.
2. Створення архітектури, котра зможе поєднати потрібні механіки. Створена архітектура повинна бути гнучкою та легко підтримуватись, щоб в подальшому не виникало проблем при доданні нових механік та доданні нового контенту.
3. Створення ігрових механік. Реалізація основних ігрових механік визначених в жанрі FPS Roguelike. А саме можливість переміщення головного героя, можливість атакувати ворогів зі зброєю, котра може

посилюватись за допомогою модифікаторів зброї в інвентарі гравця, а також наявність різноманітних ворогів, котрі зможуть давати гравцю опір при спробі проходження гри. Гравець повинен отримувати нагороду за знищення ворогів та/або за успішне завершення рівнів.

4. Створення демонстраційного рівня в якому використовуються створені механіки, де можна побачити використання кожної з них, а саме можливість переміщення гравця, бойова система, інвентар модифікаторів гравця, посилення поточної зброї гравця з використанням модифікаторів в інвентарі і процедурна генерація ігрових рівнів, де гравець може поборотись з ворогами та продемонструвати можливість вбивства ворогів та можливість своєї смерті разом з можливістю отримання модифікаторів в нагороду за перемогу над ворогами.

1.9 Висновки до першого розділу

Отже було розглянуто проблему створення відеогри в жанрі FPS Roguelike. Було розглянуто проблему створення ігрових світів, розібрані різноманітні механіки для гравця, ворогів, підсилення гравця та ворогів, а також за рахунок чого можна підсилити відчуття гравця, щоб в нього була більша мотивація повернутися в гру для ще одного забігу.

2 МЕТОДИ ТА ЗАСОБИ, ВИКОРИСТАНІ ПРИ ВИКОНАННІ РОБОТИ

2.1 Огляд літературних джерел

В сучасній ігровій індустрії ігровий рушій грає важливу роль, так-як він забезпечує розробників деяким набором інструментів, котрий може сильно прискорити та спростити розробку відеоігор. Unity — один з найактуальніших ігрових рушіїв, котрі використовуються в найрізноманітніших командах починаючи від інді розробників, закінчуючи великими студіями розробки ігор.

Основною метою при підготовці дипломної роботи є огляд літературних джерел, так-як це може розширити розуміння обраної теми та розглянути ігри жанру FPS Roguelike в сучасній ігровій індустрії.

При написанні дипломної роботи велика увага приділяється збору і аналізу літературних джерел, котрі становлять фундаментальну основу дипломної роботи. При аналізі літературних джерел використовувались онлайн-бібліотеки, статті з сайтів посвячений відеоіграм та зокрема жанру Roguelike, офіційна документація Unity та академічні журнали.

При аналізі літературних джерел велика увага була зосереджена на основних темах того як зробити хорошу гру жанрі FPS Roguelike, галузі розробки відеоігор на ігровому рушії Unity, основних інструментах та використання фреймворку Unity для створення ігрових механік. Для створення різноманітних ігрових механік можуть використовуватися різні інструменти з урахуванням того, як це впливає на геймплей.

Під час аналізу літературних джерел були визначені наступні основні теми:

1. Опис жанру відеоігор FPS Roguelike, які його основні особливості, найпопулярніші представники даного жанру, історія зародження жанру.
2. Дослідження ігрового рушія Unity, його основні інструменти, історія зародження, популярні проекти створені на основі ігрового рушія Unity.

3. Використання фреймворку ігрового рушія Unity для створення ігрових механік та їх поєднання для створення геймплею гри.

Жанр відеоігор FPS Roguelike може бути досить складним в реалізації, так як даний жанр може поєднувати в собі велику кількість ігрових механік, котрі повинні взаємодіяти один-з-одним. Аналіз представників жанру допоможе краще виразити основні особливості жанру, щоб в подальшому було від чого відштовхуватись при додаванні нової механіки.

Дослідження ігрового рушія Unity та аналіз інструментів допоможе зрозуміти основні можливості даного ігрового рушія, щоб в подальшому знати які елементи є і які компоненти необхідно використовувати для виконання деяких дій та для створення ігрових механік. Також при аналізі основних інструментів ігрового рушія є можливість зрозуміти за що відповідає кожен компонент ігрового двигуна.

Використання ігрового рушія Unity має під собою необхідність використання його фреймворку для взаємодії компонентів ігрового рушія між собою. Повне розуміння того, як можуть взаємодіяти різноманітні компоненти ігрового рушія за допомогою його фреймворку допоможе пришвидшити створення нових ігрових механік та полегшити створення гнучкої архітектури, котру можна буде зручно підтримувати.

Великою проблемою під час створення інді FPS Roguelike на Unity є те, що не завжди можна знайти безкоштовні ресурси на бажану тематику для додавання їх в гру. Дану проблему розробник може нівелювати при наявності деяких навичок наприклад в створенні 3D моделей, запису та редагуванні звуків, анімацій і так далі, але не завжди у розробника є час на освоєння нових умінь для створення своїх ігрових ресурсів.

Також ще однією проблемою може стати складність розробки ігрових механік та поєднання їх в архітектурі. Дана складність більш помітна в командах з однієї людини, коли їй приходиться реалізувати велику кількість механік. Саме поєднання даних ігрових механік може стати випробуванням

для невеликих команд, так як без хорошої архітектури проєкту, даний проєкт буде складно налагоджувати та підтримувати.

На сьогоднішній день у людей досить часто змінюються інтереси до різноманітних жанрів відеоігор і часто інді-розробники та невеликі команди не вгадують з тим, чого саме хочуть геймери, і створені проєкти, скільки б часу та грошей не було в них вкладено, часто стають не поміченими, або і зовсім не доходять до свого релізу.

Але не дивлячись на всі ці проблеми, котрі можуть виникнути в процесі розробки FPS Roguelike гри на ігровому рушії Unity можуть нівелюватися можливістю виникнення таких подій:

1. Можливість створити чудовий продукт та стати поміченим або навіть знайти інвестора та/або видавця, котрий може взяти на себе деякі витрати на рекламу продукту та на підтримку проєкту.
2. Якщо проєкт цього заслуговує, то можна активувати збір на Kick-Starter і якщо все буде добре, то гра може отримати деяку кількість фанатів, котрі можуть підтримати проєкт фінансово, чи особисто. Наприклад — допомогти в розробці тих чи інших елементів гри.
3. Якщо гра так сказати вгадає те, що може “вистрелити” при публікуванні гри, то вона може швидко здобути свою аудиторію та почати приносити деякі фінанси.

2.2 Ігровий рушій Unity

Unity — це ігровий рушій, розроблений і підтримуваний компанією Unity Technologies. Він використовується для створення ігор, мобільних додатків, веб-додатків, створення анімацій та інших інтерактивних додатків.

Unity має широкий набір інструментів для розробки ігор. Наприклад графічний рушій, котрий чудово працює як з 2D графікою, так і з 3D графікою, фізичний рушій, систему штучного інтелекту, систему звуку, систему створення анімацій та багато інших інструментів, котрі мають низький поріг

входження та об’ємну документацію, що дозволяє розробникам швидко приступити до створення різноманітних ігор з різними механіками та захоплюючим геймплеєм. А завдяки тому, що Unity вміє працювати як з двовимірною графікою, так і з тривимірною графікою, то мало що зможе зупинити розробника на шляху створення стилізованих або реалістичних ігор.

В ігровому рушії Unity використовується мова програмування C#, а також Visual Scripting. Мова Visual Scripting дозволяє розробникам, котрі мало знайомі з C# створювати гру без написання коду.

При розробці ігор на Unity розробник взаємодіє з графічним інтерфейсом ігрового рушія, котрий дозволяє керувати об’єктами на сцені, створювати компоненти, редагувати об’єкти і так далі. На Рисунку 6 зображено графічний інтерфейс ігрового рушія Unity.

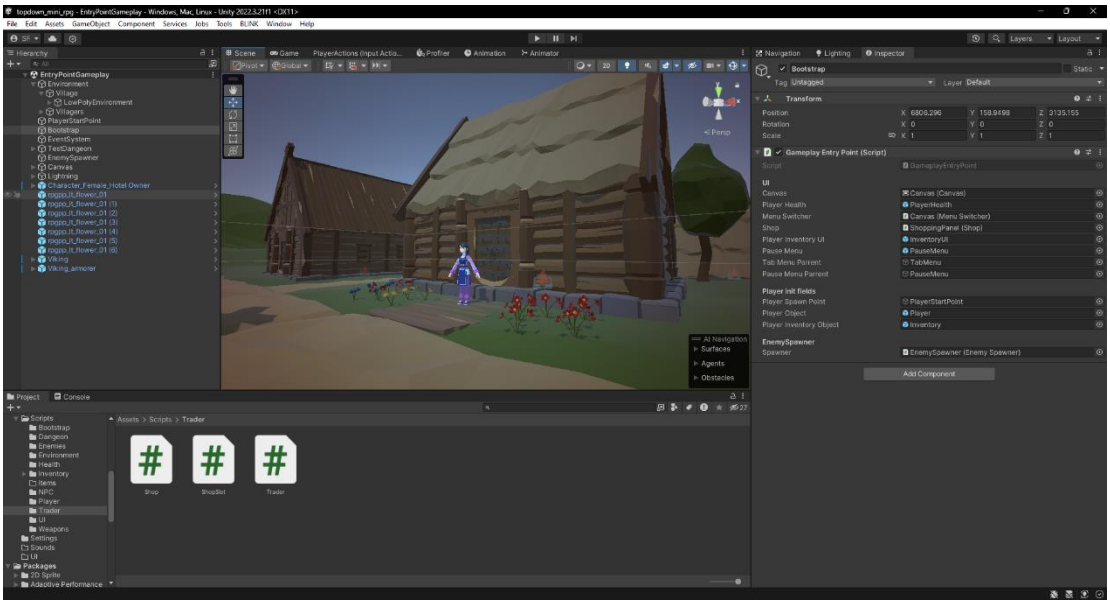


Рисунок *6 — Графічний інтерфейс ігрового рушія Unity*

Даний ігровий рушій має багату систему для знаходження проблем гри, а саме Profiler, котрий дозволяє почати захоплення ігрових кадрів та проаналізувати їх, щоб знайти можливі місця для оптимізації гри. Наприклад виконання деякого скрипту може займати багато часу і це можна побачити в профайлері, або деяке джерело світла може мати незвичайні параметри та

потребувати великого часу обчислення. Загалом це дуже гнучкий інструмент, котрий вміє працювати також з Android та iOS пристроями, для котрих можна збирати збірки для розробки та вмикати автоматичне під'єднання до профайлеру, котре дозволить знайти причини лагів в додатку.

Також говорячи про інструменти відлагоджування необхідно згадати про вбудовану можливість використання LogCat для андроїд пристроїв. LogCat використовується для отримання доступу до логів додатку. Даний інструмент допоможе в знаходженні помилок при виконанні додатку на цільовому пристрої.

Також Unity має вбудовану систему редагування шейдерів Shader Graph, котра дозволяє створювати чудові ігрові ефекти не використовуючи стару мову шейдерів Shaderlab, а використовувати редактор у вигляді графу. На Рисунку 7 зображений інструмент Shader Graph.

Продовжуючи розповідь про широкі вбудовані можливості необхідно також згадати про наявність в останній версії даного ігрового рушія вбудовано асистента зі штучним інтелектом – Muse. Даний вбудований штучний інтелект має широкий спектр можливостей, котрий може допомогти розробникам пришвидшити створення прототипу гри різноманітними способами. Наприклад Muse AI може генерувати спрайти та текстури для використання в грі, додання нових анімацій для людиноподібних персонажів, генерування заготовок коду за допомогою чату з ШІ та навіть можливість налаштування персонажів за допомогою Muse Behavior. Користувач має можливість описати потрібну поведінку і Unity Muse створить дерева поведінки в ігровому рушії Unity [12]. Також при генерації спрайтів розробник має можливість редагувати їх з допомогою генеративного заповнення, коли розробник виділяє зону на спрайті та описує, що в ній повинно з'явитися.

Muse AI не єдиний інструмент зі штучним інтелектом, котрий можна використовувати в Unity. Unity також має доступ до багатьох інших інструментів, котрі створені його спільнотою. Одні з найпопулярніших це:

- Convai – дозволяє швидко зробити ігрових персонажів живими надавши їм можливість відповідати на запитання гравця, відтворюючи анімацію лиця дуже близьку до природної. Також з даним інструментом створені ігрові персонажі можуть відтворювати інші анімації тіла
- Inworld – має схожий функціонал як у попереднього як написано на сайті [13] — даний інструмент використовується щоб «надавати гравцям новаторські ігрові механіки, динамічних неігрових персонажів та світи які розвиваються з кожною подією».
- Layer – використовується для генерування різноманітних спрайтів в різних стилях, таких як: реалізм, піксель-арт, мультяшний, комічний та в багатьох інших. Даний інструмент, як говориться на головній сторінці даного інструменту [14] є завіреним компанією Unity.
- Polyhive – інструмент, який використовується для високоякісного текстурування об’ємних моделей за допомогою штучного інтелекту [15].
- LMNT – використовується для генерування реалістичної людської мови з можливістю додавати емоції.

Існує ще багато інструментів, які часто використовуються з ігровим рушієм Unity і якщо повернутись до інструментів, які розробляються та тестуються самою Unity, то необхідно згадати про Unity Sentis [16]. Даний інструмент, як говориться на його головній сторінці [16] – «Сентіс швидким темпом вносить використання інноваційних моделей штучного інтелекту в розробку ігор. Sentis надає можливість імпортувати свої моделі штучного інтелекту та використовувати їх прямо в Unity». Також необхідно сказати про те, що Сентіс запускається та використовується на комп’ютерах клієнтів, а не в хмарі, «усуває складну хмарну інфраструктуру, затримки мережі і періодичні витрати на підтримку системи» [16]. Можна привести декілька прикладів використання Sentis AI в своїх іграх, а саме використання моделей штучного інтелекту для отримання поточного знаку користувача, який він показує в камері або можна використовувати дані моделі для перетворення природної мови в текст, створення інтерактивних ігрових персонажів, читати рукописи,

ідентифікувати об’єкти через камеру пристроя клієнта або через ігрову камеру або оцінки глибини світу при використанні з AR, щоб правильно розміщати та відображати віртуальні об’єкти. Всі ці приклади описані тут [16].

Якщо ще більше зануритися в те, які можливості представляє ігровий рушій Unity, то необхідно згадати про платформу від розробників Unity Learn, котра допоможе починаючим розробникам швидше поринути в світ розробки своїх ігор. Дана платформа має можливість користувачам оглянути майже всі інструменти в юніті читаючи короткий опис з прикладами, а також маючи можливість продивитись відео з використанням даних інструментів для глибшого розуміння їх можливостей та призначення.

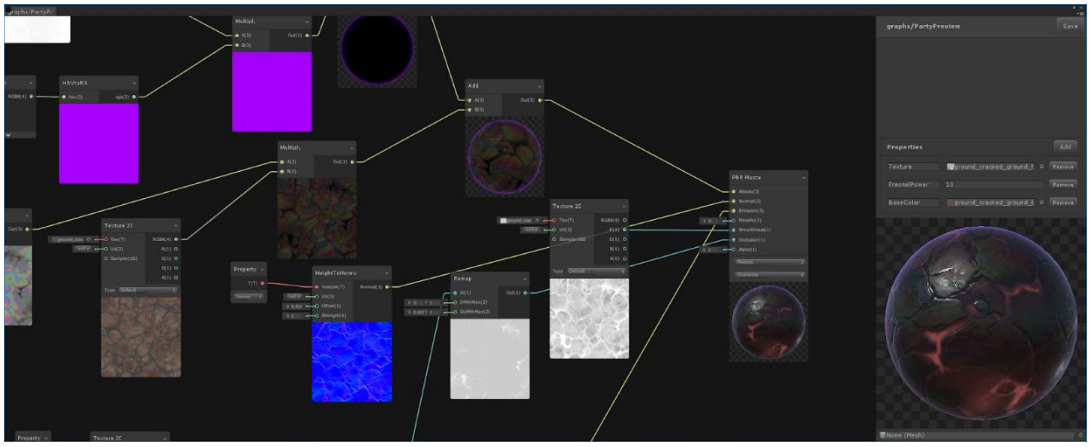


Рисунок 7 — Інтерфейс Shader Graph

Ігровий рушій Unity добре себе зарекомендував у створенні різноманітних шедеврів ігрової індустрії, таких як Cities: Skylines, Beat Saber, Rust, Genshin Impact, Phasmophobia, Escape from Tarkov, Risk Of Rain, Risk Of Rain 2 та сотні інших. Ігровий рушій має одну з найбільших спільнот серед ігрових рушіїв, що породжує велику кількість інформації, відповідей на запитання, різноманітних плагінів для ігрового рушія, графічних та звукових ресурсів.

Unity має вбудовану систему мультиплатформенності, що дозволяє зібрати білд своєї гри майже під будь-яку популярну платформу, а саме ПК, Android, PS, Xbox, WebGL, Nintendo Switch, TV OS, VisionOS, iOS та лінукс.

Це дає розробникам створювати різноманітні ігри під ті платформи на яких їм буде найзручніше. Також ігровий рушій Unity є найкращим вибором, якщо говорити про мобільні ігри та ігри для не потужних комп'ютерів, але водночас дозволяє створювати ігри з реалістичною графікою.

Те, що Unity став найпопулярнішим ігровим рушієм на мобільних пристроях породило те, що юніті має вбудовану систему аналітики, реклами для мобільних пристроїв. Також даний ігровий рушій має різноманітні дебагери, котрі дозволяють на ранніх стадіях розробки виявляти та виправляти помилки.

Unity має вбудовані функції для роботи з тривимірною та двовимірною графікою, звуком, шейдерами, матеріалами, штучним інтелектом, анімацією та багатьма іншими аспектами гри.

Описуваний ігровий рушій має підтримку розробки AR та VR ігор, що робить Unity ще більш універсальним інструментом для розробки фантастичних ігор. Прикладом інструменту для реалізації ігор з використанням AR є поєднання ARCore разом з Geospatial Creator, котрий надає можливість використовувати реальний світ як ігровий використовуючи дані про геолокацію користувача та про направлення компасу його пристрою. Прикладом використання даного поєднання є уже стара гра Pokémon Go в якій гравцю необхідно подорожувати реальним світом, щоб спробувати себе в колекціонуванні покемонів.

Сьогодні Unity використовується широким колом користувачів по всьому світу, починаючи з людей, для яких створення ігор на даному ігровому рушії просто хобі або вони займаються цим для навчання, завершуючи користувачами з комерційних організацій, котрі варіюються від так званих студій в "гаражі" або інді розробників до великих інтернаціональних студій, котрі використовують Unity для створення величезної кількості ігор.

Також на сьогоднішній день в Unity підтримуються три рендер пайплайни [17], а саме Built-In Render Pipeline, Universal Render Pipeline та

High Definition Render Pipeline. Пройдемося по основним особливостям кожного з них:

1. Built-In Render Pipeline — це найперший рендер пайплайн, котрий зазвичай використовується для слабких пристроїв, в якому немає можливості використовувати найсучасніші можливості юніті.

2. Universal Render Pipeline — це другий рендер пайплайн в юніті. Даний рендер пайплайн є золотою серединою між Built-In Render Pipeline та High Definition Render Pipeline, котрий дозволяє використовувати більшість нових інструментів, нові графічні елементи і так далі, але разом із цим дозволяє створювати оптимізовані ігрові світи для для великої різноманітності платформ, починаючи з мобільних пристроїв, Wii, Nintendo Switch аж до потужних консолей та ПК.

3. High Definition Render Pipeline — це найпотужніший рендер пайплайн, котрий підтримує всі найновіші технології в юніті. Наприклад освітлення за допомогою трасування променів, відображення за домопогою того ж трасування променів, разноманітні Post-ефекти, котрі в поєднанні з вбудованими кінематографічними ефектами, трасуванням шляху та іншими неназваними технологіями можуть дати досить реалістичне зображення в вашій грі. Також даний рендер пайплайн разом з універсальним рендер пайплайном використовується в архітектурних візуалізаціях, симуляціях, фільмах та на живих виступах.

Включаючи все описане раніше Unity є потужним інструментом для розробки ігор з великою кількістю вбудованих інструментів, що дозволяє розробникам не витрачати купу часу на створення своїх інструментів, а зосередитися на створенні якісних, різноманітних ігор для багатьох платформ.

2.3 Visual Studio

Visual Studio — це середовище програмування, підтримуване компанією Microsoft, котре дозволяє писати програми на різних мовах програмування та має багато влаштованих інструментів, котрі пришвидшують написання коду, можуть підвищити його якість та багато інших речей. З основних влаштованих інструментів це автозавершення коду, виконання збірки проєкту, інструменти для відлагодження, тестування та інші. Visual Studio добре себе зарекомендувала в відлагодженні ігрових механік в Unity.

Також дане середовище програмування дозволяє пришвидшити створення графічних інтерфейсів на різних технологіях, таких, як MAUI, WPF, Xamarin та інших.

2.4 Висновки до розділу 2

Оглянуті основні методи та інструменти, які будуть використані при створенні відеогри в жанрі FPS Roguelike, такі як Unity, її можливості, інструменти, а також Visual Studio.

3 ОПИС СТВОРЕННЯ ВІДЕОГРИ ЖАНРУ FPS ROGUELIKE

3.1 Опис предметної області роботи та її мета

Предметна область цієї роботи включає в себе розробку відеогри в жанрі FPS Roguelike, та основний фокус в даній роботі присвячений саме створенню ігрових механік, котрі можуть ефективно комунікувати одна-з-одною для створення приємного геймплею за допомогою навичок використання ігрового рушія Unity та за допомогою навичок програмування з використанням мови програмування C#.

При розробці відеогри на ігровому рушії Unity дуже важливо розбити систему на підсистеми, щоб в подальшому було зручніше підтримувати та розширювати систему без внесення великих змін до системи та без великої кількості дубльованого коду.

Дипломна робота зосереджується на створенні ігрових механік притаманних жанру FPS Roguelike з використанням ігрового рушія Unity. Використання ігрового рушія Unity виправдане відносно низьким порогом входження до створення ігор з якісною та оптимізованою графікою, котра зможе добре працювати навіть на слабких пристроях з використанням джерел світла, тіней, текстур, шейдерів, скайбоксів. Також даний ігровий рушій має вбудовані можливості для додання більш складних ефектів, такий як “об’ємний туман”, котрий може привнести в гру більше глибини, що сприяє кращому зануренню гравця в ігровий процес. Також описуваний ігровий рушій використовує мову програмування C#, котра має значно менший поріг входження для написання ефективного коду в порівнянні з використанням C++ в ігровому рушії Unreal Engine.

В даній роботі також були розглянуті основні архітектурні рішення при розробці відеоігор на основі ігрового рушія Unity.

Виконання даної роботи сприяє покращенню навичок розробки відеоігор загалом та підвищенню навичок при розробці відеоігор з використанням ігрового рушія Unity.

Мета кваліфікаційної роботи - це створення прототипу відео-гри жанру FPS Roguelike з використанням ігрового рушія Unity, котрий в майбутньому можна буде розширяти та наповнювати новим контентом. Наприклад новими класами гравця, предметами, ворогами і так далі.

3.2 Опис систем, котрі необхідно реалізувати

3.2.1 Система модифікаторів гравця

Необхідно реалізувати систему модифікаторів для сутностей в ігровому світі, котра дозволить використовувати модифікатори для різних подій і на ворогах і на гравці та його зброї. Повинно бути декілька модифікаторів котрі б давали наглядно зрозуміти що вони роблять. Для прикладу необхідно зробити модифікатор заморозки, вампіризму(регенерування здоров’я за частиною від нанесеної шкоди противнику) та модифікатор на пришвидшення перезарядки зброї.

Дана система надає можливість підсилювати можливості гравця різноманітними засобами. Модифікатори повинні відповідати таким вимогам:

- Можливість додавати та видаляти модифікатори з сутностей.
- Можливість активувати модифікатори на різноманітні ігрові події(такі як атака гравця, стрибок, перезарядка, влучення в ворога).
- Деякі модифікатори повинні працювати на зброї, а деякі лише з компонентами переміщення ігрового об’єкта.
- Деякі модифікатори повинні мати можливість посилюватись від кількості наявних однакових модифікаторів з такою властивістю.

3.2.2 Система переміщення гравця

Гра повинна надавати можливість гравцю переміщуватись по ігровому світу з використанням клавіатури і миші спостерігаючи за світом від першої особи. Дана система повинна забезпечувати такі можливості, як:

- Керування камерою(головою) гравця за допомогою миші.
- Можливість переміщатися в режимі бігу.
- Можливість робити стрибки, до дозволить гравцю наприклад уникнути отримання шкоди від ворогів або просто переміщатись швидше по ігровому світу.

Створення данної механіки надає гравцю можливість виживати та переміщуватись в ігровому світі уникаючи атак ворогів да двигатись в сторону своїх цілей.

3.2.3 Система ворогів

Вороги як і система модифікаторів грають ключові ролі в жанрі Roguelike, так як вони одночасно показують гравцю його прогрес та самі посилюються впродовж гри, що робить гру цікавішою. В даному випадку система ворогів має відповідати таким вимогам:

- Можливість налаштовувати різні пресети ворогів да допомогою конфігураційних файлів створених за допомогою Scriptable Object`s в юніті.
- Необхідно реалізувати декілька класів ворогів:
  - Melee(ворог ближнього бою) - клас ворога, котрий для виконання своїх атак повинен підійти до гравця як можна ближче.
  - Ranged(ворог середнього бою) - клас ворога, котрий для виконання своїх атак повинен триматись на середній дистанції від гравця і намагається попасти по гравцю снарядами.

- Sniper(снайпер) - клас ворога, котрий атакує з великих відстаней. Для виконання своєї атаки він робить заряджаючу атаку та робить досить швидкий і потужний постріл снарядом.

- Необхідно реалізувати хоча-б один класс боса котрий зустрічається в кінці рівня. Даний бос повинен використовувати комбінації механік різних ворогів або виконувати кращу варіацію атак одного з них.

Також дана система повинна включати в себе фабрику для спауну ворогів на ігровій сцені. Дана система повинна тримати в собі набір конфігурацій ворогів. При спробі спавну ворога необхідно передавати те який ти клас ворога хочеш створити та де саме(позицію).

3.2.4 Система кімнат

Дана система включає в себе декілька невеликих підсистем котрі об’єднуються для створення системи кімнати. Вона включає в себе систему спавна ворогів в кімнаті, генерацію тригерів для активації кімнати, закриття виходів з кімнати та відкриття виходів з кімнати після перемоги всіх запланованих ворогів.

3.2.5 Система генерації рівнів

Дана система включає в себе систему генерації та розміщення кімнат на ігровому рівні на ігровій сітці, а також систему генерації коридорів між розміщеними кімнатами.

В основі даної системи стоїть кубічна координатна сітка з клітинами. Кожна клітина може містити в собі один блок розміром 1x1x1 юніти. Дана сітка дозволить зручно перевіряти чи пересікаються кімнати між собою при спробах розміщення їх на координатній сітці, а також дозволить без особливих складностей перевіряти наявність сусідів у клітинки при генерації проходів між кімнатами.

Підсистема для генерації кімнат на ігровому рівні включає в себе декілька форм кімнат. Як мінімум було визначено круглу кімнату, квадратну кімнату та кімнату Г-виду. Це дозволить додати ігровому рівні більше варіативності при генерації ігрових рівнів.

Підсистема для генерації проходів включає в себе використання алгоритму A*(A-стар) для генерації проходів і вона повинна враховувати те, що проходи повинні будуватись від проходу до проходу і ще згенеровані проходи не повинні пересікатись з розміщеними кімнатами. Також при генерації проходів повинна бути можливість налаштовувати висоту, ширину проходів, а також можливість на різних рівнях гри заміняти ассети при генерації для додавання варіативності зовнішньому вигляду гри.

Також при генерації рівнів необхідно використовувати систему генерації випадкових чисел для створення унікальних рівнів і можливість точного повторення рівня якщо зерно генерації однакове.

3.2.6 Система нанесення шкоди об’єктам

Уніфікована система нанесення шкоди об’єктам також є необхідною, так-як вона дозволить наносити шкоду будь яким об’єктам використовуючи один інтерфейс, що значно спростить наприклад систему руйнування пропів на сцені, одночасно використовуючи даний інтерфейс для нанесення шкоди ворогам та гравцю, а також наносячи шкоду об’єктам не зброєю, а наприклад за допомогою ефектів, котрі також можна накладати на ігрові об’єкти

3.3 Вимоги до апаратного забезпечення системи

Створена гра повинна запускатись та бути іграбельною навіть на слабких пристроях, так-як це дозволяє охопити більше коло аудиторії та додати шансів до успішного запуску гри.

До мінімальних вимог даної гри я вношу досить старий ноутбук з такими характеристиками:

- ОЗУ - 8GB DDR3/DDR4/DDR5
- Процессор - Intel Core i3 8130 u
- Відеокарта - Intel UHD 620 або інша з підтримкою DirectX 12
- Сховище - ~1GB SSD

Така потреба в наявності такої кількості ОЗУ на персональному комп'ютері зумовлена тим, що система генерації рівня потребує досить багато пам'яті для процедурної генерації рівнів на воксельному рівні. ПК з даними характеристиками зможе надати грабельну кількість кадрів для проходження гри без великих проблем.

3.4 Вимоги до програмного забезпечення системи

Мінімальні вимоги до програмного забезпечення системи включають в себе такі пункти:

- ОС — Windows 10 або новіше
- Останні оновлення драйверів відеокарти

3.5 Вимоги до користувачів відеогри

Головною метою цього аналізу було з'ясування потреб та очікувань цільової аудиторії, а також визначення вимог, за яких гравці могли б комфортно грати в розроблену відеогру.

З'ясувалось, що головними користувачами створеного проєкту будуть гравці котрі в пріоритеті ставлять зручне керування в грі, приємну графічну складову та динамічний геймплей. Також дана гра повинна надавати випробування навіть для гравців з великим ігровим досвідом в іграх подібного жанру.

Було визначено таку основну роль користувачів розробленої відеогри:

- Гравець.

- Функції: Проходження гри, взаємодія з елементами керування в грі, з клавіатурою та мишкою, налаштування параметрів гри.
- Можливості системи: Запуск відеогри, налаштування параметрів гучності, якості графіки, керування грою.
- Вимоги: Мінімальні або відсутність ігрових навичок на комп’ютері, розуміння взаємодії з типічними ігровими пристроями, знання типічних основ геймплею.

- Тестувальник.
  - Функції: Проходження гри, знаходження багів при типічній поведінці гравця, пошук багів при нетипічній поведінці гравця, передача знайдених багів та помилок до розробників.
  - Можливості системи: Запуск відеогри, керування грою, налаштуваннями гри.
  - Вимоги: Середні ігрові навички, креативне та критичне мислення.
- Розробник.
  - Функції: Створення гри, додавання контенту, виправлення багів, додавання нових механік, квестів та все інше, що можна рахувати за підтримку та оновлення гри.
  - Можливості системи: Повний контроль над проектом відеогри, її контентом, механіками, архітектурою.
  - Вимоги: Впевнене користування ігровим рушієм Unity, середні навички мови програмування C#, розуміння принципів роботи з компонентами в Unity.

3.6 Висновки до розділу 3

Отже в даному розділі було розглянуто та описано системи необхідні для створення в грі, а саме такі системи, як:

- Система модифікаторів
- Система переміщення гравця

- Система ворогів
- Система кімнат
- Система генерації рівнів
- Система нанесення шкоди об’єктам

Також було розглянуто апаратні та системні вимоги для успішного запуску для проходження гри, а також було визначено вимоги до користувачів даної гри FPS Roguelike.

4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ ВИКОНАНОЇ РОБОТИ

4.1 Реалізація відеогри в жанрі FPS Roguelike на ігровому рушії Unity

4.1.1 Система переміщення гравця

Розробка якісного FPS контроллера в юніті може бути досить цікавим завданням, так-як тут можна застосувати багато різних підходів. По-перше ігрові контролери для переміщення персонажів можна зробити двома основними способами:

1. За допомогою вбудованого компонента CharacterController в ігровому рушії Unity. Даний компонент уже має багато вбудованих систем, таких як система колізій з об’єктами, система висот кроків, переміщення контроллера на координату в світі і так далі
2. За допомогою вбудованого Rigidbody компонента. Даний підхід надає найбільшу гнучкість при створенні контролерів персонажів в юніті, але й вимагає трохи більшого розуміння того як робити хороші ігрові контролери в відеоіграх, щоб досягти плавного переміщення персонажу, а також зробити нормальне переміщення по сходам.

В моєму випадку я обрав створення FPS контроллера через Rigidbody компонент, так-як я хотів додати йому фізично вірне відкидання від атак ворогів. Це б додало трохи веселощів до геймплею, що є досить важливо.

Для обробки зіткнень з навколишнім світом героя я обрав Capsule Collider, після цього було додано компонент Rigidbody на тіло гравця, а після цього було виставлено деякі параметри на компоненті, а саме:

- вимкнено обертання фізичного тіла гравця по осям X та Z, так-як якщо їх залишити незамороженими, то при доданні фізичних сил до тіла воно може просто почати котитись як звичайний капсуль
- налаштовано масу гравця, параметр тріння його з іншими колайдерами на сцені, а також було перевірено щоб параметр IsKinematic був

вимкнений. Даний параметр робить так, що даний фізичний об’єкт може впливати на інші фізичні об’єкти, але сам він не буде отримувати ніякої віддачі від цього зіткнення.

Далі було створено скрипт зручної підписки на події вводу з системи Input System. Даний скрипт дає можливість звернутися до статичного об’єкта для підписки на події вводу гравця. Уривок даного скрипта можна побачити в Лістингу 1.

Лістинг 1 — Клас InputManager для глобального менеджмента активних інпутів гравця

```
public class InputManager : MonoBehaviour
{
    public static InputManager Instance { get; private set; }

    [SerializeField]
    private InputActionAsset actionsAsset;

    private InputActionMap actionMap;
    private InputAction moveAction;
    private InputAction lookAction;
    private InputAction reloadAction;

    public enum InputState { Started, Performed, Canceled }

    public UnityEvent<Vector2> OnMove = new();
    public UnityEvent<Vector2> OnLook = new();
    public UnityEvent<InputState> OnJump = new();
```

```
    private bool isJumpHolding = false;


 private void OnEnable()
    {
        actionMap?.Enable();
        if (moveAction != null) moveAction.performed += InvokeMove;
        if (moveAction != null) moveAction.started += InvokeMove;
        if (moveAction != null) moveAction.canceled += InvokeMove;
}
private void Update()
    {
        if (isJumpHolding)
            InvokeJump(InputState.Performed);
    }


    private void InvokeLook(InputAction.CallbackContext context)
    {
        Vector2 lookDelta = context.ReadValue<Vector2>();
        OnLook?.Invoke(lookDelta);
    }
  private void OnDisable()
    {
        if (moveAction != null) moveAction.performed -= InvokeMove;
        if (lookAction != null) lookAction.performed -= InvokeLook;
        if (lookAction != null) lookAction.canceled -= InvokeLook;
}
```

```
//та методи які викликаються при події вводу.
```

Як видно з представленного Лістингу 1 спочатку створюється статичне поле яке буде зберігати в собі глобальне посилання на єдиний InputManager, потім створюються поля для підписки та зберігання подій. Для прикладу там є посилання на події переміщення, стрибка, атаки, спрінта та інших подій вводу гравця. Далі в методі OnEnable котрий викликається при ввімкненні ігрового об’єкта на сцені відбувається підписка на події з InputActionAsset. В методі OnDisable(котрий викликається при вимкненні даного ігрового об’єкта) відбувається протилежна дія - відписування від подій мапи вводу гравця. Також в даному методі було створено окремий enum для станів вводу. Для прикладу в мене воно використовується для показу того, що пробіл затиснений в даний момент і стрибки можна повторювати при приземленні гравця поки він не відпустить пробіл.

Після цього було створено скрипт FPS Controller, котрий уже підписується через Instance на InputManager та використовує описані події для керування тілом гравця. На Лістингу 2 можемо побачити частину реалізації класу FPS Controller:

Лістинг 2 — FPSController, поля класу, підписка на події вводу, методи оновлення рухів гравця.

```
[RequireComponent(typeof(Rigidbody), typeof(CapsuleCollider))]
public class FPSController : MonoBehaviour
{
    public bool IsGrounded => isGrounded;

    [Header("Movement Settings")]
    [SerializeField] private float walkSpeed = 5f;
```

```
[SerializeField] private float sprintSpeed = 10f;

[SerializeField] private float acceleration = 10f;

[SerializeField] private float airControl = 0.5f;

[SerializeField] private float jumpForce = 5f;

[SerializeField] private float crouchSpeed = 2.5f;

[SerializeField] private float crouchHeight = 1f;

[SerializeField] private float standingHeight = 2f;

[SerializeField] private float groundDrag = 1f;

[SerializeField] private float jumpCooldown = 1f;


[Header("Slope Handling")]

[SerializeField] private float maxSlopeAngle = 60f;

private RaycastHit slopeHit;


[Header("Components")]

private Rigidbody rb;

private CapsuleCollider col;


[Header("Move")]

private float moveSpeed;

private Vector2 moveInput;

private Vector3 moveDirection;


[Header("Booleans")]

private bool isGrounded;

private bool isSprinting;

private bool isCrouching;
```

```
    private bool readyToJump = true;


    private float slowMultiplier = 1f;
    private float slowDuration = 0f;


    private bool isFrozen = false;
    private float freezeDuration = 0f;



    private void Awake()
    {
        rb = GetComponent<Rigidbody>();
        col = GetComponent<CapsuleCollider>();
        rb.freezeRotation = true;
        rb.interpolation = RigidbodyInterpolation.Interpolate;
        rb.collisionDetectionMode                                    =
CollisionDetectionMode.ContinuousDynamic;
        rb.isKinematic = true;
        this.enabled = false;
    }


    private void Start()
    {
        Instance.OnMove.AddListener(OnMove);
        Instance.OnJump.AddListener(OnJump);
        Instance.OnSprint.AddListener(OnSprint);
        Instance.OnCrouch.AddListener(OnCrouch);
    }
```

```
private void Update()
{
    CheckGround();
    ControlMaxSpeed();
    Drag();
    UpdateStatusEffects();
}

private void FixedUpdate()
{
    if (!isFrozen)
        Move();
}
```

Як видно з представленої частини коду даний скрипт залежить від компонентів Rigidbody(фізичне тіло) та Capsule Collider(фізичний колайдер для обробки зіткнень тіла гравця зі світом. В методі Awake ініціюються деякі поля класу, а в методі Start, котрий викликається перед відмальовуванням першого кадру в грі проходить підписка на події з InputManager.Instance.

Далі в методі Update котрий викликається кожен кадр відбувається оновлення даних про те як і куди повинен двигатись гравець, а саме перевіряється чи він намагається йти по схилу чи ні, чи він у повітрі, а також відбувається оновлення статусів ефектів, котрі накладуються на скрипт переміщення гравця(наприклад заморозка та сповільнення). А в самому методі FixedUpdate котрий викликається кожен фізичний такт, а саме на стандартних налаштуваннях Unity 50 разів в секунду відбувається саме додавання фізичних сил до тіла гравця для його переміщення.

Також для реалізації керування FPS необхідно створити контроллер для керування камери від першої особи. В нашому випадку через те що ми обрали використання Rigidbody в ролі основного компонента через який відбувається керування переміщенням гравця то довелось керувати камерою як зовсім окремим компонентом, так-як через фізичне оновлення всього 50 разів на секунду рухи персонажу можуть виглядати не плавно і можна побачити як камера прям телепортується невеликими кроками, що виглядає в грі досить жахливо. Для виправлення цієї ситуації камера залишилась на окремому ігровому об’єкті і на неї був доданий скрипт Camera Follow, котрий фактично кожен кадр зі згладжуванням рухів телепортує камеру на позицію голови. Також для самого керування головою гравця було розроблено скрипт FPS Camera Controller, котрий можна побачити на Лістингу 3:

Лістинг 3 — FPS Camera Controller

```
public class FPSCameraController : MonoBehaviour
{
    [SerializeField] private Transform player;
    [Header("Camera Settings")]
    public float sensitivity = 2f;
    public float verticalClamp = 85f;

    private Vector2 lookInput;
    private float verticalRotation = 0f;
    private float horizontalRotation = 0f;

    private void Awake()
    {
        Cursor.lockState = CursorLockMode.Locked;
```

```
        Cursor.visible = false;
    }

    private void Start()
    {
        InputManager.Instance.OnLook.AddListener(OnLook);
    }

    private void Update()
    {
        RotateCamera();
    }

    private void RotateCamera()
    {
        float mouseX = lookInput.x * sensitivity;
        float mouseY = lookInput.y * sensitivity;

        verticalRotation -= mouseY;
        verticalRotation = Mathf.Clamp(verticalRotation, -
verticalClamp, verticalClamp);

        horizontalRotation += mouseX;

        transform.localRotation = Quaternion.Euler(verticalRotation,
horizontalRotation, 0f);

        player.rotation = Quaternion.Euler(0f, horizontalRotation, 0f);
```

```
    }


    public void OnLook(Vector2 context)
    {
        lookInput = context;
    }
}
```

Як видно з представленого коду даний контроллер також підписується на події доступ до яких надає InputManager, також даний контроллер має налаштування того з якою чутливістю двигається камера гравця від рухів мишкою. В залежності від вводу гравця в методі Update оновлюються повороти камери, а також повертається саме тіло гравця в сторону куди дивиться камера(повертається по осі Y - вертикальна вісь).

4.1.2 Система модифікаторів

Як описувалось раніше система модифікаторів являється ключовою в жанрі Roguelike, так-як через неї показується прогрес сили гравця в нагороду за його старання при боротьбі з супротивниками. Сама реалізація системи модифікаторів може стати досить складним завданням через велику кількість компонентів котрі по сути повинні працювати однаково при використанні для багатьох сутностей. Створення системи модифікаторів я почав зі створення класу EventEmitter, котрий дозволив мені підписуватись на події описані класами, щоб уніфікувати спосіб підписання та відписування на будь яку подію описану деяким класом. Даний основний класс показаний на Лістингу 4.

Лістинг 4 — EventEmitter

```
public class EventEmitter : MonoBehaviour, IEventEmitter
```

```
{
    private readonly Dictionary<Type, List<Delegate>> _listeners =
new();

    public void AddListener<TEvent>(Action<TEvent> listener)
    {
        var type = typeof(TEvent);
        if (!_listeners.ContainsKey(type))
            _listeners[type] = new List<Delegate>();

        _listeners[type].Add(listener);
    }

    public void RemoveListener<TEvent>(Action<TEvent> listener)
    {
        var type = typeof(TEvent);
        if (_listeners.TryGetValue(type, out var list))
            list.Remove(listener);
    }

    public void TriggerEvent<TEvent>(TEvent eventData)
    {
        var type = typeof(TEvent);
        if (_listeners.TryGetValue(type, out var list))
        {
            foreach (var del in list.ToArray())
                ((Action<TEvent>)del)?.Invoke(eventData);
        }
```

```
    }
}
```

Як видно з лістингу методи для тригеру івентів та підписки/відписки від івентів шаблонні і можуть використовуватися для найрізноманітніших подій, піднімаючи івенти для всіх слухачів поточного EventEmitter.

Також в пару до EventEmitter було створено різноманітні класи подій. Для прикладу в Лістингу 5 показано структуру події смерті ворога.

Лістинг 5 — EnemyDiedEvent

```
public struct EnemyDiedEvent
{
    public GameObject Enemy { get; }
    public GameObject Killer { get; }

    public EnemyDiedEvent(GameObject enemy, GameObject killer)
    {
        Enemy = enemy;
        Killer = killer;
    }
}
```

Як бачимо дана структура події надає всю необхідну інформацію про смерть ворога. Подібним чином організовані і інші модифікатори в грі. Після створення системи для зручної підписки на події уже почалось саме створення системи модифікаторів. Спочатку був створений Scriptable Object для зручного створення конфігурацій модифікаторів, а саме їх ідентифікатор, чи стакається даний модифікатор, скільки максимум стаків гравець може мати даного модифікатора, а також в даному конфігу вказується іконка

модифікатора. Все це необхідно для зручного налаштування модифікаторів. На Лістингу 6 можна побачити даний абстрактний клас Modifier:

Лістинг 6 — Modifier Scriptable Object

```
public abstract class Modifier : ScriptableObject
{
    public string ModifierId;
    public bool IsStackable;
    public int MaxStacks;
    public int CurrentStacks;
    public Sprite icon;

    public virtual void OnDamageDealt(DamageInfo damageInfo, IDamagable damage, uint stacks) { }
    public virtual void OnDamageTaken(DamageInfo damageInfo, IDamagable damage, uint stacks) { }
    public virtual void OnEvent(string eventType, object data, uint stacks) { }
}
```

Далі на основі даного класу було створено декілька модифікаторів котрі працюють з різними аспектами гри. Для прикладу розглянемо декілька з них, а саме Stackable Vampirism Modifier, Low Health Modifier, Reload Modifier, Jump Modifier та Freeze Modifier.

Лістинг 7 StackableVampirismModifier

```
[CreateAssetMenu(menuName = "Modifiers/StackableVampirism")]
public class StackableVampirismModifier : VampirismModifier
{
```

```
    public override void OnDamageDealt(DamageInfo damageInfo, IDamagable
damage, uint stacks)
    {
        if (IsDamageTargetPlayer(damageInfo))
        {
            return;
        }

        float totalLifeStealPercent = LifeStealPercent * stacks;

        float healAmount = damageInfo.Amount * totalLifeStealPercent;

damageInfo.Source.GetComponent<IDamagable>()?.Heal(healAmount);
    }
}
```

Як бачимо даний модифікатор спрацьовує при нанесенні шкоди цілі. Коли це відбувається то вираховується спочатку відсоток в залежності від кількості стаків того яка частина здоров’я від нанесеної шкоди буде використана для лікування власника модифікатора.

Лістинг 8 — LowHealthModifier

```
[CreateAssetMenu(menuName = "Modifiers/LowHealthDamageModifier")]
public class LowHealthDamageModifier : Modifier
{
    public float Threshold = 0.3f;
    public float DamageMultiplier = 2f;
```

```
        public override void OnEvent(string eventType, object data,
uint stacks)
        {
            if (eventType == "WeaponAttack")
            {
                var attackEvent = (WeaponAttackEvent)data;
                var playerHealth = Player.Instance.CurrentHealth /
Player.Instance.MaxHealth;

                if (playerHealth <= Threshold)
                {
                    attackEvent.DamageInfo.Amount *=
DamageMultiplier;
                }
            }
        }
```

Як бачимо в Лістингу 8 показано код модифікатора низького здоров'я. Даний модифікатор при нанесенні шкоди ворогам якщо поточна кількість здоров'я власника менша за визначений поріг, то даний модифікатор в залежності від кількості поточних стаків даного модифікатора збільшує нанесену шкоду по цілі.

Лістинг 9 — ReloadModifier

```
[CreateAssetMenu(menuName = "Modifiers/ReloadModifier")]
public class ReloadModifier : Modifier
{
    public float ReloadSpeedBonus = .95f;
```

```
        public override void OnEvent(string eventType, object data,
uint stacks)
        {
            base.OnEvent(eventType, data, stacks);

            if(eventType == "WeaponReload")
            {

Player.Instance.weaponHolder.GetActiveWeapon().ApplyReloadModifier(Mat
hf.Pow(ReloadSpeedBonus, stacks));
            }
        }
    }
```

Як бачимо в Лістингу 9 показано код класу Reload Modifier. Даний модифікатор працює для зброї гравця. Модифікатор під час івенту *Reload Event* збільшує швидкість перезарядки. За кожен стак перезарядка пришвидшується на 5 відсотків.

Лістинг 10 — JumpModifier

```
[CreateAssetMenu(menuName = "Modifiers/JumpAttackModifier")]
public class JumpAttackModifier : Modifier
{
    public float DamageMultiplier = 1.5f;
    public float CriticalChanceBonus = 0.2f;

    public override void OnDamageDealt(DamageInfo damageInfo,
IDamagable damage, uint stacks)
```

```
        {
            if (Player.Instance.PlayerMovement.IsJumping())
            {
                float res = DamageMultiplier * stacks;
                if (Random.Range(0, 100) < CriticalChanceBonus * 100)
                {
                    damageInfo.Amount *= DamageMultiplier;

                }
            }
        }
    }
```

Даний модифікатор збільшує кількість нанесеної шкоди по ворогу у випадку коли власник знаходиться в повітрі.

Лістинг 11 — FreezeModifier

```
[CreateAssetMenu(menuName = "Modifiers/FreezeModifier")]
public class FreezeModifier : Modifier
{
    public float Duration;
    public TemporaryEffect TemporaryEffect;

    public override void OnDamageDealt(DamageInfo damageInfo,
IDamagable damage, uint stacks)
    {
        base.OnDamageDealt(damageInfo, damage, stacks);
```

```
            var movement =
damageInfo.Target.GetComponent<EnemyMovementBase>();
            if (movement != null)
            {

damageInfo.Target.GetComponent<EnemyMovementBase>().ApplyFreeze(Durati
on * (stacks * 0.5f));
            }

            Instantiate(TemporaryEffect,
damageInfo.Target.transform.position, Quaternion.identity);
        }
    }
```

Модифікатор FreezeModifier при атакі по цілі накладує на неї ефект заморозки, а також інстанціює візуальний ефект заморозки котрий після свого відтворення сам себе знищує. Описуваний модифікатор в залежності від кількості стаків накладає ефект заморозки на переміщення цілі.

Далі після створення такої основи для існування самих модифікаторів було створено Modifier Handler, котрий прикріплюється до різноманітних об’єктів та тримає в собі список модифікаторів, а також уже має реалізації методів для підіймання модифікаторів для найчастіших подій.

Лістинг 12 — ModifierHandler

```
public class ModifierHandler : MonoBehaviour
{
    private List<ModifierInstance> _modifiers = new();
    public IEventEmitter EventEmitter;

    public void ApplyModifier(ModifierInstance modifier)
```

```
    {
        var existing = _modifiers.FirstOrDefault(m => m.Template.ModifierId == modifier.Template.ModifierId);

        if (existing != null && modifier.Template.IsStackable)
        {
            int stacks = Mathf.Clamp(existing.Template.CurrentStacks + modifier.GetCurrentStacks(), 1, existing.Template.MaxStacks);
            existing.IncreaseCurrentStack(stacks - 1);
        }
        else if(existing != null && !modifier.Template.IsStackable)
        {
            // don`t pickup mod if it is already in inventory and is not stackable
            return;
        }
        else
            _modifiers.Add(new ModifierInstance(modifier.Template));
    }

    public void RemoveModifier(string modifierId)
    {
        var existing = _modifiers.FirstOrDefault(m => m.Template.ModifierId == modifierId);

        if (existing != null && existing.Template.IsStackable && existing.Template.CurrentStacks > 0)
            existing.DecreaseCurrentStack(1);
        else
```

```
            _modifiers.Remove(existing);
    }

    public void ProcessDamageDealt(ref DamageInfo damageInfo, IDamagable
damage)
    {
        foreach (var instance in _modifiers)
        {
            instance.Template.OnDamageDealt(damageInfo,         damage,
(uint)instance.GetCurrentStacks());
        }
    }

    public void ProcessDamageTaken(ref DamageInfo damageInfo, IDamagable
damage)
    {
        foreach (var instance in _modifiers)
        {
            instance.Template.OnDamageTaken(damageInfo,         damage,
(uint)instance.GetCurrentStacks());
        }
    }

    public List<ModifierInstance> GetModifiers() => _modifiers;
}
```

Також для подальшого використання модифікаторів було створено обгортку Modifier котра використовується як ігровий об’єкт наприклад для відображення модифікаторів в інвентарі. При створенні інвентаря гравця було використано шаблон MVP(Model View Presenter) де скрипт Modifier Instance

виступає в якості моделі, Inventory System виступає в якості контролера, а Inventory UI виступає в якості Presentor`а. Нижче на Рисунку 8 можна побачити дану схему.

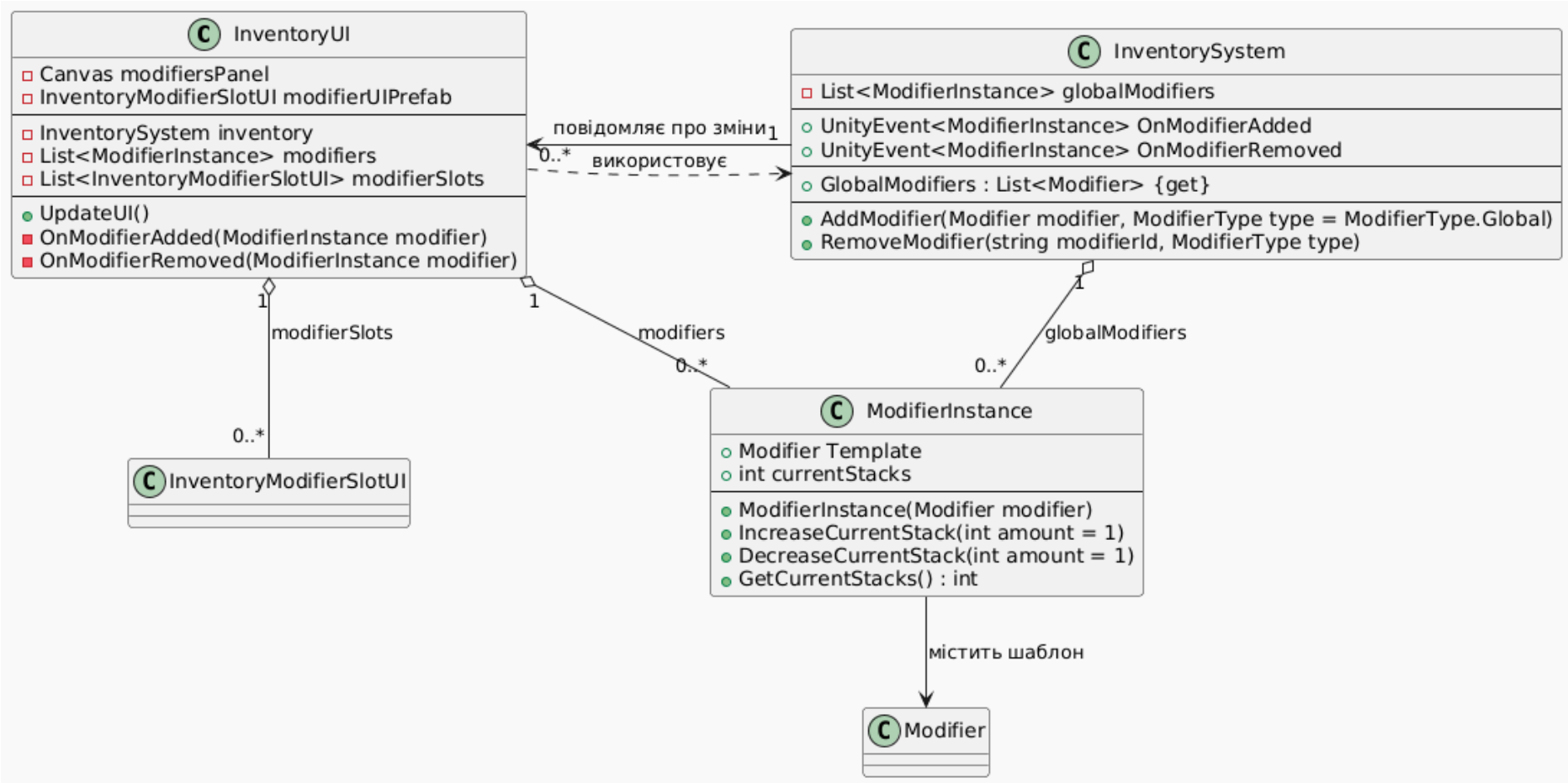


Рисунок 8 — Model View Presenter from Inventory System

Як бачимо Inventory UI підписується на події додавання та видалення модифікаторів з об’єкту Inventory System та оновлює інтерфейс інвентаря за потреби.

4.1.3 Система отримання шкоди

Так як в даному проєкті досить часто необхідно наносити шкоду різним об’єктам, то необхідно створити уніфіковану систему отримання шкоди. Для цього було створено інтерфейс як на Лістингу 13.

Лістинг — IDamagable

```
public interface IDamagable
{
    void TakeDamage(DamageInfo damageInfo);
    void Heal(float amount);
}
```

Як бачимо при наслідуванні від даного інтерфейсу наслідники зобов’язуються реалізувати методи TakeDamage та Heal. Це необхідно було зробити щоб кожна сутність могла обробити отримувану шкоду своїми модифікаторами. Щоб наприклад якщо є модифікатор зміни шкоди якщо гравець знаходиться в повітрі або коли він перезаряджається, то щоб він міг використати дані модифікатори перед тим як отримувати шкоду і компенсувати її. Через даний прошарок роботи з модифікаторами також було створено клас Damage Receiver, котрий в собі має методи для отримання шкоди та для лікування, а також підіймає івенти для EventEmitter поточної сутності при отриманні шкоди або смерті. Нижче в Лістингу 14 можна побачити реалізацію даного класу:

Лістинг 14 — DamageDealer

```
public class DamageReceiver : MonoBehaviour
{
```

```
    private EventEmitter EventEmitter;

    [Header("Damage Resistance Profile")]
    [SerializeField] private DamageResistanceProfile resistanceProfile;

    public float MaxHealth { get; private set; } = 100f;
    public float CurrentHealth { get; private set; }

    private Dictionary<DamageType, float> damageMultipliers;

    private void Start()
    {
        CurrentHealth = MaxHealth;

        if (resistanceProfile != null)
        {
            damageMultipliers                                    =
resistanceProfile.GetModifiersDictionary();
        }
        else
        {
            damageMultipliers = new Dictionary<DamageType, float>();
        }
    }

    public void SetupEvents(EventEmitter emitter)
    {
        this.EventEmitter = emitter;
```

```
    }
```

Як бачимо через метод Setup Events можна прив’язати EventEmitter до даного об’єкта здоров’я. Також в даного Damage Dealer присутнє поле з назвою damage Multipliers, котре зберігає в собі список типів шкоди та мультиплікатори шкоди від них. Це зроблено для того, щоб деякі класи ворогів або гравців могли мати вразливості або захист від деяких типів шкоди.

4.1.4 Системи ворогів

Так-як ворогів в процесі розвитку гри може ставати набагато більше, то й системи для них повинні працювати майже повністю універсально. Раніше ми вже розглянули систему модифікаторів та уніфіковану систему здоров’я сутностей і тепер дійшли до ворогів. В основі кожного ворога є клас Enemy, котрий реалізує інтерфейс IDamagable, що можна побачити на Лістингу 15.

Лістинг 15 — Реалізація інтерфейсу IDamagable класом Enemy

```
    public void TakeDamage(DamageInfo damageInfo)
    {
        Debug.Log($"Enemy take damage!");
        modifierHandler.ProcessDamageTaken(ref damageInfo, damagable);
        damageReceiver.TakeDamage(damageInfo);
    }

    public void DealDamage(DamageInfo damageInfo)
    {
        if (!damageInfo.Target.TryGetComponent<IDamagable>(out var
damagable))
        {
            return;
```

```
        }

        modifierHandler.ProcessDamageDealt(ref damageInfo, damagable);
        damagable.TakeDamage(damageInfo);

        modifierHandler.EventEmitter?.TriggerEvent(new
EnemyMadeDamageEvent
        {
            DamageInfo = damageInfo,
            Target = damagable
        });
    }
```

Як бачимо реалізація отримання шкоди ворогом та нанесення шкоди також використовує Modifier Handler для нанесення шкоди, як і в гравця, що дозволить в майбутньому без великих проблем додати їм цікаві властивості як заморозка гравця або ще щось.

Також вороги окрім того що повинні мати інтерфейс для нанесення та отримання шкоди повинні також уміти переміщатись. Для переміщення ворогів я використав компонент Unity - AI Navigation System. Дана система дозволяє без великих затрат часу переміщати агентів по навігаційній сітці що значно спрощує реалізацію навігації ворогів. Також необхідно було врахувати, що вороги можуть перебувати як мінімум в 3 станах, а саме стан Idle, коли ворог не бачe свою ціль, стан Chase, коли ворог намагається зблизитись з гравцем та в стані Attack, коли ворог уже дійшов до цілі на необхідну дистанцію для виконання атаки. Для уніфікованої реалізації переміщення ворогів було створено базовий абстрактний клас Enemy Movement Base. Даний скрипт уже включає в себе реалізацію станів ворогів та налаштування дистанції на яких повинні вмикатись ті чи інші стани, а також даний скрипт

викликає атаку через ще один скрипт - Advanced Enemy. Advanced enemy скрипт котрий тримає в своїх полях час останньої атаки, конфіг поточного ворога. На основі класу AdvancedEnemy було створено декілька різних ворогів, таких як:

- Melee Enemy
- Ranged Enemy
- Sniper Enemy
- Mini Boss

Через наявність в скрипті Enemy Movement Base та Advanced Enemy деяких важливих реалізацій кількість необхідної роботи для створення різних видів ворогів значно зменшилася. Необхідно було всього-лиш створити свої реалізації абстрактного класу Enemy Movement Base, що являється дуже тривіальною задачею і з роботи яка реально требувала часу залишилося лише реалізувати атаки ворогів для різних класів. Почнемо з найпершого ворога - Melee Enemy(ворог ближнього бою). Для даного ворога необхідно було трохи доробити методи для визначення того чи повернутий ворог до своєї цілі під час атаки, а також реалізувати сам метод атаки котрий можна побачити на Лістингу 16. З нього видно що спочатку перед ворогом за допомогою Physics.OverlapBox шукаються коллайдери цілей. Якщо коллайдери з таким шаром було знайдено то атака заходить в цикл де для кожного колайдера викликається накладання фізичного відкидання цілей від позиції ворога. Це дозволяє посилити відчуття від атак ворогів. Після виконання фізичної атаки ворог починає шукати в коллайдерах IDamagable компонент, виконує обробку атаки своїм Modifier Handler, а потім наносить оброблену модифікаторами шкоду по цілі.

Лістинг 16 — MeleeEnemy Attack Sequence

```
private void ExecuteMeleeAttack()
    {
```

```
        Vector3 attackPosition = attackOrigin.position + attackOrigin.forward * attackForwardOffset;

        Collider[] hitColliders = Physics.OverlapBox(
            attackPosition,
            new Vector3(attackWidth * 0.5f, 1f, config.attackRange * 0.5f),
            attackOrigin.rotation,
            attackableLayers
        );

        foreach (var collider in hitColliders)
        {
            Rigidbody rb = collider.attachedRigidbody;
            if (rb != null)
            {
                float distance = Vector3.Distance(transform.position, rb.worldCenterOfMass);
                float distanceNormalized = Mathf.InverseLerp(knockbackForce, 0f, distance);
                float adjustedExplosionForce = knockbackUpwardForce * knockbackForce * (1 - distanceNormalized) * knockbackForce * 0.025f;

                rb.AddExplosionForce(adjustedExplosionForce, transform.position, attackWidth, knockbackUpwardForce, ForceMode.Impulse);
            }

            Vector3 toTarget = (collider.transform.position - transform.position).normalized;
```

```
            float angleToTarget = Vector3.Angle(transform.forward,
toTarget);

            if (angleToTarget <= attackAngle * 0.5f)
            {
                if (collider.TryGetComponent<IDamagable>(out var
target))
                {
                    DamageInfo damageInfo = new DamageInfo
                    {
                        Amount = config.damage,
                        Source = gameObject,
                        Target = collider.gameObject,
                    };

                    modifierHandler.ProcessDamageDealt(ref damageInfo,
target);
                    target.TakeDamage(damageInfo);
                }
            }
        }
    }
```

Наступним було необхідно реалізувати Range Enemy. Тут ситуація дуже схожа на створення ворога ближнього бою, але тут уже було необхідно виконувати стрільбу снарядами(Projectile) по своїй цілі. Так-як гравцем також планувалось виконувати дистанційні атаки за допомогою снарядів то було вирішено винести необхідну логіку в окремий клас Projectile(дивитись Лістинг 17).

Лістинг 17 — Клас Projectile

```
[RequireComponent(typeof(Rigidbody))]

public class Projectile : MonoBehaviour

{

    [Header("Explosion settings")]

    [SerializeField] private List<Modifier> modifiers = new();

    [SerializeField] public float damage = 10f;

    [SerializeField] private float lifeTime = 5f;

    [SerializeField] private float explosionForce = 100f;

    [SerializeField] private float explosionRadius = 10f;

    [SerializeField] private float upwardsModifier = 5f;

    [SerializeField] private LayerMask attackableLayers;


    [Header("Visuals")]

    public TemporaryEffect explosionEffectPrefab;


    protected GameObject owner;
    protected ModifierHandler modifierHandler;
```

Оглянемо які поля має основний клас снаряду. По перше бачимо що він потребує мати компонент Rigidbody на ігровому об’єкті. Це необхідно для того щоб обробляти фізичні зіткнення з іншими об’єктами. Далі бачимо категорію полів *Explosion Settings.* Дана категорія як видно з назви керує різноманітними параметрами пов’язаними з тим як працює вибух. Це включає в себе кількість шкоди, максимальний час життя снаряду, сила фізичного вибуху, радіус в якому працює даний вибух, сила підкидання вверх фізичних тіл від вибуху, а також вибір ігрових шарів яким власник даного снаряду зможе наносити шкоду. Наприклад для гравця це шар Enemy, а для ворога це шар Player.

Далі бачимо категорію *Visuals.* Вона відповідає за те який префаб буде інстантіватись на місці вибуху. Для такої-ж універсальності було створено і клас Temporary Effect, котрий в своїй основі має вбудований час життя ефекту, звук, візуальний ефект даного тимчасового об’єкту. Нижче в полях бачимо окремо поле для зберігання посилання на власника снаряду, а також modifier handler, котрий відповідає за те які вбудовані ефекти має снаряд. Нижче на Лістингу 18 можна побачити реалізацію класу *Temporary Effect.*

Лістинг 18 — TemporaryEffect

```
public class TemporaryEffect : MonoBehaviour
{
    [SerializeField] private float lifeTime = 1f;
    [SerializeField] private AudioSource audioSource;
    [SerializeField] private float clipStartTime = 0f;

    private void Start()
    {
        StartCoroutine(LifetimeCoroutine());
    }

    private IEnumerator LifetimeCoroutine()
    {
        if (audioSource != null && audioSource.clip != null)
        {
            audioSource.pitch = Random.Range(0.9f, 1.15f);
            audioSource.volume = Random.Range(0.9f,1f);
            audioSource.time = clipStartTime;
            audioSource.PlayDelayed(Random.Range(0, 0.1f));
```

```
            yield return new WaitForSeconds(audioSource.clip.length);
        } else
        {
            yield return new WaitForSeconds(lifeTime);
        }

        Destroy(gameObject);
    }
}
```

Далі створений клас Projectile використовується для атаки гравця Ranged та Sniper ворогами, не забуваючи про можливе використання даного снаряду Боссом та самим гравцем. Нижче, на Лістингу 19 можна побачити використання Projectile для атаки по гравцю:

Лістинг 19 — Атака RangedEnemy

```
protected virtual void Attack()
        {
            lastAttackTime = Time.time;

            if (config.projectilePrefab != null && projectileSpawnPoint != null)
            {
                Projectile projectile = Instantiate(
                    config.projectilePrefab,
                    projectileSpawnPoint.transform.position,
                    projectileSpawnPoint.transform.rotation
                );
```

```
projectile.transform.LookAt(playerTransform.position);


                projectile.SetupProjectile(
                    config.damage,
                    gameObject,
                    config.innateModifiers,
                    eventEmitter
                );
            }
        }
```

Як видно з коду префаб проджектайла дістається з конфігу ворога, далі даний проджектайл створюється на сцені, повертається в сторону цілі. Після спавна та початкової ініціалізації снаряду він ініціюється ще й параметрами ворога, такими як шкода, власник снаряду, модифікатори ворога та EventEmitter снаряду для підняття необхідних подій під час зіткнення снаряду з ігровими об’єктами. Якщо розглядати реалізацію ворога Sniper Enemy, то вона майже нічим не відрізняється окрім того, що ворог виконує “зарядження” атаки перед пострілом.

А ось що дійсно цікаво розглянути так це скрипт роботи Міні Босса. Він уже не просто атакує гравця снарядами поодинокими пострілами, а атакує свою ціль серією пострілів. На наступному лістингу розглянемо дану реалізацію.

Лістинг 20 — MiniBoss.VoleyAttack

```
    private IEnumerator PerformVolley()
    {
        if (isPerformingAction) yield break;
```

```
        isPerformingAction = true;
        movementBase?.StopMovement();
        // charge / aim (sniper-like in higher phases)
        float charge = (currentPhase == 1) ? 0.1f : (currentPhase == 2
? 0.6f : 0.9f);
        yield return new WaitForSeconds(charge);
        Transform spawn = projectileSpawnPoint ?? attackOrigin;
        Vector3 targetPos = Player.Instance.transform.position;
        Vector3 baseDir = (targetPos - spawn.position).normalized;
        int count = volleyCount + (currentPhase == 3 ? 2 : 0);
        float spread = volleySpreadAngle * (currentPhase == 1 ? 1f :
(currentPhase == 2 ? 0.8f : 0.6f));
        for (int i = 0; i < count; i++)
        {
            // compute spread
            float t = (count == 1) ? 0f : (i - (count - 1) * 0.5f) /
(count - 1);
            float angle = t * spread;
            Quaternion rot = Quaternion.AngleAxis(angle, Vector3.up) *
Quaternion.LookRotation(baseDir);
            if (projectilePrefab != null)
            {
                Projectile p = Instantiate(projectilePrefab,
spawn.position, rot);
                p.SetupProjectile(config.damage * (currentPhase == 3 ?
1.4f : 1f), gameObject, config.innateModifiers, eventEmitter);
            }

            yield return new WaitForSeconds(volleyDelayBetween);
```

```
        }

        // set next volley cooldown (faster in higher phases)

        nextVolleyTime = Time.time + volleyCooldown * Mathf.Lerp(1f,
0.6f, (currentPhase - 1) * 0.6f);

        lastAttackTime = Time.time;

        isPerformingAction = false;

        movementBase?.ResumeMovement();

    }
```

Розглянемо дану атаку. По перше бачимо що Advanced Enemy міні босса викликає команду зупинки переміщення боса. Далі проходить деяка міні-фаза зарядження атаки і після цього розраховується в залежності від поточної фази боса скільки снарядів створити та як сильно робити їхній розкид від цільової точки атаки. Після цього в циклі уже знайомим нам методом створюються снаряди та вираховується час через який буде виконана наступна атака дальнього бою та викликається команда продовження переміщення ворога.

4.1.5 Система генерації рівня

Так-як для даного проєкту було прийнято рішення створити систему процедурної генерації ігрового рівня то після невеликого дослідження було обрано генерацію рівня по воксельній сітці. Воксельна сітка - це сітка в якій блоки рівня складаються з кубів розміром 1x1x1 юніти. Кожен блок може бути як пустим, так і бути блоком стіни кімнати або коридору. Також він може бути блоком полу та потолку кімнат та коридорів. Вибір воксельної сітки був обраний в більшій мірі через можливість більш точно керувати розміщенням кімнат на рівні та проходами між ними. До цього було також розглянуто генерацію мапи за допомогою Binary Space Partitioning, але даний спосіб мені не вдалося використати для генерації цікавих рівнів.

Для генерації таких рівнів перше що було створено так це клас Voxel Grid(Лістинг 21).

Лістинг 21 — Частина класу Voxel Grid

```
public class VoxelGrid
{
    public Vector3Int Size { get; private set; }
    private Cell[,,] cells;
    private HashSet<(int x, int z)> occupiedXZ = new();

    public VoxelGrid(Vector3Int size)
    {
        Size = size;
        cells = new Cell[size.x, size.y, size.z];

        for (int x = 0; x < size.x; x++)
        {
            for (int y = 0; y < size.y; y++)
            {
                for (int z = 0; z < size.z; z++)
                {
                    cells[x, y, z] = new Cell
                    {
                        type = CellType.Empty,
                        roomId = -1
                    };
                }
            }
        }
    }
```

```
public bool IsInside(Vector3Int point)
{
    return point.x >= 0 && point.x < Size.x &&
           point.y >= 0 && point.y < Size.y &&
           point.z >= 0 && point.z < Size.z;
}

public Cell GetCell(Vector3Int position)
{
    if (!IsInside(position))
        return new Cell { type = CellType.Empty, roomId = -1 };

    return cells[position.x, position.y, position.z];
}

public void SetCell(Vector3Int pos, CellType type, int roomId = -1)
{
    if (!IsInside(pos))
    {
        return;
    }
    cells[pos.x, pos.y, pos.z].type = type;
    cells[pos.x, pos.y, pos.z].roomId = roomId;
    MarkOccupiedXZ(pos);
}
```

Розглянемо код воксельної сітки. По-перше бачимо параметр розміру сітки, далі бачимо тривимірний масив клітинок сітки та хешсет зайнятих клітинок по X Z координат, що було необхідно зробити для використання при генерації коридорів та при розміщенні кімнат на сітці.

Також в лістингу присутній метод ініціалізації сітки пустими клітинами, а також методи для присвоєння та отримання клітин сітки за координатами.

Далі потрібно було визначитись з тим кімнати якої форми можуть бути на координатній сітці. Спочатку було додано 3 форми, а саме квадратну, Г-шейп, та круглу кімнату, але під час тестування гри та під час генерації оптимальних коллайдерів для кімнат було прийнято рішення тимчасово перейти просто на квадратні кімнати щоб успішно завершити проєкт. Отже клас форми квадратної кімнати можна побачити на Лістингу 22.

Лістинг 22 — Квадратна кімната

```
[Serializable]
public class RectShape : Shape
{
    public Vector3Int Size;
    public override List<Vector3Int> GetOccupiedCells(Vector3Int origin)
    {
        List<Vector3Int> cells = new();
        for (int x = 0; x < Size.x; x++)
        {
            for (int y = 0; y < Size.y; y++)
            {
                for (int z = 0; z < Size.z; z++)
                {
                    cells.Add(origin + new Vector3Int(x, y, z));
```

```
                }
            }
        }
        return cells;
    }
    public override List<Vector3Int> GetPassageCells(Vector3Int origin)
    {
        List<Vector3Int> passages = new();
        int floorY = 1;
        passages.AddRange(GeneratePassage(
            origin + new Vector3Int(Size.x / 2 - PassageWidth / 2,
floorY, 0),
            new Vector3Int(1, 0, 0), new Vector3Int(0, 1, 0)));
        passages.AddRange(GeneratePassage(
            origin + new Vector3Int(Size.x / 2 - PassageWidth / 2,
floorY, Size.z - 1),
            new Vector3Int(1, 0, 0), new Vector3Int(0, 1, 0)));
        passages.AddRange(GeneratePassage(
            origin + new Vector3Int(0, floorY, Size.z / 2 - PassageWidth
/ 2),
            new Vector3Int(0, 0, 1), new Vector3Int(0, 1, 0)));
        passages.AddRange(GeneratePassage(
            origin + new Vector3Int(Size.x - 1, floorY, Size.z / 2 -
PassageWidth / 2),
            new Vector3Int(0, 0, 1), new Vector3Int(0, 1, 0)));
        return passages;
    }

    public override List<Vector3Int> GetInnerCells(Vector3Int origin)
```

```
    {
        List<Vector3Int> inner = new();
        int y = origin.y;
        for (int x = 1; x < Size.x - 1; x++)
        {
            for (int z = 1; z < Size.z - 1; z++)
            {
                inner.Add(new Vector3Int(origin.x + x, y, origin.z +
z));
            }
        }

        return inner;
```

Як бачимо даний клас форми має в собі декілька корисних методів, таких як метод для отримання зайнятих клітин, метод отримання клітинок в котрих будуються проходи та метод для отримання клітин котрі знаходяться всередині кімнати. Все це використовується далі в проєкті.

Після визначення форм кімнат було створено клас RoomArchetype поля котрого можна побачити на Лістингу 23.

Лістинг 23 — RoomArchetype

```
    [CreateAssetMenu(fileName = "RoomArchetype", menuName =
"Dungeon/RoomArchetype")]
    public class RoomArchetype : ScriptableObject
    {
        public string roomName;
        public RoomType type;
```

```
[Header("Геометрія")]
public RoomShapeType shapeType;
public Shape shape => GetShape(shapeType);
public Vector3Int size;
public int radius;

[Header("Контент")]
public GameObject floorPrefab;
public GameObject wallPrefab;
public GameObject roofPrefab;
public List<Feature> features;

[Header("Баланс")]
[Range(0f, 1f)] public float weight = 0.5f;
```

Як бачимо цей клас виступає конфігураційним файлом для багатьох кімнат. Він дозволяє гнучко та легко модифікувати те якого розміру може бути кімната, який шанс її випадіння, зовнішній вигляд кімнати, а також форму на тип кімнати. Тип кімнати визначає чи це звичайна кімната, бос кімната, кімната з цінностями або кімната-пастка, що додає грі варіативності.

Після того я ми вже знаємо які типи кімнат будуть розміщуватись на рівні необхідно згенерувати граф рівня. За допомогою цього графу буде зрозуміло між якими кімнатами необхідно буде генерувати проходи. В генераторі наведеному в Лістингу 24 ми бачимо те, як генерується граф кімнат разом зі стартовою кімнатою, стандартними кімнатами та рандомними додатковими кімнатами по типу пастки або кімнати з цінностями.

Лістинг 24 — Генерація графу кімнат

```
public RoomGraph Generate()
```

```
    {
        Random.InitState(DateTime.Now.Second + DateTime.Now.Millisecond
+ DateTime.Now.Hour);
        RoomGraph roomGraph = new RoomGraph();

        roomGraph.AddRoom("start", RoomType.Start);
        for (int i = 0; i < RoomCount; i++)
            roomGraph.AddRoom($"room_{i}", RoomType.Normal);
        roomGraph.AddRoom("boss", RoomType.Boss);

        roomGraph.Connect("start", "room_0");
        for (int i = 0; i < RoomCount - 1; i++)
            roomGraph.Connect($"room_{i}", $"room_{i + 1}");
        roomGraph.Connect($"room_{RoomCount - 1}", "boss");

        for (int i = 0; i < RoomCount; i++)
        {
            if (Random.value < 0.4f)
            {
                string sideId = $"side_{i}";
                RoomType type = (Random.value < 0.5f) ?
RoomType.Treasure : RoomType.Trap;
                roomGraph.AddRoom(sideId, type);
                roomGraph.Connect($"room_{i}", sideId);
            }
        }

        return roomGraph;
```

```
}
```

Наступний крок к генерації ігрового рівня це розміщення кімнат на рівні. Для розміщення кімнат було створено клас Room Placer. Даний клас намагається розмістити кімнати шукаючи наступну позицію для спроби розміщення по спіралі від початкової точки перевірки. Якщо кімнату не вдалось розмістити відразу, то за збільшенням числа поточного кроку кімната намагається розміститися все дальше від початкової точки поки все ж не зможе стати на своє місце. Нижче на Лістингу 25 можна побачити метод який в залежності від номера спроби віддає наступну позицію для спроби розміщення кімнати.

Лістинг 25 — RoomPlacer.GetOffsetOrigin

```
private static Vector3Int GetOffsetOrigin(Vector3Int origin, int attempt, int step)
    {
        if (attempt == 0) return origin;

        int radius = (attempt / 8 + 1) * step * 2;
        float angle = (attempt % 8) * Mathf.PI / 4f;
        int dx = Mathf.RoundToInt(Mathf.Cos(angle) * radius);
        int dz = Mathf.RoundToInt(Mathf.Sin(angle) * radius);
        return new Vector3Int(origin.x + dx, origin.y, origin.z + dz);
    }
```

Також було розроблено метод який перевіряє чи пересікається поточка кімната з іншими клітинами на рівні. Якщо таких клітин не було знайдено, то кімната розміщується на координатній сітці.

Лістинг 26 RoomPlacer.TryPlaceInternal

```
private static bool TryPlaceInternal(VoxelGrid grid, RoomArchetype archetype, Vector3Int origin, int roomId, int minDistance)
    {
        List<Vector3Int> cells;
        try
        {
            cells = archetype.shape.GetOccupiedCells(origin);
        }
        catch (System.Exception ex)
        {
            Debug.LogError($"[RoomPlacer] Exception in GetOccupiedCells for '{archetype.name}' at {origin}: {ex}");
            return false;
        }
        if (cells == null || cells.Count == 0)
            return false;
        foreach (var cell in cells)
        {
            if (!grid.IsInside(cell))
                return false;
            var cellData = grid.GetCell(cell);
            if (cellData.type != CellType.Empty)
                return false;

            for (int dx = -minDistance; dx <= minDistance; dx++)
            {
                for (int dz = -minDistance; dz <= minDistance; dz++)
                {
```

```
                    Vector3Int checkPos = new Vector3Int(cell.x + dx,
cell.y, cell.z + dz);
                    if           (grid.IsInside(checkPos)           &&
grid.GetCell(checkPos).type != CellType.Empty)
                        return false;
                }
            }
        }
        // розміщення
        foreach (var cell in cells)
        {
            grid.SetCell(cell, CellType.Room, roomId);
        }
        MarkRoomBufferZone(grid, cells, minDistance, roomId);
        return true;
    }
```

Спочатку в методі TryPlaceInternal перевіряється чи всі клітини Room Archetype валідні(знаходяться всередині координатної сітки), а також чи дану клітину вже щось займає. Якщо клітини проходять ці перевірки то далі відбувається перевірка чи вільні клітини на мінімальній дистанції між кімнатами. Якщо і це пройшло успішно то тоді кімната успішно розміщується на координатній сітці і генератор мапи переходить до наступних кімнат.

Після успішного розміщення кімнат на рівні необхідно використовуючи той-же граф кімнат згенерувати проходи між ними. Для генерації проходів було використано алгоритм пошуку шляху A*, де для пошуку найкоротшого шляху необхідно вираховувати Манхеттенську відстань для різних клітин по яким проходить шлях. Дана манхеттенська відстань в моєму випадку просто

вираховує відстань між поточною клітиною та цільовою клітиною чого в моєму випадку вистачає(див. Лістинг 26).

Лістинг 27 — CalculateManhattanDistance

```
  private float CalculateManhattanDistance(Vector3Int a, Vector3Int b)
    {
        return Mathf.Abs(a.x - b.x) + Mathf.Abs(a.z - b.z);
    }
```

Також при генерації коридорів необхідно було вичисляти ціну проходу через деякі клітини шляху. В моєму випадку було обрано поставити ціну проходу на мінімальну, якщо коридор буде пересікатися з іншим коридором, а на пустій клітинці поставити ціну проходу рівну одному. А якщо клітина зайнята чимось іншим ніж пустотою або полом коридору, то ціна проходу на таку клітину встановлюється максимальна - float.MaxValue. На Лістингу 27 можна побачити реалізацію даного методу.

Лістинг 28 — Вирахування ціни проходження клітини

```
    private float CalculateMoveCost(Vector3Int cell)
    {
        if (_grid.IsXZOccupiedByRoom(cell))
            return float.MaxValue;

        var cellType = _grid.GetCell(cell).type;

        if (cellType == CellType.Empty)
            return 1f;

        if (cellType == CellType.CorridorFloor)
```

```
            return 0.5f;


        return float.MaxValue;
    }
```

Після того як уже було визначено, які клітини чим зайняті та які кімнати з якими з’єднані, необхідно це все візуалізувати. Спочатку була спроба просто створити для кожної клітини свій об’єкт, який би мав відразу свій колайдер та меш з матеріалом, але це виявилось жахливим рішенням, так-як через велику кількість малих колайдерів страждав процесор, а через велику кількість мешів з матеріалами навіть з увімкненим GPU Instancing уже почав страждати як процесор так і відеокарта. Це все відбувається через те що кожен ігровий об’єкт надіслав свій окремий запит на рендер об’єктів на відеокарту, а коли таких запитів багато то навантаження на процесор зростає.

Таких об’єктів могло бути і декілька тисяч на рівень. Як дану проблему було виявлено було прийнято використовувати більш ефективні способи рендеру кімнат та коридорів. Тут на допомогу приходить статичний метод Graphics.DrawMeshInstanced. Він дозволяє в один виклик передавати матрицю розміщення та поворотів мешів, меш та матеріал для рендеру. Завдяки тому, що за допомогою такого виклику можна за один виклик рендеру до відеокарти рендерити відразу тисячі мешів з одним матеріалом навантаження на процесор і відеокарту, а також на відеопам’ять різко впали.

Код реалізації даного рендеру для кімнат можна побачити нижче на Лістингу 29.

Лістинг 29 — Використання Graphics.DrawMeshInstanced для рендеру кімнат

```
private void Update()
    {
        if (!initialized || RoomInstance == null) return;
```

```
        // === Рендер ===
        if (drawFloor && RoomInstance.floorMatrix.Count > 0)
            Graphics.DrawMeshInstanced(RoomInstance.FloorMesh,       0,
RoomInstance.FloorMat, RoomInstance.floorMatrix);
        if (drawWalls && RoomInstance.wallMatrix.Count > 0)
            Graphics.DrawMeshInstanced(RoomInstance.WallMesh,        0,
RoomInstance.WallMat, RoomInstance.wallMatrix);
        if (drawRoof && RoomInstance.roofMatrix.Count > 0)
            Graphics.DrawMeshInstanced(RoomInstance.RoofMesh,        0,
RoomInstance.RoofMat, RoomInstance.roofMatrix);


        if (cullingTarget != null && cachedColliders != null &&
cachedColliders.Length > 0)
        {
            cullingTimer += Time.deltaTime;
            if (cullingTimer >= cullingUpdateInterval)
            {
                cullingTimer = 0f;
                HandleColliderCulling();
            }
        }
    }
```

Як бачимо в Room Renderer в функції яка викликається кожен кадр відбувається виклик рендеру мешів для стін, полу та даху кімнати, а також відбувається Collider Culling, який вимикає колайдери які знаходяться занадто далеко від гравця. Дана оптимізація і дозволила грі існувати та нормально працювати навіть на такому слабкому залізі як GPU Intel UHD 620.

Також було створено окремі методи для генерації оптимальних колайдерів для кімнат та коридорів. Дані методи брали одну з точок кімнати або коридору та шукали сусідні клітини, які в поєднанні з поточною клітиною змогли б утворити BoxCollider, котрі являються найпростішими для розрахунку процесором фізики в грі.

Після успішного вирішення проблеми рендеру ігрового рівня потрібно було реалізувати ще декілька невеликих систем, а саме Room NavMesh Generator, який генерує оптимальний нав меш для переміщення навігаційних агентів, GameRoom Controller, котрий генерує колайдер тригера для активації кімнати, а саме для початку спавну ворогів в кімнаті, ворота, котрі зачиняються при активації кімнати та відчиняються після перемоги ворогів. Почнемо з опису GameRoom Controller як з найвищого шару абстракції ігрової логіки кімнати.

Лістинг 30 — GameRoomController - ініціалізація та поля класу

```
[RequireComponent(typeof(RoomGates), typeof(RoomEnemySpawner))]
public class GameRoomController : MonoBehaviour
{
    public UnityEvent<RoomInstance> OnRoomBossDied = new();
    private RoomInstance gameRoom;
    private RoomGates roomGates;
    private RoomEnemySpawner enemySpawner;
    private bool roomActive = false;
    private int difficulty = 2;

    [SerializeField] private Material spawnPointsMaterial;

    [SerializeField] private LayerMask targetLayerMask;
```

```
        public void Initialize(RoomInstance room)
        {
            gameRoom = room;

            // Привязываем контроллер непосредственно к RoomGO
            transform.SetParent(gameRoom.RoomGO.transform, false);
            transform.localPosition = Vector3.zero;
            transform.localRotation = Quaternion.identity;
            transform.localScale = Vector3.one;

            roomGates = GetComponent<RoomGates>();
            enemySpawner = GetComponent<RoomEnemySpawner>();

            enemySpawner.OnBossDied.AddListener(OnBossDied);
enemySpawner.OnAllEnemiesDestroyed.AddListener(OnAllEnemiesDied);

            GenerateTrigger();
        }
```

Як бачимо в полях класу зберігаються основні компоненті якими керує GameRoomController, а саме RoomGates і RoomEnemySpawner. В методі Start відбувається ініціалізація контроллера кімнати та його компонентів, а також викликається генерація тригера для активації кімнати. Також можна побачити івенти які піднімає кімната якщо всі вороги померли та/або босс даної кімнати помер.

4.1.6 Ігровий менеджмент

Після того як побудовано всі основні системи необхідно побудувати систему найвищого шару - GameManager. Дана система відповідає за основний Game Loop. Даний клас GameManager відповідає за складність гри, скільки рівнів повинен перемогти гравець для перемоги в забігу, а також відповідає за перегенерацію мапи з новим ключем після перемоги на кожному рівні. Для прикладу оглянемо процедуру рестарту рівня гри при виграшу гравцем рівня на Лістингу 31.

Лістинг 31 — GameManager.RestartLevel

```
    public void RestartLevel()
    {
        StopwatchTimer.Instance.PauseTimer();
        currentLevel += 1;
        if (currentLevel >= levelsToWin)
        {
            float seconds = StopwatchTimer.Instance.GetTimeSeconds();
            string                    secondsString                    =
StopwatchTimer.Instance.GetTimeString();

            SaveManager.Instance.SaveMaxKills();
            SaveManager.Instance.SaveMaxPoints();
            SaveManager.Instance.SaveRecordWinningTime(seconds);

            WinningGameScreenUI.Instance.Activate(currentLevel);
            Cursor.lockState = CursorLockMode.None;
            Cursor.visible = true;
            return;
        }
        Debug.Log($"Player Won -> {currentLevel}");
        WinningScreenUI.Instance.Deactivate();
        Debug.Log("[GameManager] Виклик рестарту рівня.");
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
        DestroyAllGeneratedRoomGates();
        generator.StartGenerating();
    }
```

Як бачимо спочатку після проходження рівня секундомер проходження гри ставиться на паузу, так-як той час за який буде відбуватися перегенерація рівня не повинен впливати на час проходження гри, так-як на персональних комп’ютерах з різною потужністю генерація ігрового рівня та його очистка перед наступною генерацію може займати дуже різну кількість часу. Після

цього порівнюється кількість пройдених рівнів з кількістю необхідною для успішного завершення гри. Якщо дана умова виконалась, то генерація рівня не відбувається, а вмикається екран перемоги гри та результати проходження записуються через окремий клас SaveManager в PlayerPrefs для майбутнього відображення на екрані статистики в головному меню гри. Якщо все ж гравець не пройшов необхідну кількість рівнів то спочатку відбувається очистка рівня, а потім відбувається нова генерація рівня.

4.2 Висновки до розділу 4

1. Отже було досліджено проблеми з якими я зіткнувся під час розробки гри, а саме проблему створення системи переміщення гравця, системи ворогів, генерації рівнів і так далі.
2. Розроблено найосновніші системи гри, а саме системи нанесення шкоди, модифікаторів, генерації рівнів, системи ворогів, менеджменту гри, інтерфейсу та багато іншого. Після створення даних систем можна сказати, що гра жанру FPS Roguelike була створена на рівні прототипу.

3. В реалізації гри гравець має можливість повністю пройти гру побачивши всі створені системи в дії, а саме ігровий процес виглядає наступним чином:

1. Гравець з’являється в ігровому світі
2. Проходить по коридорам в кімнати
3. Кожна кімната генерує деяку кількість ворогів для її проходження
4. Після проходження декількох кімнат гравець може натрапити на кімнату боса.
5. В кімнаті боса він бореться з босом
6. Якщо гравець програв босу то йому показується екран програшу на якому він може повернутися в головне меню або почати гру з початку
7. Якщо гравець виграє боса, то він має можливість пройти наступний рівень.

8. Після проходження трьох рівнів гравець бачить екран виграшу гри та статистика його проходження записується в його статистику, яку він може побачити в головному меню

4.3 Що можна покращити в виконаній роботі

Через велику складність створення ігор такого жанру з процедурною генерацією світа було створено лише MVP проєкту, де гравець може пройти основний цикл геймплею, скористатися однією зброєю та отримувати рандомні модифікатори з ворогів, але це лише мінімальний прототип. В подальшому для розвитку даної гри необхідно покращити UI, додати ігрові моделі та анімації для ворогів(було тимчасово відкладено це через слабкість ПК на якому створювалась дана гра), оптимізувати генерацію мапи та зробити рівні більш різнотипними. Також можна було б додати більше варіацій ворогів, кімнат та оптимізувати круглі та кімнати інших форм для більшої варіативності рівнів та багато чого іншого. Даний проект виявився досить амбітним для виконання його в якості дипломної роботи магістра тому було створено лише прототип гри в жанрі FPS Roguelike.

ВИСНОВКИ

За результатами дослідження та розробки відеогри в жанрі FPS Roguelike було зроблено такі висновки:

1. Досліджено теоретичні та практичні аспекти створення відеоігор у жанрі FPS Roguelike. Проаналізовано ключові проблеми, які постають при розробці таких проєктів, зокрема, необхідність ефективної реалізації процедурної генерації рівнів, розробки системи ворогів, механізмів мотивації гравця (через прогрес та посилення), а також забезпечення оптимального балансу зброї та створення читабельного користувацького інтерфейсу (UI) в умовах динамічного геймплею. Проведено огляд аналогічних успішних відеоігор (Roboquest, BPM: Bullets Per Minute, Gunfire Reborn, Deadlink та ін.), що дозволило визначити та описати важливі аспекти та найкращі практики жанру.
2. Обрано та обґрунтовано вибір Unity як основного засобу розробки. Цей вибір був обумовлений можливістю розробки проєкту на слабкому комп’ютері, сильною спільнотою та меншим порогом входження для ефективної реалізації ігрових механік. Для програмування використовувалася мова програмування C# , що забезпечило необхідну гнучкість та ефективність при роботі з ігровою логікою.
3. Розроблено прототип відеогри в жанрі FPS Roguelike. В рамках проєкту були реалізовані такі ключові компоненти:
   - Базова механіка стрільби від першої особи та пересування.
   - Система процедурної генерації рівнів/локацій, що забезпечує унікальність кожного проходження.
   - Реалізація чотирьох типів ворогів із базовим штучним інтелектом.
   - Система модифікаторів , що дозволяє гравцеві посилюватися в процесі гри та створювати різноманітні збірки.
   - Розроблено HUD для відображення необхідної інформації для гравця з легкою читабельністю в динамічних боях.

4. Проаналізовано ефективність розроблених механік з точки зору ігрового балансу. Зокрема, було проаналізовано співвідношення між потужністю посилень та складністю ворогів, що дозволило побудувати адекватну криву складності. Зроблено висновки щодо необхідності щодо необхідності оптимізації генерації ігрового рівня, покращення ігрового інтерфейса та візуальної та звукової частини гри.
5. Надано рекомендації щодо подальшого розвитку та використання розробленого проєкту. Для використання як основи для інших проєктів рекомендовано:
   - Розширити контент: додати нові біоми, класи персонажів та зброю, нових ворогів.
   - Провести масштабне ігрове тестування для тонкого налаштування ігрового балансу.
   - Оптимізувати генерацію рівнів та генерацію колайдерів для сцени.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. C# Data Structures and Algorithms: Harness the Power of C# to Build a Diverse Range of Efficient Applications, лютий 2024 р. С. 374 (дата звернення: 11.07.2025).
2. Learning C# by Developing Games with Unity: Get to Grips with Coding in C# and Build Simple 3D Games in Unity 2022 from the Ground Up, листопад 2022 р. С. 460 (дата звернення: 10.06.2025).
3. Level Up!: The Guide to Great Video Game Design, квітень 2014 р. С. 535 (дата звернення: 11.07.2025).
4. Rules of Play: Game Design Fundamentals (Mit Press), вересень 2003 р. С. 688 (дата звернення: 11.06.2025).
5. Gunfire Reborn Wiki : веб-сайт. URL: https://gunfirereborn.fandom.com/wiki/Gunfire_Reborn_Wiki (дата звернення: 07.06.2025).
6. Roboquest Wiki : веб-сайт. URL: https://roboquest.fandom.com/wiki/Roboquest_Wiki (дата звернення: 07.05.2025).
7. Unity Learn : веб-сайт. URL: https://learn.unity.com/ (дата звернення: 07.05.2025).
8. Google Images : веб-сайт. URL: https://images.google.com/ (дата звернення: 07.05.2025).
9. Rogueliker — Топ найкращих FPS Roguelike : веб-сайт. URL: https://rogueliker.com/fps-roguelikes/ (дата звернення: 07.05.2025).
10. Unity Documentation. URL: https://docs.unity.com/ (дата звернення: 07.05.2025).
11. Figma: the collaborative interface design tool. : веб-сайт. URL: https://www.figma.com/ (дата звернення: 05.05.2025).
12. Unity Muse AI : веб-сайт. URL: https://unity.com/en/products/muse (дата звернення: 10.06.2025).

13. Inworld AI : веб-сайт. URL: https://inworld.ai/ (дата звернення: 10.06.2025).
14. Layer AI. URL: https://www.layer.ai/ (дата звернення: 10.06.2025).
15. Polyhive AI: веб-сайт. URL: https://polyhive.ai/ (дата звернення: 10.06.2025).
16. Unity Sentis: веб-сайт. URL: https://unity.com/en/products/sentis (дата звернення: 10.06.2025).
17. Unity Render Pipelines: веб-сайт. URL: https://docs.unity3d.com/Manual/render-pipelines.html (дата звернення: 10.06.2025).

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Красножон Іван Олегович, студент 2 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz20bd-104@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Покращення ігрової механіки комп'ютерних ігор жанру FPS Roguelike Game**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- підтверджую, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 01.12.2025 _____ Красножон Іван Олегович
(підпис) (студент)

Дата 01.12.2025 Підпис _____ Скрипник Ірина Анатоліївна
(підпис) (науковий керівник)