

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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка 3D відеогри "Rogue Decay" на рушії Unity в жанрі roguelike»

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

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

_____ Денис ТАНЧУК
(підпис)

Виконав: здобувач вищої освіти групи ПД-43

_____ Денис ТАНЧУК

Керівник: _____ Микита ТРЕНЬОВ
асист. кафедри ІІЗ

Рецензент: _____

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ Ірина ЗАМРІЙ

« ____ » _____ 2026 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Танчуку Денису Ігоровичу

1. Тема кваліфікаційної роботи: «Розробка 3D відеогри "Rogue Desay" на рушії Unity в жанрі roguelike»

керівник кваліфікаційної роботи асистент кафедри ІПЗ Микита ТРЕНЬОВ,
затверджені наказом Державного університету інформаційно-комунікаційних
технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про жанри FPS і roguelike, матеріали з процедурної генерації рівнів та поведінки ігрових об'єктів, а також технічна документація Unity.

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

1. Аналіз предметної галузі та існуючих аналогів відеоігор у жанрі roguelike.

2. Визначення функціональних та нефункціональних вимог для розробки відеогри.

3. Аналіз існуючих рушіїв та програмного забезпечення для створення відеоігор, графіки та аудіосупроводу.

4. Реалізація та опис розробленої відеогри у жанрах FPS та roguelike

5. Тестування застосунку.

5. Перелік графічного матеріалу: *презентація*

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

6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Огляд існуючих рушіїв та методів для створення відеогри у жанрах FPS та rougelike	08.03-14.03.2026	
4	Проектування відеогри	15.03-22.03.2026	
5	Програмна реалізація відеогри	23.03-20.04.2026	
6	Тестування застосунку	21.04-24.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	25.04-30.04.2026	
8	Розробка демонстраційних матеріалів	01.05-10.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач вищої освіти _____
(підпис)

Денис ТАНЧУК

Керівник
кваліфікаційної роботи _____
(підпис)

Микита ТРЕНЬОВ

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 56 стор., 2 табл., 21 рис., 21 джерел.

Мета роботи – підвищення рівня зацікавленості гравця у процедурно згенерованих рівнях шляхом оптимізації FPS механік та впровадження динамічних моделей взаємодії, що створюють нетиповий для жанру ігровий досвід. Зокрема імплементація систем динамічного руху гравця, моделі стрілецької взаємодії, алгоритми послідовної генерації рівнів, механізмів прогресії та підсилення персонажа, а також технологічні засоби реалізації цих систем у ігровому рушії Unity.

Об'єкт дослідження – процеси формування ігрового досвіду в FPS грі із послідовною генерацією кімнат.

Предмет дослідження – механіки, методи та технології формування ігрового досвіду в FPS грі з покімнатною roguelike структурою.

Короткий зміст роботи: У роботі проаналізовано жанрові особливості FPS та roguelike, визначено ключові механіки, що формують ігровий процес у межах поєднання цих напрямів. Проведено огляд сучасних рушіїв та інструментів для створення відеоігор, на основі якого обґрунтовано вибір Unity як основного середовища розробки. Реалізовано основні системи гри: керування персонажем, стрільбу, процедурну генерацію рівнів, поведінку ворогів і босів, карткову систему покращень та інтерфейс користувача. Проведено тестування функціональних можливостей, перевірено стабільність роботи гри та взаємодію між її компонентами.

Сферою використання застосунку є індустрія комп'ютерних ігор та цифрових розваг.

КЛЮЧОВІ СЛОВА: FPS, ROUGELIKE, UNITY, ГЕНЕРАЦІЯ РІВНІВ, СИСТЕМА ЗБРОЇ, ВОРОГИ, КАРТКОВІ ПОКРАЩЕННЯ ХАРАКТЕРИСТИК.

ЗМІСТ

ВСТУП.....	9
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ.....	11
1.1 Розвиток жанрів FPS та roguelike у відеоігровій індустрії	11
1.2 Аналіз ключових механік, притаманних FPS та roguelike відеоіграм.....	12
1.3 Аналіз сучасних аналогів	12
1.3.1 Dead Cells	13
1.3.2 Enter the Gungeon	14
1.3.3 Hades.....	15
1.4 Формування вимог на основі аналізу аналогів.....	17
2 АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБЛЕННЯ 3D ВІДЕОІГОР	19
2.1 Вибір програмного забезпечення	19
2.1.1 Рушій Unity	19
2.1.2 Рушій Unreal Engine.....	21
2.1.3 Рушій Godot	22
2.2 Обране програмне забезпечення для створення відеогри.....	23
2.2.1 Unity.....	24
2.2.2 Blender	24
2.2.3 Reaper	25
2.2.4 C#	25
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ВІДЕОГРИ.....	26
3.1 Загальна архітектура гри	26
3.1.1 Архітектурні рівні та їх взаємодія.....	26
3.1.2 Внутрішня структура підсистем та їх роль у формуванні ігрового процесу	28
3.2 Реалізація механік відеогри	30
3.2.1 Рух та керування персонажем.....	31
3.2.2 Збройова система	33
3.2.3 Генерація рівня.....	36
3.2.4 Система ворогів.....	39
3.2.5 Система босів	42

3.2.6 Карткова система покращень	43
3.3 Проектування даних.....	46
3.4 Інтерфейс користувача	51
3.5 Оптимізація гри	55
4 ТЕСТУВАННЯ.....	57
4.1 Тестові сценарії	58
ВИСНОВКИ.....	63
ПЕРЕЛІК ПОСИЛАНЬ	65
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ	67
ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ.....	77

ВСТУП

Актуальність: відеоігри на сьогодні займають високу позицію у індустрії розваг, поступово перетворившись із нішевого хобі на масову культуру. Вплив відеоігор охоплює різні вікові категорії, а сучасні технології дозволяють створювати продукт, який конкурує за рівнем занурення з кіно. Стрімкий розвиток програмного забезпечення та поява доступних інструментів для розробки та поширення продукту сприяє тому, що ігрова індустрія стала одним із найшвидше зростаючих сегментів ринку. Особливо популярними сьогодні являються ігри жанру roguelike, які поєднують в собі варіативність проходження та можливість комбінації з іншими жанрами. З огляду на це, особливої популярності набули ігри жанру roguelike, які поєднують варіативність проходження та здатність ідеально комбінуватися з іншими жанрами.

Особливу увагу привертає комбінація FPS та roguelike. Поєднання динамічного шутеру з механіками, притаманними roguelike, створює унікальну синергію, що забезпечує високу реіграбельність і підтримує інтерес гравця.

Об'єктом дослідження є процеси формування ігрового досвіду в FPS грі із послідовною генерацією кімнат.

Предметом дослідження є механіки, методи та технології формування ігрового досвіду в FPS грі з покімнатною roguelike структурою, зокрема системи динамічного руху гравця, моделі стрілецької взаємодії, алгоритми послідовної генерації кімнат, механізми прогресії та підсилення персонажа, а також технологічні засоби реалізації цих систем у ігровому рушії Unity.

Метою роботи є підвищення рівня зацікавленості гравця у послідовних процедурно згенерованих кімнат шляхом оптимізації FPS механік та впровадження динамічних моделей взаємодії, що створюють нетиповий для жанру ігровий досвід

Методи дослідження: спочатку було проаналізовано сучасні методи

створення відеоігор у жанрах FPS та roguelike, а також методи їх поєднання. Далі були сформовані функціональні та нефункціональні вимоги до продукту.

Наступним кроком стало дослідження інструментів розробки, яке здатне забезпечити швидку та ефективну розробку. Ігровим рушієм був обраний Unity, програмою для створення 3D моделей став Blender. Мова програмування – C#, аудіоредактор – Reaper.

Наукова новизна роботи полягає у розробці відеогри, що поєднує динамічний геймплей із процедурними механіками, характерними для roguelike ігор. У межах проєкту реалізовано покімнатну генерацію рівнів, модульна система зброї, яка дозволяє швидко створювати нові одиниці зброї та система покращень гравця за допомогою карткової системи.

Практична значущість результатів полягає у створенні продукту, що демонструє можливості поєднання FPS механік із покімнатною процедурною генерацією рівнів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Розвиток жанрів FPS та roguelike у відеоігровій індустрії

Відеоігри жанру FPS та roguelike суттєво вплинули на становлення сучасної ігрової індустрії. Перші шутери від першої особи з'явилися ще на початку 1990-х років, коли можливості тогочасних комп'ютерів були обмежені, але дозволяли створювати тривимірні середовища. Ключовими проектами, що визначили жанр, стали Wolfenstein 3D та Doom, розроблені невеликою студією id Software[1]. Саме завдяки їх відеоіграми були створені фундаментальні механіки FPS: швидкий темп та динамічна стрільба[2]. В майбутньому розвиток жанру був пов'язаний із вдосконаленням графічних рушіїв, появою мультиплеєру та розширенням можливості взаємодії з середовищем гри. Вже у 2000-х роках шутери від першої особи стали провідним жанром у відеоігровій індустрії, а такі ігри, як Half-Life 2, Halo та Call of Duty, продемонстрували потенціал жанру як для сюжетних історій, так і для кіберспортивної дисципліни. Дослідники відзначають, що саме FPS став одним із жанрів, який найбільше вплинув на розвиток технологій рендерингу, фізики та штучного інтелекту в іграх.

Паралельно з розвитком шутерів формувався абсолютно унікальний жанр roguelike, який бере свій початок від гри Rogue[3]. Ключовими механіками цієї гри стала процедурна генерація рівнів, випадковість подій та концепція «перманентної смерті», що робило кожне проходження не схожим на попереднє[4]. Довгий час жанр залишався доволі нішевим, але разом із розвитком інді сцени та цифрової дистрибуції жанр отримав нове життя. Такі проекти, як The Binding of Isaac, Spelunky та Dead Cells стали широко відомі у світі відеоігор, та своїм прикладом показали, що такий вузький жанр може бути надпопулярним[5].

1.2 Аналіз ключових механік, притаманних FPS та roguelike відеоіграм

Механіки, що формують ігровий досвід у шутерах від першої особи, базуються на поєднанні динамічного руху та стрільби. Для відеоігор у жанрі FPS характерні: висока швидкість переміщення, точний контроль камери, відчуття від зброї та динамічна модель поведінки противників. Дослідники відзначають, що саме поєднання руху, стрільби та поведінкових моделей ворогів визначає темп гри та рівень залучення користувача[6]. У сучасних FPS важливу роль відіграють також системи колізій, фізика, анімаційні переходи та аудіовізуальні ефекти, які формують цілісне сприйняття бою та впливають на якість ігрового досвіду.

Roguelike ігри, на відміну від FPS, роблять акцент на варіативності та непередбачуваності. Їхні ключові механіки включають: процедурну генерацію рівнів, випадковість подій, систему ризику та винагороди, а також концепцію перманентної смерті, що стимулює гравця до стратегічного мислення та адаптації[4]. Важливою складовою є прогресія, як тимчасова (покращення в межах одного забігу), так і постійна (метапрогресія), що формує різні стилі гри та підвищує реіграбельність. Поєднання динаміки шутера з roguelike системами створює гібридний формат, у якому швидкі бойові рішення взаємодіють із процедурною непередбачуваністю, забезпечуючи унікальний ігровий досвід, що поєднує реактивність та стратегічну глибину.

1.3 Аналіз сучасних аналогів

Сучасні проєкти жанру roguelike та roguelite відіграють важливу роль у формуванні підходів до побудови варіативного ігрового процесу. Для аналізу було обрано три популярні та впливові ігри: Dead Cells, Enter the Gungeon та Hades[5]. Кожна з них реалізує власну інтерпретацію процедурної генерації, бойових механік та системи прогресії, що дозволяє порівняти різні геймдизайнерські рішення та визначити елементи, релевантні для розробки FPS-roguelike проєкту.

1.3.1 Dead Cells

Dead Cells - це 2D action-roguelite, який поєднує швидкий бій, процедурну генерацію рівнів та систему тимчасових і постійних покращень[7]. Гра вирізняється високою динамікою, чіткою читаемістю анімацій та акцентом на реактивності гравця. На рисунку 1.1 наведено приклад бойової сцени, що демонструє характерний темп гри та структуру рівня.



Рис.1.1 Бойова сцена з Dead Cells

У Dead Cells рівні формуються за модульним принципом: гра має набір заздалегідь створених сегментів, які комбінуються процедурно. Це дозволяє зберігати баланс між варіативністю та контрольованою якістю дизайну. Бойова система базується на швидких атаках, ухиленнях та використанні зброї з різними властивостями. Важливим елементом є система «клітин», ресурсу, який гравець збирає під час забігу та витрачає на постійні покращення після смерті.

Попри сильні сторони, Dead Cells має низку обмежень, притаманних 2D платформерам. Через двовимірну площину взаємодія з простором є доволі лінійною, а вертикальність використовується переважно як допоміжний елемент, що зменшує глибину просторових рішень. Модульна структура рівнів забезпечує варіативність, але водночас створює відчуття повторюваності після тривалого

ігрового досвіду. Крім того, швидкий темп бою та акцент на рефlekсах інколи обмежують стратегічність, роблячи геймплей більш реактивним, ніж тактичним.

1.3.2 Enter the Gungeon

Enter the Gungeon - це топ-даун roguelike шутер, який робить акцент на стрільбі, ухиленнях та покімнатній структурі рівнів[8]. На рисунку 1.2 наведено приклад типової кімнати, що демонструє bullet hell характер бою.



Рис. 1.2 Ігрова кімната з Enter the Gungeon

Гра використовує покімнатну генерацію, де кожна кімната є окремим модулем із власними ворогами, пастками та об'єктами. Бойова система побудована навколо ухилення («dodge roll»), яке дозволяє гравцеві уникати великої кількості куль. Це створює унікальний темп бою, де гравець постійно перебуває в русі, аналізує патерни атак та шукає можливості для контратаки. Велика кількість зброї, понад 300 одиниць, що забезпечує значну варіативність стилів гри, оскільки кожна зброя має унікальні властивості.

Прогресія в Enter the Gungeon є повністю тимчасовою: усі покращення, зброя та предмети зникають після смерті персонажа. Це підсилює акцент на майстерності

гравця та вмінні адаптуватися до випадкових умов кожного забігу. Покімнатна структура рівнів робить гру близькою до класичних roguelike, але з акцентом на динамічний bullet hell бій. Такий підхід дозволяє створювати високий рівень напруги та непередбачуваності, що є важливими характеристиками жанру.

Enter the Gungeon має низку обмежень, пов'язаних із топ-даун перспективою та структурою арен. Через плоский вигляд зверху взаємодія з простором є доволі обмеженою, а бої відбуваються переважно на горизонтальних площинах без можливості використання вертикальності чи складних тривимірних маневрів. Покімнатна генерація забезпечує варіативність, але водночас робить структуру рівнів передбачуваною після тривалого ігрового досвіду, оскільки кімнати мають схожі форми та сценарії. Крім того, bullet hell характер бою може бути надмірно хаотичним для нових гравців, а велика кількість зброї не завжди компенсує обмеженість просторової взаємодії.

1.3.3 Hades

Hades - це action roguelite з ізометричним видом, який поєднує динамічний бій, сюжетну прогресію та систему випадкових покращень («boons»)[9]. На рисунку 1.3 наведено приклад бойової сцени з гри.

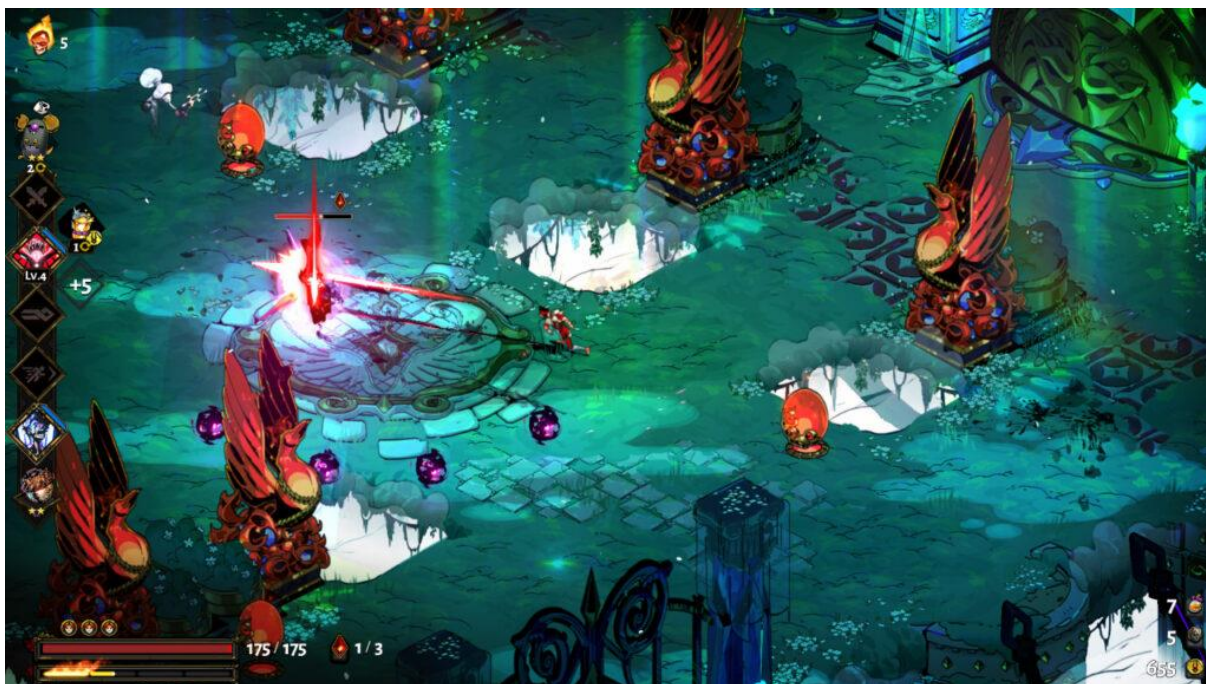


Рис. 1.3 Бойова сцена з Hades

Гра використовує покімнатну структуру, де кожна кімната має власний набір ворогів, винагород та умов. Однією з ключових особливостей є система «благословень» від богів Олімпу, які випадково випадають під час проходження та формують унікальні збірки здібностей. Це створює значну варіативність і дозволяє гравцю експериментувати з різними стилями бою. На відміну від багатьох roguelike, Hades активно використовує сюжетну прогресію, яка розкривається через багаторазові спроби гравця, що робить гру більш доступною для широкої аудиторії.

Бойова система Hades поєднує ближній та дальній бій, ухилення та використання спеціальних здібностей. Важливою частиною дизайну є чітка читаємість анімацій та візуальних ефектів, що дозволяє гравцеві швидко реагувати на загрози. Прогресія включає як тимчасові покращення під час забігу, так і постійні елементи, що відкриваються між проходженнями. Такий підхід забезпечує баланс між складністю та відчуттям розвитку, що робить гру однією з найуспішніших у жанрі.

Разом із тим ізометрична перспектива обмежує глибину просторової взаємодії, оскільки бої відбуваються переважно на плоских аренах без можливості повноцінного використання вертикальності чи складних тривимірних маневрів. Покімнатна структура базується на повторюваних шаблонах, що з часом може знижувати відчуття новизни. Додатково акцент на сюжетній прогресії частково зменшує рівень непередбачуваності, властивий класичним roguelike, роблячи загальний ігровий процес більш керованим і менш хаотичним.

Задля узагальнення результату проведеного аналізу та чітко окреслити спільні риси й відмінності між розглянутими іграми, нижче наведено зведену порівняльну таблицю. Вона систематизує ключові характеристики кожного проекту, включаючи тип генерації рівнів, бойову динаміку, структуру простору, особливості прогресії та унікальні дизайнерські рішення. Такий формат дозволяє наочно оцінити сильні та слабкі сторони кожного з аналогів, а також визначити, які підходи можуть бути адаптовані або вдосконалені під час розробки власного ігрового проекту.

Таблиця 1.1

Зведена таблиця порівняння ігрових аналогів

Характеристика	Dead Cells	Enter the Gungeon	Hades
Взаємоді з простором	2D площина	Top-down	Ізометрія
Темп та стиль бою	Дуже швидкий, реактивний	Bullet hell	Комбо-орієнтований
Тип генерації рівнів	Модульна, сегментована	Покімнатна, модульна	Покімнатна, фіксовані шаблони
Прогресія	Тимчасові та постійні покращення	Лише тимчасові	Тимчасові та постійні покращення
Глибина взаємодії з простором	Лінійна 2D	Переважно горизонтальна	Обмежена перспективою
Ігровий стиль	реактивний	реактивний з аналізом патернів	збалансований

1.4 Формування вимог на основі аналізу аналогів

Проведений аналіз ігрових проєктів Dead Cells, Enter the Gungeon та Hades дозволив визначити ключові підходи до побудови бойових систем, структури рівнів та прогресії, які формують сучасний досвід у жанрі roguelike. Кожна з розглянутих ігор демонструє власний баланс між динамікою, варіативністю та доступністю, що дає змогу окреслити як ефективні дизайнерські рішення, так і обмеження, притаманні різним перспективам та механікам.

Отримані результати показали, що успішні проєкти поєднують швидкий темп бою з високою читаємістю подій, забезпечують достатню варіативність рівнів завдяки модульній або покімнатній генерації та використовують системи прогресії, які підтримують інтерес гравця протягом багатьох забігів. Водночас аналіз виявив

і типові недоліки: обмеженість просторової взаємодії у 2D проєктах, повторюваність шаблонів рівнів та залежність від реактивного стилю гри[10].

Узагальнення цих спостережень дозволило сформулювати вимоги до власного ігрового проєкту таким чином, щоб зберегти сильні сторони жанру та уникнути поширених недоліків. Нижче наведено перелік функціональних і нефункціональних вимог, сформованих на основі проведеного аналізу.

Функціональні вимоги:

1. Забезпечення динамічного переміщення гравця в межах згенерованих покімнатних локацій.
2. Реалізація базових та розширених FPS-механік, зокрема системи стрільби та взаємодії зі зброєю.
3. Впровадження системи карткової прокачки характеристик, що впливає на ігровий стиль та прогресію.
4. Розробка механізмів збереження та відновлення стану гри.
5. Створення системи випадання зброї з урахуванням її рідкості.
6. Реалізація механізму поступового підвищення складності після проходження кожного рівня.

Нефункціональні вимоги:

1. Підтримка ОС не менше Windows 10.
2. Стабільність ігрового процесу з частотою кадрів 60-100к/с.
- 3.Мінімальні системні вимоги: Процесор: Intel CORE I3-8100/AMD Ryzen 3 3300 X RAM: 8 гб Відеокарта: NVIDIA GeForce GTX 1050/AMD Radeon RX 570.
- 4.Рекомендовані системні вимоги: Процесор: Intel CORE I5-8700/AMD Ryzen 5 3600 X RAM: 16 гб Відеокарта: NVIDIA GeForce GTX 1060Ti/ AMD Radeon RX 5700 XT.

2 АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБЛЕННЯ 3D ВІДЕОІГОР

2.1 Вибір програмного забезпечення

Розроблення сучасних відеоігор неможливе без використання спеціалізованих інструментальних засобів, які забезпечують створення графіки, фізики, анімацій, логіки та інтерактивних систем. На відміну від класичних програмних продуктів, ігри вимагають комплексного підходу, де поєднуються технічні, художні та дизайнерські рішення. Саме тому вибір рушія є одним із ключових етапів під час планування будь-якого ігрового проєкту. Від можливостей рушія залежить не лише якість кінцевого продукту, а й швидкість розроблення, зручність роботи команди, продуктивність гри та потенціал її масштабування.

У межах бакалаврської роботи розглядається створення тривимірної гри жанру FPS roguelike, що поєднує динамічний бойовий процес із процедурною генерацією рівнів. Для реалізації такого проєкту необхідно проаналізувати існуючі програмні засоби, які дозволяють ефективно працювати з 3D графікою, системами фізики, поведінкою ворогів, а також підтримують механіки випадкової генерації. На ринку ігрових рушіїв найбільш поширеними є Unity, Unreal Engine та Godot, саме вони формують основу сучасної індустрії та використовуються як у великих студіях, так і в інді розробці[11].

2.1.1 Рушій Unity

Unity є одним із найпоширеніших рушіїв для створення 2D та 3D відеоігор, що активно використовується як інді-розробниками, так і великими студіями[12]. Його популярність пояснюється універсальністю, зручністю інтерфейсу та широкою екосистемою інструментів. Рушій підтримує компонентну архітектуру, яка дозволяє швидко створювати ігрові об'єкти та керувати їхньою поведінкою за

допомогою C#. Це робить Unity особливо привабливим для проєктів, де важлива гнучкість та можливість швидкого прототипування.

Unity добре підходить для створення FPS ігор завдяки наявності готових систем фізики, контролерів персонажа, анімаційних інструментів та можливості легко інтегрувати сторонні пакети з Asset Store. Для реалізації roguelike механік рушій також є зручним: процедурна генерація рівнів може бути реалізована через префаби та скрипти, які формують структуру рівня.

До переваг Unity належать: простота освоєння, велика кількість навчальних матеріалів, активна спільнота, підтримка багатьох платформ та можливість швидкого створення прототипів. Недоліками є те, що у великих сценах продуктивність може знижуватися без додаткової оптимізації, а графічні можливості поступаються більш потужним рушіям. На рисунку 2.1 наведено сцену рушія Unity

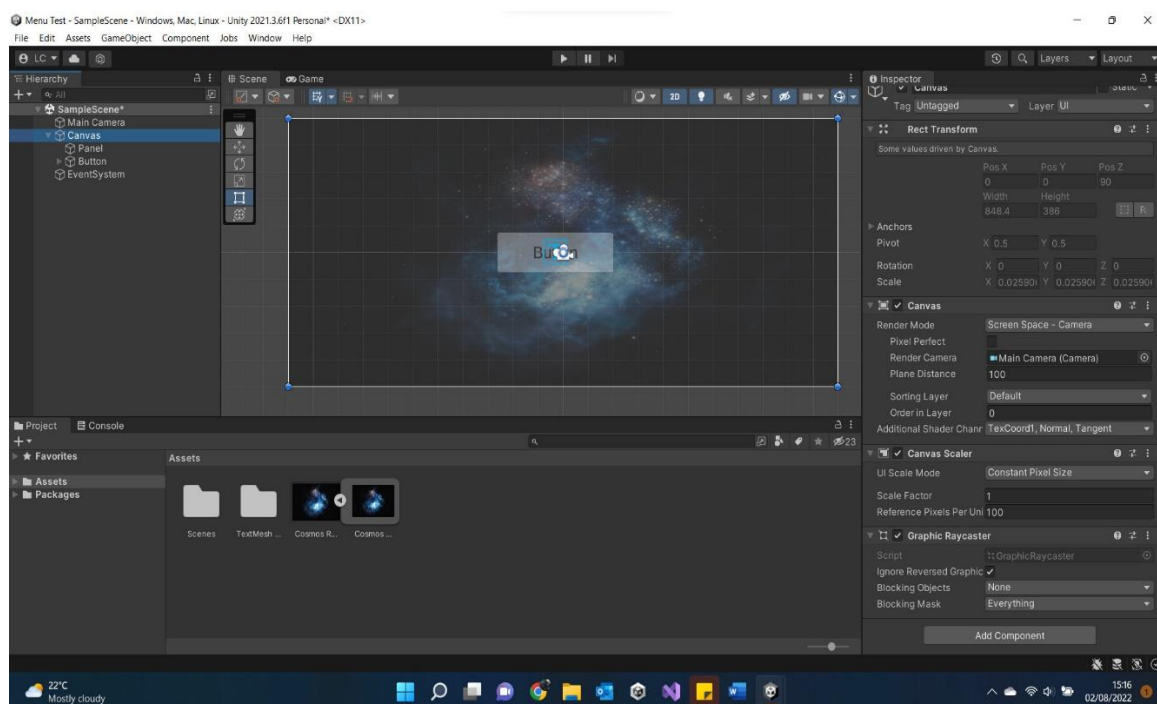


Рис 2.1 Сцена в рушії Unity

2.1.2 Рушій Unreal Engine

Unreal Engine - це високопродуктивний рушій, який широко використовується у великих студіях для створення AAA проєктів[13]. Його ключовою перевагою є потужний графічний рендеринг, що дозволяє створювати реалістичні сцени, складні ефекти освітлення та деталізовані моделі. Unreal Engine підтримує як C++, так і Blueprints - візуальну систему програмування, яка дозволяє створювати складні механіки без написання коду та дозволяє обґрунтувати вибір інструментального засобу для реалізації проєкту.

Для FPS-ігор рушій пропонує готові шаблони, системи обробки зброї, анімаційні графи та інструменти для роботи з поведінкою ворогів. Це робить Unreal Engine одним із найкращих варіантів для проєктів, де важлива графічна якість та масштабність. Проте рушій має високі вимоги до апаратного забезпечення, а його освоєння потребує більше часу порівняно з Unity.

Процедурна генерація в Unreal Engine можлива, але її реалізація складніша через необхідність глибшого розуміння внутрішньої архітектури рушія. Незважаючи на це, рушій дозволяє створювати дуже якісні та масштабні проєкти. На рисунку 2.2 наведено екранна форма рушія Unreal Engine

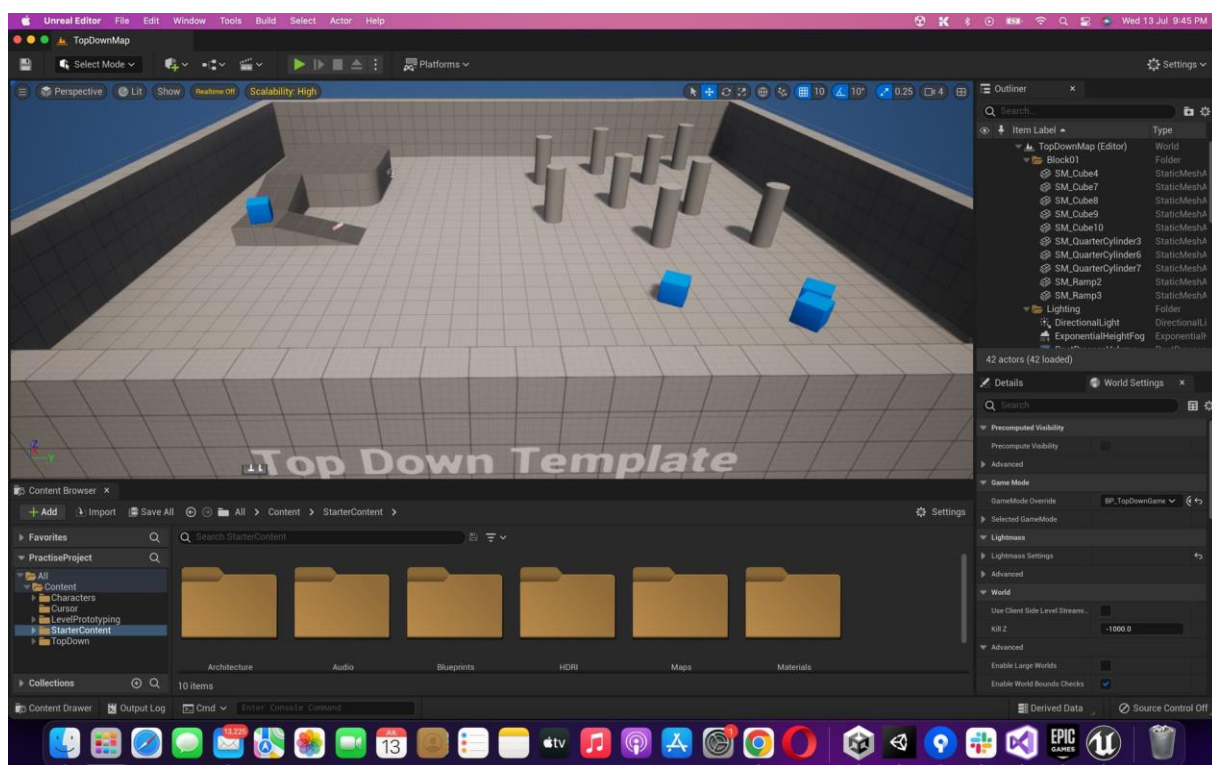


Рис 2.2 Екранна форма рушія Unreal Engine

2.1.3 Рушій Godot

Godot - це відкритий рушій, який активно розвивається та привертає увагу інди розробників завдяки простоті, легкості та повністю безкоштовній ліцензії[14]. Він використовує власну мову GDScript, яка нагадує Python, а також підтримує C#. Godot має зручну сцену систему, що дозволяє швидко створювати структуру рівнів, об'єктів та логіки.

Рушій добре підходить для невеликих 2D проєктів, але також підтримує 3D розробку. Проте у складних 3D-FPS-проєктах Godot поки що поступається Unity та Unreal Engine. Його система фізики менш оптимізована, а графічні можливості обмежені. Процедурна генерація можлива, але потребує більше ручної роботи через меншу кількість готових інструментів.

Переваги Godot: простота, легкість, відкритий код, низькі системні вимоги, швидке освоєння. Недоліки: слабша 3D-графіка, менша спільнота, обмежені можливості для складних FPS проєктів. На рисунку 2.3 наведено стандартну сцену з рушія Godot

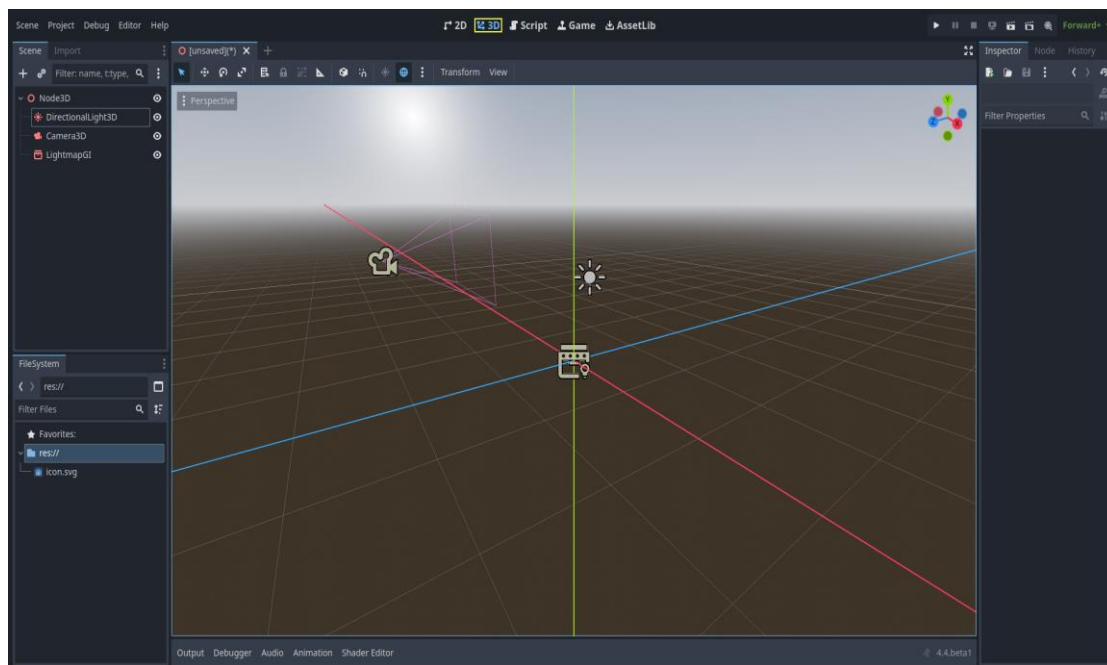


Рис 2.3 Сцена з рушія Godot

Таблиця 3.1

Порівняльна таблиця рушіїв

Характеристика	Unity	Unreal Engine	Godot
Мова програмування	C#	C++, Blueprints	GScript, C#
Графічні можливості	Середні	Дуже високі	Базові/середні
Продуктивність у 3D	Висока, потребує оптимізації	Дуже висока	Середня
Процедурна генерація	Зручна, гнучка	Можлива	Можлива, обмежена
Поріг входу	Низький	Високий	Низький
Підходить для FPS	Так	Так	Частково
Ліцензія	Freemium	Безкоштовно до певного доходу	Повністю безкоштовно
Спільнота	Дуже велика	Велика	Середня

2.2 Обране програмне забезпечення для створення відеогри

Розроблення сучасної відеогри потребує використання комплексу інструментальних засобів, які охоплюють різні аспекти створення цифрового продукту: графіку, звук, логіку, фізику, анімацію та інтерактивність. На відміну від класичних програмних систем, ігрові проєкти поєднують технічні та творчі компоненти, тому вибір технологій безпосередньо впливає на якість кінцевого результату. У межах бакалаврської роботи для створення гри жанру FPS roguelike використовуються три основні інструменти: рушій Unity для реалізації ігрової логіки та інтерактивності, Blender для створення 3D моделей та Reaper для обробки звукового супроводу. Крім того, важливу роль відіграє мова програмування C#, яка забезпечує роботу всіх внутрішніх механік гри. Нижче наведено огляд цих засобів, їхніх можливостей та ролі у створенні проєкту.

2.2.1 Unity

Unity є основним рушієм, на якому реалізовано гру. Це універсальна платформа для створення 2D та 3D-проектів, що поєднує інструменти для роботи з графікою, фізикою, анімаціями, звуком та логікою. Рушій використовує компонентну архітектуру, яка дозволяє будувати складні системи шляхом комбінування окремих модулів. Для програмування застосовується мова C#, що забезпечує високу гнучкість і контроль над поведінкою об'єктів [15].

Unity особливо добре підходить для FPS проектів завдяки наявності готових інструментів для роботи з камерою від першої особи, системами колізій, фізикою та анімаційними графами. У межах розроблення гри Unity використовується для реалізації руху персонажа, стрільби, поведінки ворогів, системи зброї, генерації кімнат та логіки переходів між ними. Рушій також дозволяє створювати процедурну генерацію рівнів, що є ключовою частиною жанру roguelike.

2.2.2 Blender

Blender використовується для створення 3D-моделей, які застосовуються у грі: зброї, ворогів, кімнат, предметів та декоративних елементів. Це професійний інструмент із відкритим кодом, який підтримує повний цикл створення 3D-контенту: моделювання, UV розгортку, текстурування, скульптинг, анімацію та рендеринг [16].

У межах проекту Blender дозволяє створювати низькополігональні моделі, оптимізовані для реального часу, що є важливим для FPS ігор. Завдяки підтримці експорту у формати FBX та OBJ моделі легко інтегруються в Unity. Blender також використовується для створення анімацій ворогів та об'єктів, які потім імпортуються в рушій.

Переваги Blender: безкоштовність, широкий набір інструментів, активна спільнота та можливість створювати як прості, так і складні моделі. Недоліком є складність освоєння для новачків через велику кількість функцій.

2.2.3 Reaper

Reaper є цифровою звуковою робочою станцією (DAW), яка використовується для створення та обробки звукових ефектів у грі. Програма дозволяє записувати, редагувати та міксувати аудіо, а також працювати з VST-плагінами, що розширюють можливості обробки звуку [17].

У межах розроблення гри Reaper застосовується для створення звуків пострілів, кроків, ударів, взаємодій із предметами, звуків ворогів та атмосферних ефектів. Звуковий супровід є важливою частиною FPS проєктів, оскільки він впливає на відчуття динаміки, небезпеки та просторової орієнтації. Reaper дозволяє створювати якісні звукові ефекти з мінімальними ресурсами.

Переваги Reaper: доступна ліцензія, висока гнучкість, швидкість роботи та можливість налаштування інтерфейсу. Недоліком є те, що програма потребує певного досвіду роботи зі звуком.

2.2.4 C#

C# є основною мовою програмування, яка використовується в Unity для реалізації всієї ігрової логіки. Мова поєднує об'єктно-орієнтований підхід, високу читабельність та продуктивність, що робить її зручною для створення складних систем взаємодії між об'єктами [18].

У межах розроблення гри C# використовується для створення механіки стрільби, руху персонажа, поведінки ворогів, системи зброї, підбору предметів, роботи інвентаря та процедурної генерації кімнат. Мова підтримує подієву модель програмування, корутини, делегати та інші інструменти, які дозволяють ефективно керувати процесами в реальному часі.

Переваги C#: висока продуктивність, зручний синтаксис, велика кількість бібліотек, активна спільнота та повна інтеграція з Unity. Недоліком є те, що мова орієнтована переважно на екосистему .NET, але для Unity це не є обмеженням.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ВІДЕОГРИ

3.1 Загальна архітектура гри

Архітектура розробленої гри побудована за принципами модульності, розділення обов'язків та чіткої взаємодії між компонентами. Такий підхід дозволяє забезпечити масштабованість, гнучкість та простоту подальшої підтримки проєкту. Уся система складається з кількох взаємопов'язаних рівнів, кожен з яких виконує власну роль у формуванні ігрового процесу. В основі архітектури лежить ідея розподілу логіки між окремими підсистемами: управління ігровим циклом, поведінка гравця, система зброї, штучний інтелект ворогів, генерація рівнів, система босів, механіка покращень, інтерфейс користувача та аудіосистема. Кожен із цих компонентів функціонує автономно, але водночас тісно взаємодіє з іншими, утворюючи цілісну структуру гри.

3.1.1 Архітектурні рівні та їх взаємодія

На найвищому рівні архітектура гри може бути представлена як система, що складається з трьох логічних шарів: рівня керування ігровим циклом, рівня ігрової логіки та рівня представлення. Рівень керування ігровим циклом відповідає за зміну сцен, запуск нового забігу, перехід між рівнями та обробку поразки. Цей шар реалізовано через центральний компонент RunManager, який виступає координатором між усіма іншими підсистемами. Саме він визначає, коли гравець має бути перенесений у хаб, коли починається новий забіг, коли потрібно скинути характеристики персонажа та які сцени слід завантажити після перемоги над босом. У цьому ж шарі працюють допоміжні елементи, такі як телепорти між рівнями, тригери входу в забіг та кнопки перезапуску гри.

Другий шар, рівень ігрової логіки, є найбільш насиченим і включає поведінку гравця, систему зброї, логіку ворогів, генерацію кімнат, систему босів та механіку покращень. Усі ці компоненти працюють у межах однієї сцени, але виконують різні

функції. Наприклад, PlayerStats відповідає за здоров'я, досвід, рівень та множники характеристик, тоді як PlayerMovement реалізує фізичну поведінку персонажа: рух, стрибок, даш, взаємодію з поверхнями та звуковий супровід. Система зброї, побудована на основі ScriptableObject, дозволяє легко додавати нові види озброєння, змінювати їх характеристики та керувати анімаціями стрільби. Вороги, у свою чергу, реалізовані через ієрархію класів EnemyBase, HumanEnemyBase та Zombie-класи, що забезпечує різноманітність поведінкових патернів: від ближнього бою до складних burst-атак і стрільби з розльотом. Генерація рівнів здійснюється через систему кімнат, де кожна кімната має власні точки спавну ворогів, двері, логіку хвиль та шанс появи скрині або телепорту до босса. Боси представлені окремою підсистемою, що включає унікальні механіки, фази бою, пулінг снарядів та спеціальний інтерфейс здоров'я.

Третій шар, рівень представлення, відповідає за відображення інформації гравцю. До нього належать елементи інтерфейсу: панель здоров'я, індикатор досвіду, підказки взаємодії, іконка зброї, кількість патронів, хітмаркер та панель вибору покращень. Усі UI елементи об'єднані в окремий UIRoot, який переноситься між сценами та гарантує стабільність інтерфейсу незалежно від переходів. На цьому ж рівні працює аудіосистема, що відповідає за відтворення музики, звуків кроків, стрільби, перезарядки, ударів та інших ефектів, які формують атмосферу гри. Для кращого розуміння взаємодії гравця з основними підсистемами гри доцільно представити загальну модель поведінки користувача у вигляді діаграми варіантів використання.

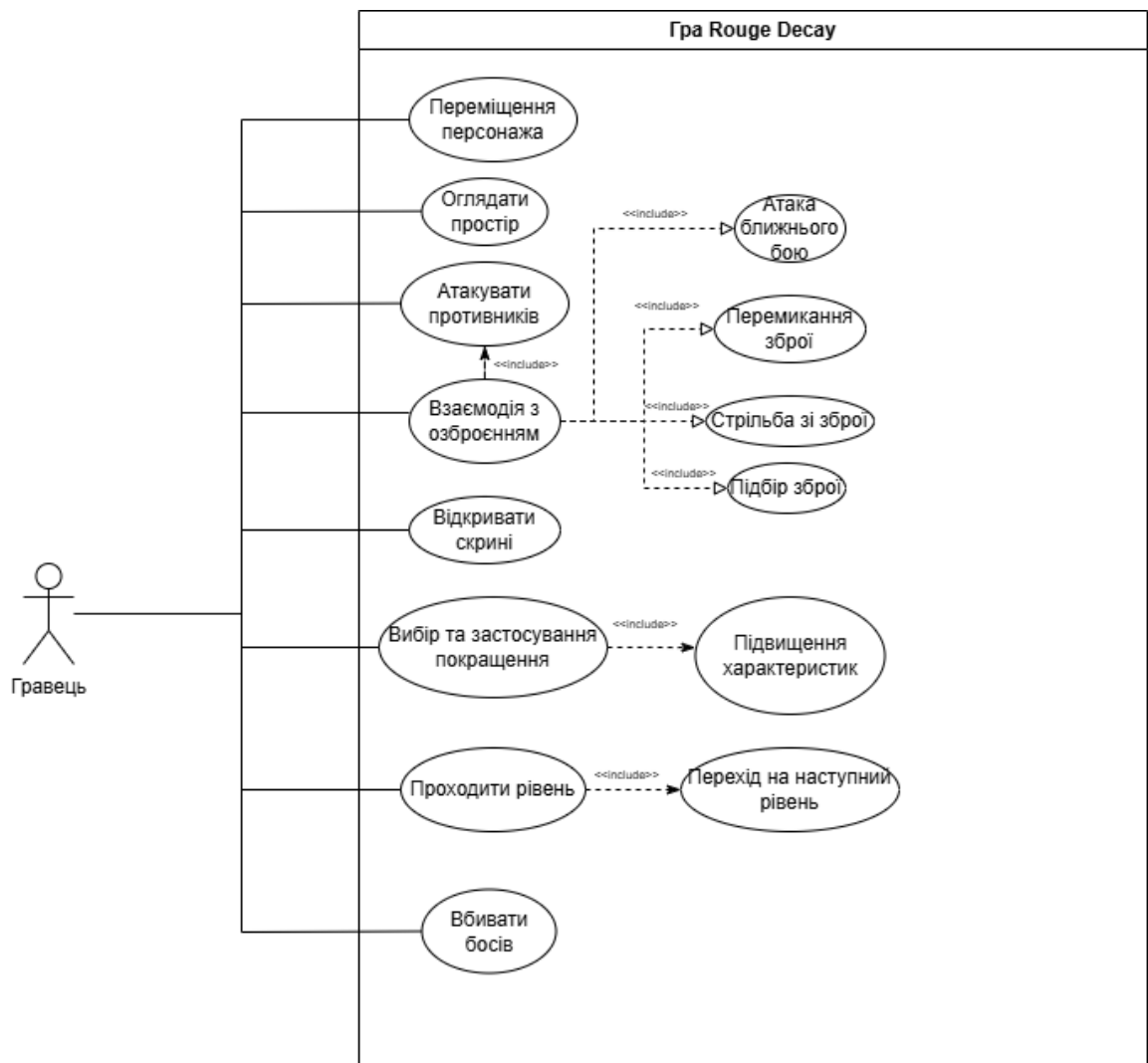


Рис. 3.1 Діаграма варіантів використання

3.1.2 Внутрішня структура підсистем та їх роль у формуванні ігрового процесу

Кожна підсистема гри має власну внутрішню структуру та набір відповідальностей. Наприклад, система гравця побудована за принципом розділення логіки: `PlayerStats` відповідає за числові параметри та їх зміну, `PlayerMovement` за фізичну поведінку, `CameraMovement` за керування камерою та віддачею, а `PlayerInteract` за взаємодію з об'єктами. Такий підхід дозволяє уникнути надмірної концентрації логіки в одному класі та забезпечує легкість розширення функціоналу.

Система зброї реалізована через абстрактний клас `WeaponBase`, від якого успадковуються `GunWeapon` та `MeleeWeapon`. Це дозволяє використовувати

єдиний інтерфейс для стрільби, перезарядки та анімацій, але водночас надає можливість реалізувати унікальну поведінку для різних типів зброї. `WeaponController` виступає центральним елементом, що керує перемиканням зброї, обробкою натискань кнопок, recoil-ефектами та взаємодією з UI. Завдяки використанню `ScriptableObject` для зберігання параметрів зброї, додавання нового виду озброєння зводиться до створення нового об'єкта даних та префабу.

Поведінка ворогів реалізована через ієрархію класів, що дозволяє легко розширювати набір противників. `HumanEnemyBase` забезпечує базову логіку для ворогів-людей: використання `NavMeshAgent`, перевірку видимості гравця, рух до цілі та виконання атак. На його основі побудовані `RaiderMelee`, `RaiderPistol`, `MilitaryAK`, `MilitaryShotgun` та `BurstShooter` — кожен із власним патерном поведінки. Зомбі реалізовані окремо, оскільки мають іншу модель поведінки: вони використовують систему `separation` для уникнення скупчення, мають різні типи атак та можуть стріляти снарядами, що обробляються через `ZombieBulletPool`. Така архітектура дозволяє легко додавати нових ворогів, змінювати їх характеристики та створювати унікальні сценарії бою.

Генерація рівнів побудована на системі кімнат, де кожна кімната є автономним модулем із власною логікою. `Room` відповідає за запуск хвиль ворогів, закриття дверей, появу скринь та телепортів. `RoomSpawner` генерує нові кімнати на основі напрямку, а `RoomManager` контролює зайняті позиції, щоб уникнути накладання кімнат одна на одну. `RoomDoorController` аналізує сусідні кімнати та автоматично відкриває або закриває двері, забезпечуючи логічну структуру рівня. Такий підхід дозволяє створювати процедурно згенеровані рівні, які щоразу виглядають по-іншому, але зберігають логічну цілісність.

Система босів є окремим модулем, оскільки бій із босом має іншу структуру, ніж звичайні кімнати. `BossRoom` активує боса при вході гравця, відображає панель здоров'я та після перемоги створює телепорт на наступний рівень. Кожен бос реалізує власну поведінку: `MilitaryTurretBoss` має систему перегріву та burst-стрільбу, `ArmoredBanditBoss` поєднує ближній бій із призовом мінійонів, а

FatZombieBoss має три фази з різними патернами атак. Усі боси використовують пулінг снарядів через BossBulletPool, що забезпечує оптимальну продуктивність.

Система покращень реалізована через UpgradeManager, який відповідає за вибір однієї з трьох карток після підвищення рівня. Картки зберігаються у вигляді ScriptableObject, що дозволяє легко додавати нові покращення та комбінувати їх у різних варіаціях. Після вибору картки її ефекти застосовуються до PlayerStats, змінюючи характеристики гравця на поточний забіг.

Інтерфейс користувача побудований таким чином, щоб забезпечити гравцю всю необхідну інформацію без перевантаження екрану. Панель здоров'я, індикатор досвіду, кількість патронів, іконка зброї, підказки взаємодії та хітмаркер працюють у реальному часі та оновлюються відповідно до подій у грі. Завдяки використанню UIRoot інтерфейс зберігається між сценами, що забезпечує стабільність та цілісність UI.

Окремим компонентом архітектури є аудіосистема, яка відповідає за відтворення музики та звукових ефектів. MusicManager автоматично перемикає треки, забезпечуючи різноманітність музичного супроводу, а звуки кроків, стрільби, ударів та взаємодій створюють атмосферу та підсилюють ігровий досвід.

Усі описані підсистеми утворюють цілісну архітектуру, де кожен компонент виконує власну роль, але водночас взаємодіє з іншими через чітко визначені точки. Така структура дозволяє легко розширювати гру, додавати нові види ворогів, зброї, кімнат або механік, не порушуючи загальної логіки. Для наглядності нижче наведено узагальнену діаграму архітектури гри, яка відображає основні модулі та зв'язки між ними.

3.2 Реалізація механік відеогри

У процесі створення гри було прийнято рішення використовувати модульну архітектуру, оскільки саме такий підхід дозволяє підтримувати чітку структурованість проєкту та забезпечує зручність подальшого розширення. Кожна ключова система гри була виділена в окремий модуль, який відповідає за власну

частину логіки і не залежить від внутрішньої реалізації інших компонентів. Це дало можливість уникнути надмірної кількості зв'язків між класами та зберегти зрозумілу ієрархію, що особливо важливо для проєктів, де одночасно працює багато взаємодіючих механік.

Модульна архітектура дозволила організувати роботу так, щоб кожна система виконувала свою роль незалежно, але при цьому могла взаємодіяти з іншими через чітко визначені точки доступу. Наприклад, система зброї працює окремо від системи ворогів, але обидві взаємодіють через механізм нанесення шкоди. Генерація рівня не залежить від конкретної поведінки ворогів, але саме вона визначає умови, у яких ці вороги з'являються. Система прокачки не втручається у внутрішню логіку руху чи стрільби, але впливає на їхні параметри, змінюючи стиль гри та загальну динаміку проходження.

Завдяки такому підходу вдалося створити гнучку структуру, у якій кожен модуль може розвиватися окремо, не порушуючи роботу інших частин проєкту. Це особливо важливо для ігор жанру FPS з елементами roguelike, де велика кількість механік повинна працювати узгоджено, а зміни в одній системі можуть впливати на загальний баланс. Модульність також спростила підтримку проєкту, оскільки будь-які зміни або доповнення можна вносити локально, не зачіпаючи інші механіки.

У наступних підрозділах буде детально розглянуто реалізацію основних ігрових механік, їхню внутрішню структуру та принципи взаємодії з іншими системами. Це дозволить показати, як саме модульний підхід вплинув на організацію проєкту та забезпечив стабільну роботу всіх елементів геймплею.

3.2.1 Рух та керування персонажем

Система руху та керування персонажем є однією з ключових складових ігрового процесу у проєкті, оскільки саме вона визначає те, наскільки комфортно гравець взаємодіє з ігровим середовищем. У жанрі FPS будь-які затримки, ривки або неприродні рухи одразу відчуються, тому реалізація цієї механіки вимагала особливої уваги. Основна мета полягала у створенні плавного, передбачуваного та

водночас динамічного керування, яке відповідало б загальному темпу гри та дозволяло гравцю швидко реагувати на події.

У проєкті використовується компонент `CharacterController`, який забезпечує стабільну взаємодію персонажа з поверхнями та дозволяє уникнути проблем, пов'язаних із фізичними колайдерами. Це рішення було обрано через його надійність та простоту налаштування, що особливо важливо у швидких іграх, де фізика не повинна створювати зайвих перешкод. Логіка руху реалізована у скрипті `PlayerMovement`, який відповідає за обробку вводу, розрахунок швидкості, перевірку дотику до землі, стрибки, прискорення та інші аспекти пересування.

Рух персонажа базується на зчитуванні горизонтальних та вертикальних осей, що дозволяє гравцю переміщатися у будь-якому напрямку відносно поточного положення камери. Це створює природне відчуття контролю, оскільки напрямок руху завжди відповідає тому, куди дивиться гравець. Швидкість пересування визначається характеристиками персонажа, які зберігаються у окремому модулі `PlayerStats`. Такий підхід дозволяє легко змінювати параметри руху через систему прокачки або зовнішні ефекти.

Окрему увагу було приділено реалізації стрибків та гравітації. Перевірка дотику до землі здійснюється через сферичний каст, що дозволяє точно визначати момент приземлення навіть на нерівних поверхнях. Це важливо для уникнення ситуацій, коли персонаж зависає у повітрі або не може стрибнути через некоректне визначення стану. Гравітація застосовується поступово, що забезпечує природну траєкторію падіння та дозволяє гравцю відчувати вагу персонажа.

Додатковим елементом руху є механіка ривка, яка дозволяє гравцю швидко змінювати позицію. Ривок реалізований через корутину, що тимчасово збільшує швидкість у напрямку руху або вперед, якщо гравець стоїть на місці. Такий підхід робить ривок інтуїтивним і корисним як у бою, так і під час пересування між кімнатами. Параметри ривка також залежать від характеристик персонажа, що дозволяє інтегрувати цю механіку у систему прогресії.

Важливою частиною керування є система звукового супроводу руху. У скрипті `PlayerMovement` реалізовано відтворення звуків кроків, стрибків,

приземлення та ривка. Це створює додатковий рівень занурення та допомагає гравцю краще відчувати взаємодію з поверхнями. Звуки кроків відтворюються з урахуванням швидкості руху, що робить їх природними та ненав'язливими.

Камера гравця керується окремим модулем CameraMovement, який відповідає за обробку рухів миші та формування огляду. Вертикальна вісь камери обмежена, щоб уникнути неприродних поворотів, а горизонтальний рух передається на тіло персонажа. Це створює класичну FPS-схему керування, яка є звичною для більшості гравців. Додатково реалізовано механіку віддачі, яка впливає на положення камери під час стрільби. Віддача поступово згладжується, що робить стрільбу більш відчутною та додає геймплею реалістичності.

Окремим елементом керування є система взаємодії з об'єктами, реалізована у скрипті PlayerInteract. Вона дозволяє гравцю взаємодіяти з предметами у межах певної дистанції. Перевірка здійснюється через Raycast, спрямований від камери вперед. Якщо промінь потрапляє у об'єкт, що реалізує інтерфейс Interactable, гравець отримує можливість виконати дію. Такий підхід дозволяє легко додавати нові типи інтерактивних об'єктів, оскільки достатньо лише реалізувати відповідний інтерфейс.

У сукупності всі ці елементи формують цілісну систему руху та керування персонажем, яка забезпечує плавний, динамічний та передбачуваний геймплей. Вона є основою взаємодії гравця з ігровим світом і визначає загальне відчуття від гри, тому її реалізація стала одним із найважливіших етапів розробки.

3.2.2 Збройова система

Збройова система у грі є однією з центральних механік, оскільки саме вона визначає динаміку бою, відчуття стрільби та загальний темп геймплею. У жанрі FPS будь-які недоліки у поведінці зброї одразу помітні гравцю, тому реалізація цієї системи вимагала ретельного підходу та продуманості на рівні архітектури. Основна ідея полягала у створенні гнучкої та розширюваної структури, яка дозволяє легко додавати нові види зброї, змінювати їхні характеристики та адаптувати поведінку під різні стилі гри.

У центрі всієї системи знаходиться абстрактний клас `WeaponBase`. Він визначає базовий набір властивостей та методів, які є спільними для будь-якого виду зброї. Саме тут зберігається інформація про кількість набоїв, стан перезарядки, час останнього пострілу та посилення на контролер зброї. Такий підхід дозволяє уникнути дублювання коду та забезпечує єдину точку входу для всіх похідних класів. Кожен конкретний тип зброї успадковує `WeaponBase` і реалізує власну логіку стрільби та взаємодії з анімаціями.

Усі характеристики зброї винесені у окремий `ScriptableObject` під назвою `WeaponData`. Це рішення дозволило відокремити дані від логіки та зробити систему максимально зручною для налаштування. У `WeaponData` зберігаються такі параметри, як шкода, скорострільність, розмір магазину, час перезарядки, тип стрільби, розкид, віддача, кількість дробинок для дробовиків, параметри черги для burst-режиму та інші властивості. Завдяки цьому нову зброю можна створити без написання коду, просто налаштувавши відповідний об'єкт у редакторі.

Основна логіка стрільби реалізована у класі `GunWeapon`. Він відповідає за обробку вводу, перевірку кулдаунів, зменшення кількості набоїв, відтворення анімацій та звуків, а також за виконання Raycast-перевірок для визначення попадання. Стрільба може здійснюватися у різних режимах, залежно від того, який `FireMode` вказано у `WeaponData`. Це може бути одиночний постріл, автоматичний режим або стрільба чергами. Для дробовиків реалізовано окрему логіку, яка створює кілька променів з випадковим відхиленням, що імітує розліт дробу.

Важливою частиною системи є механіка розкиду та віддачі. Після кожного пострілу зброя збільшує поточний розкид, що робить стрільбу менш точною при тривалому натисканні на кнопку. Розкид поступово відновлюється, якщо гравець перестав стріляти. Віддача реалізована через взаємодію з камерою, яка отримує зміщення по вертикалі та горизонталі. Це створює більш реалістичне відчуття стрільби та стимулює гравця контролювати зброю.

Перезарядка також має власну логіку. Після виклику методу `Reload` зброя переходить у стан перезарядки, відтворює відповідну анімацію та звук, а після завершення таймера відновлює кількість набоїв. Це дозволяє уникнути миттєвого

поповнення магазину та робить процес більш природним. Усі ці дії синхронізуються з UI через AmmoUI, який оновлює відображення кількості набоїв.

Окремим елементом системи є ближня зброя, реалізована у класі MeleeWeapon. Вона працює за іншим принципом, оскільки не використовує набої та не виконує Raycast у класичному розумінні. Замість цього ближня атака здійснюється через RaycastAll, який перевіряє наявність цілей у певному радіусі перед гравцем. Механіка ближнього бою має власні кулдауни, анімації та звукові ефекти. Це дозволяє створити окремий стиль гри, який відрізняється від стрільби, але при цьому органічно вписується у загальну систему.

Усі види зброї контролюються через WeaponController. Цей компонент відповідає за перемикання зброї, обробку вводу, відтворення анімацій, застосування віддачі та оновлення UI. WeaponController зберігає два слоти зброї, що дозволяє гравцю швидко перемикатися між ними. Він також відповідає за ефекти хитання зброї під час руху, реакцію зброї на стрибки та падіння, а також інші дрібні деталі, які роблять поведінку зброї більш живою та природною.

Взаємодія зі зброєю у світі реалізована через клас WeaponPickup, який використовує інтерфейс Interactable. Це дозволяє гравцю підібрати зброю, замінити поточну та знищити об'єкт у світі. Такий підхід робить систему взаємодії універсальною та легко розширюваною.

У результаті вдалося створити гнучку, модульну та розширювану збройову систему, яка підтримує різні типи зброї, режими стрільби, анімації, звуки та взаємодію з іншими системами гри. Вона є одним із ключових елементів геймплею та забезпечує динамічний і насичений бойовий процес. Для наочного відображення структури збройової системи та взаємозв'язків між її компонентами доцільно представити її у вигляді діаграми класів.

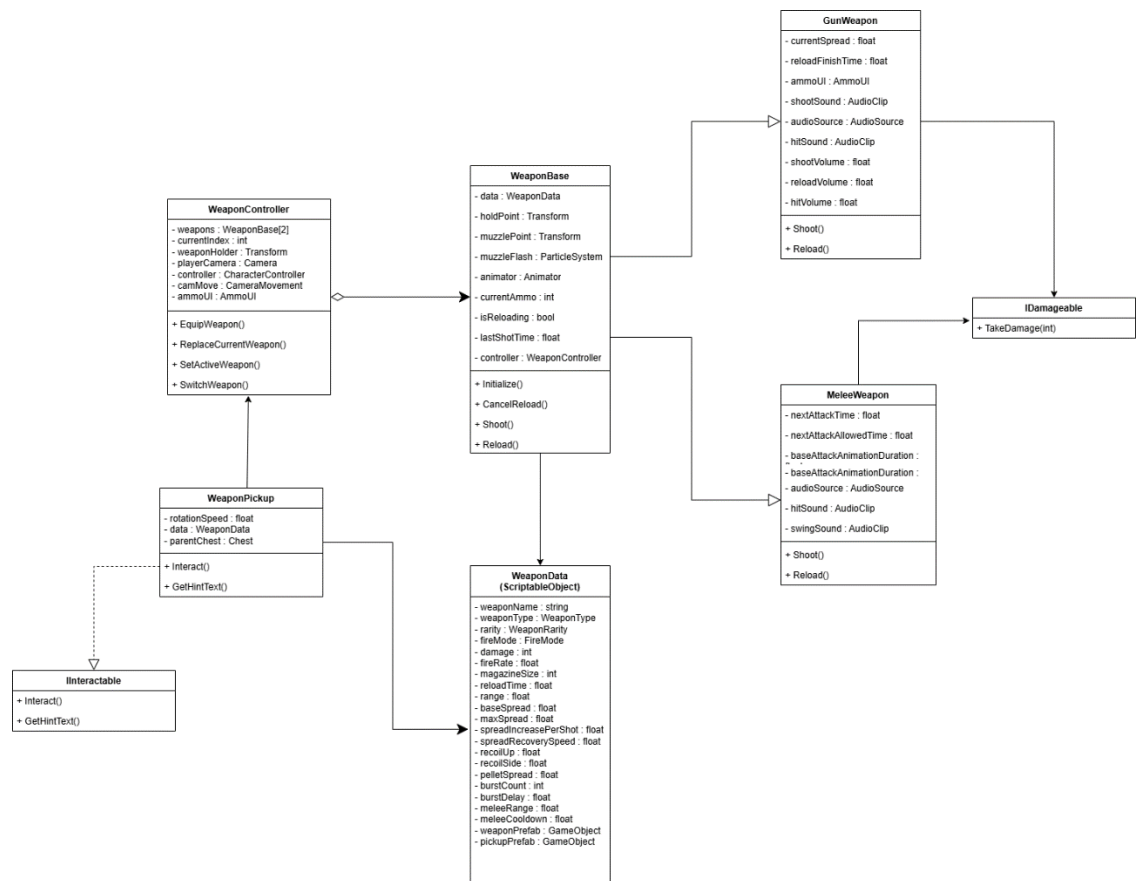


Рис 3.2 Діаграма класів збройової системи

3.2.3 Генерація рівня

Генерація рівня є однією з ключових механік гри, оскільки саме вона формує структуру проходження, визначає темп руху гравця та створює відчуття непередбачуваності, яке є характерним для жанру roguelike. Основна ідея полягала у тому, щоб побудувати систему, здатну автоматично створювати покімнатні рівні з різними варіантами розташування кімнат, дверей, ворогів та інтерактивних об'єктів. При цьому важливо було зберегти логічність структури, уникнути накладання кімнат одна на одну та забезпечити коректну взаємодію між усіма елементами.

У центрі системи знаходиться RoomManager, який відповідає за глобальний контроль зайнятих позицій. Він зберігає координати всіх уже створених кімнат у вигляді множини, що дозволяє швидко перевіряти, чи можна розмістити нову кімнату у певній точці. Під час запуску сцени RoomManager очищає список зайнятих позицій і додає до нього стартову кімнату, яка є початковою точкою для

генерації. Такий підхід дозволяє уникнути дублювання кімнат та забезпечує правильну структуру рівня.

Безпосереднє створення кімнат виконується компонентом RoomSpawner. Кожен RoomSpawner розташований у точці, де потенційно може з'явитися нова кімната. Він має напрямок, який визначає, яку саме кімнату потрібно створити. Наприклад, якщо напрямок вказує вгору, RoomSpawner вибирає випадковий префаб із набору кімнат, призначених для верхнього напрямку. Перед створенням кімнати RoomSpawner перевіряє, чи не зайнята ця позиція. Якщо позиція вже є у списку RoomManager, нова кімната не створюється. Це дозволяє уникнути накладання кімнат одна на одну та забезпечує коректну структуру рівня.

Після створення кімнати її позиція додається до глобального списку зайнятих позицій. Таким чином система поступово розширює рівень, створюючи нові кімнати у доступних напрямках. Кожна кімната має власний набір RoomSpawner-ів, тому процес генерації відбувається природно та послідовно, поки всі точки не будуть опрацьовані.

Важливою частиною системи є RoomDoorController, який відповідає за коректне відображення дверей між кімнатами. Після створення кімнати RoomDoorController перевіряє наявність сусідніх кімнат у кожному напрямку. Якщо поруч є інша кімната, двері у цьому напрямку приховуються, щоб уникнути дублювання проходів. Якщо ж поруч знаходиться стіна, вона видаляється, що дозволяє створити прохід між кімнатами. Такий підхід забезпечує логічну структуру рівня та робить переміщення між кімнатами природним.

Окрему роль у формуванні ігрового процесу відіграє клас Room. Він відповідає за поведінку кімнати після того, як гравець у неї заходить. Кожна кімната може містити кілька хвиль ворогів, які з'являються у випадкових точках. Для цього використовуються компоненти EnemySpawnPoint, які визначають можливі місця появи ворогів та типи ворогів, які можуть там з'явитися. Кожен EnemySpawnPoint має власний шанс спавну, що дозволяє створювати різні за складністю кімнати.

Після того як гравець заходить у кімнату, вона активується. Двері зачиняються, і починається перша хвиля ворогів. Коли всі вороги знищені, через певний час запускається наступна хвиля. Такий підхід дозволяє створити динамічний і напружений геймплей, де гравець повинен очистити кімнату, перш ніж рухатися далі. Після завершення всіх хвиль двері відкриваються, і гравець може продовжити шлях.

Після очищення кімнати система може створити скриню з випадковим шансом. Це робить проходження більш мотивуючим і стимулює гравця досліджувати всі кімнати. Крім того, у деяких кімнатах може з'явитися телепорт до боса. Його поява контролюється `BossTeleportManager`, який гарантує, що телепорт з'явиться лише один раз за рівень. Якщо телепорт створено, гравець може взаємодіяти з ним через компонент `TeleportToBoss`, який переносить його у спеціальну кімнату з босом.

Після перемоги над босом створюється телепорт на наступний рівень. Він працює через інтерфейс `Interactable` і дозволяє гравцю перейти до нової сцени. Після завантаження нової сцени гравець автоматично переміщується на точку появи, що забезпечує плавний перехід між рівнями.

У результаті вдалося створити гнучку та надійну систему генерації рівня, яка поєднує процедурність із логічною структурою. Вона забезпечує різноманітність проходжень, створює унікальні ситуації та підтримує загальну динаміку гри. Завдяки модульному підходу кожен компонент системи може бути розширений або змінений без впливу на інші частини проєкту, що робить систему придатною для подальшого розвитку. Для розуміння внутрішньої структури системи генерації рівня та взаємодії між її компонентами доцільно представити архітектуру у вигляді діаграми класів.

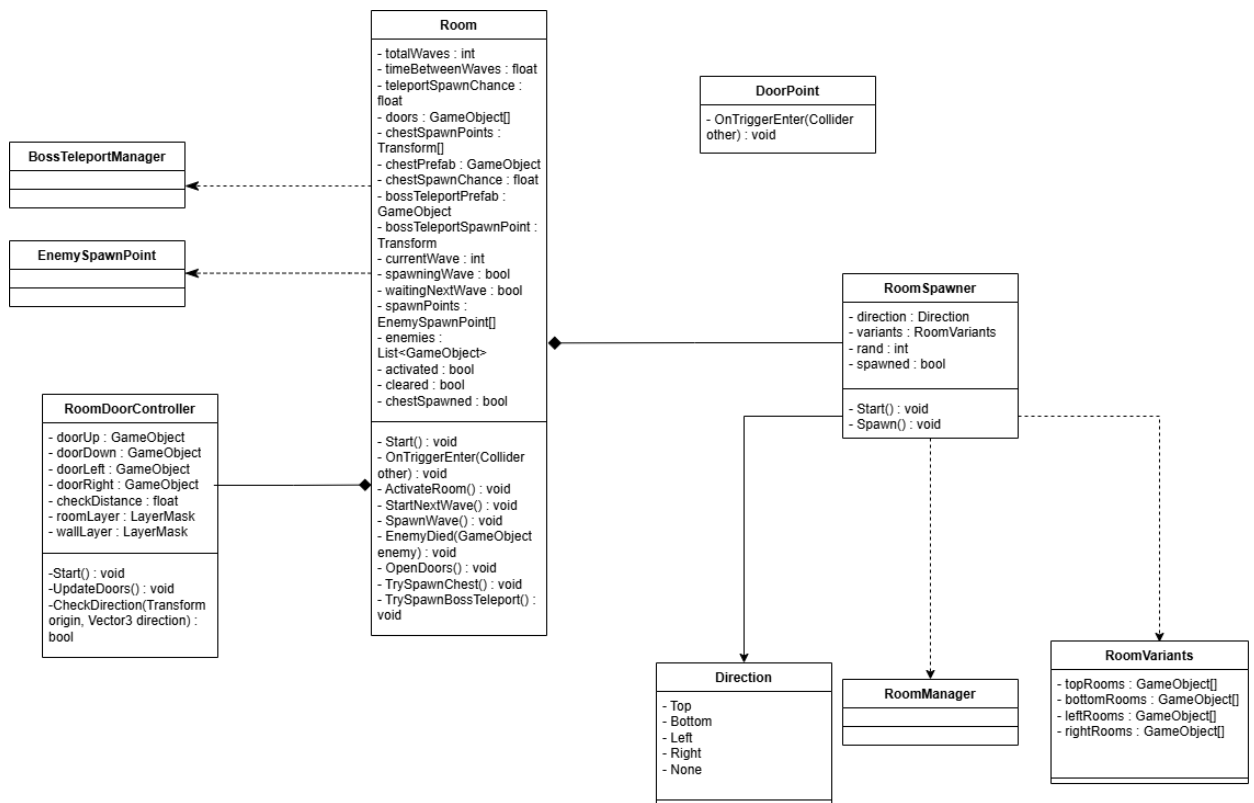


Рис. 3.3 Діаграма класів генерації рівня

3.2.4 Система ворогів

Система ворогів у грі побудована на наборі різних типів противників, кожен з яких має власні характеристики та просту модель поведінки. Основна ідея полягала у тому, щоб створити кілька різних ворогів, які відрізняються стилем атаки, дистанцією бою та загальною загрозою для гравця. Завдяки цьому бої не виглядають одноманітними, а кожен тип ворога створює свою ситуацію у кімнаті, змушуючи гравця реагувати по-різному.

Архітектура системи базується на класі `EnemyBase`, який визначає спільні для всіх ворогів властивості: здоров'я, броню, базову шкоду, частоту атак та логіку отримання ушкоджень. Саме тут реалізовано механізм зменшення шкоди з урахуванням броні, а також обробку смерті ворога. Після знищення противник повідомляє кімнату, у якій знаходиться, що дозволяє `Room` контролювати прогрес хвиль і визначати момент, коли кімната вважається очищеною.

Усі параметри ворогів винесені у `ScriptableObject EnemyData`. Це дає можливість швидко створювати нові типи противників, змінювати їхні

характеристики та балансувати складність без редагування коду. Для стріляючих зомбі використовується розширений клас `ShooterZombieData`, який додає параметри швидкості та часу існування снарядів.

У грі існує дві основні групи ворогів: людські противники та зомбі. Для людських ворогів створено проміжний клас `HumanEnemyBase`, який містить спільну логіку пересування за допомогою `NavMeshAgent`, перевірку видимості гравця та базову структуру поведінки. Саме тут визначено, як такі вороги реагують на появу гравця, коли вони починають рухатися, коли зупиняються та за яких умов можуть атакувати. Це дозволяє уникнути нелогічних ситуацій, наприклад, коли стріляючий ворог намагається атакувати крізь стіни або переслідує гравця без прямої видимості.

На основі `HumanEnemyBase` реалізовано кілька типів людських ворогів. `RaiderMelee` є ближнім бійцем, який намагається підійти максимально близько та завдати удару по області. `RaiderPistol` стріляє одиночними пострілами з невеликим відхиленням, тримаючи середню дистанцію. `BurstShooter` використовує короткі серії пострілів, контролюючи затримку між кулями та паузи між серіями. `MilitaryAK` стріляє кількома послідовними серіями, що робить його небезпечним на середній дистанції. `MilitaryShotgun` використовує фіксований набір кутів для створення широкого розльоту пострілів, що створює загрозу на близькій дистанції.

Зомбі не наслідують `HumanEnemyBase` і мають власну модель поведінки. Вони використовують окрему логіку пересування, включаючи алгоритм розосередження, який дозволяє їм не скупчуватися в одній точці та природніше рухатися групою. Їхня поведінка простіша, але добре підходить для створення масових атак та хаотичних ситуацій у кімнатах. Звичайний `Zombie` атакує на близькій дистанції, тоді як `ZombieShooter` використовує повільні снаряди, отримані з пулу снарядів. На рисунку 3.4 наведено діаграму класів зомбі ворогів

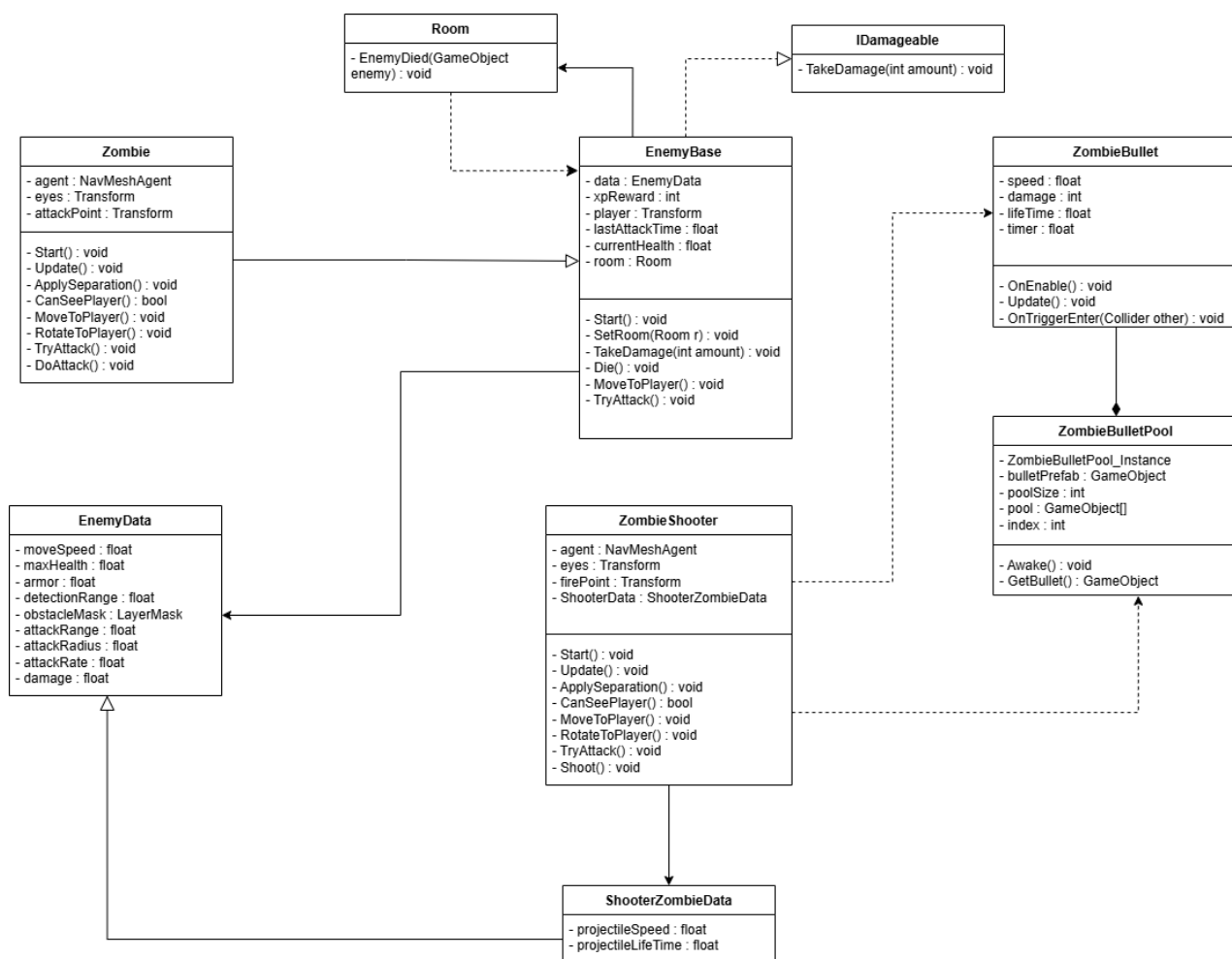


Рис. 3.4 Діаграма класів зомбі ворогів

Стріляючі вороги використовують пул куль BulletPool, що дозволяє уникнути постійного створення та знищення об'єктів. Кожна куля має власну швидкість, час існування та логіку нанесення шкоди. Після зіткнення або завершення часу життя куля повертається у пул, що робить систему ефективною з точки зору продуктивності.

Усі вороги працюють у тісній взаємодії з кімнатами, у яких вони з'являються. Кожен противник повідомляє Room про свою смерть, що дозволяє системі хвиль коректно визначати момент переходу до наступної хвилі або відкриття дверей. Завдяки цьому поведінка ворогів інтегрована у загальну структуру рівня та впливає на темп проходження.

У підсумку система ворогів складається з набору різних типів противників, кожен з яких має власну модель поведінки, стиль атаки та роль у бою. Такий підхід

дозволяє створити різноманітні ситуації у кімнатах, підтримує динаміку геймплею та робить бої більш цікавими для гравця.

3.2.5 Система босів

Система босів у грі побудована таким чином, щоб кожен бос створював окремий, відчутно складніший виклик порівняно зі звичайними ворогами. Основна ідея полягала у тому, щоб зробити босів не просто сильнішими версіями звичайних противників, а окремими бойовими одиницями з власними фазами, унікальними атаками та додатковими механіками. Завдяки цьому бій з босом стає кульмінацією рівня і вимагає від гравця більшої уваги та адаптації до змінюваних умов.

Усі боси реалізують інтерфейс IBoss, який визначає базові вимоги до будь-якого боса: максимальну кількість здоров'я та можливість прив'язати боса до кімнати, у якій він з'являється. Це дозволяє BossRoom контролювати перебіг бою та реагувати на смерть боса, наприклад, створюючи телепорт на наступний рівень.

Створення боса відбувається через BossSpawnPoint, який відповідає за появу конкретного префабу у визначеній точці кімнати. Після появи BossRoom активує інтерфейс здоров'я боса, прив'язує його до кімнати та запускає логіку бою. Коли бос знищується, BossRoom приховує панель здоров'я та створює телепорт, який дозволяє гравцю перейти далі.

У грі реалізовано кілька різних босів, кожен з яких має власну модель поведінки. MilitaryTurretBoss є стаціонарним противником, який зосереджений на стрільбі. Він використовує серії пострілів, що складаються з кількох коротких черг, між якими робить паузи. Після певної кількості таких серій турель перегрівається і на деякий час стає вразливою до шкоди. Це створює чіткий ритм бою, у якому гравець повинен ухилятися від пострілів і чекати моменту, коли бос стане доступним для атаки. Додатково турель може викликати підкріплення, створюючи навколо себе групу ворогів, що ускладнює бій.

ArmoredBanditBoss є рухомим босом ближнього бою. Він переслідує гравця за допомогою навігації, намагається підійти на близьку дистанцію та завдати сильного удару. Завдяки високій броні він отримує значно менше шкоди, що

робить бій тривалішим. Бос також може викликати підкріплення, створюючи додаткових ворогів у зоні навколо себе. Це змушує гравця постійно рухатися та контролювати простір кімнати.

FatZombieBoss має багатофазну модель поведінки, яка змінюється залежно від кількості здоров'я. На першій фазі він стріляє кільцями снарядів, що розходяться у різні боки. На другій фазі частота атак зростає, а бос отримує можливість викликати зомбі. На третій фазі він починає пересуватися за гравцем, зберігаючи при цьому стрільбу кільцями та збільшуючи інтенсивність виклику підкріплення. Така структура робить бій поступово складнішим і змушує гравця адаптуватися до нових умов.

Для стрільби боси використовують окремий буфер снарядів `BossBulletPool`, що дозволяє уникнути постійного створення нових об'єктів. Кожен снаряд має власну швидкість, напрямок, час існування та логіку нанесення шкоди. Після зіткнення або завершення часу життя снаряд повертається у буфер, що забезпечує стабільну продуктивність навіть під час інтенсивних атак.

Панель здоров'я боса реалізована у класі `BossHealthUI`. Вона відображається лише під час бою з босом і оновлюється щоразу, коли бос отримує шкоду. Після завершення бою панель приховується, а кімната створює телепорт, який дозволяє гравцю перейти на наступний рівень.

3.2.6 Карткова система покращень

Карткова система прокачки у грі відповідає за розвиток персонажа під час проходження рівнів. Основна ідея полягала у тому, щоб дати гравцю можливість поступово посилювати свого героя, але робити це не через складні дерева навичок, а через простий і зрозумілий вибір із кількох випадкових покращень. Такий підхід дозволяє зберегти динаміку гри, не відволікаючи гравця на довгі меню, і водночас створює відчуття варіативності кожного проходження.

У центрі системи знаходиться об'єкт `UpgradeCard`, який описує одну картку покращення. Кожна картка містить назву, опис, іконку та набір покращень. На

відміну від простих систем, де одна картка відповідає за одне покращення, у цій реалізації картка може містити кілька ефектів одночасно. Це дозволяє створювати цікаві комбінації, наприклад, одночасне збільшення швидкості руху та зменшення затримки ривка, або підсилення шкоди разом зі зменшенням розльоту пострілів. Кожен ефект описується типом покращення та числовим значенням, яке зазвичай виражене у відсотках.

Під час отримання нового рівня гравцю пропонується вибрати одну з трьох випадкових карток. За це відповідає `UpgradeManager`, який керує вибором карток, їхнім відображенням та застосуванням. Менеджер містить повний набір усіх доступних карток і під час показу використовує перемішування, щоб гарантувати випадковість вибору. Після перемішування гравцю пропонуються три перші картки з оновленого списку, що забезпечує унікальність кожного набору.

Під час показу карток гра ставиться на паузу: час зупиняється, а курсор миші розблоковується. Це дозволяє гравцю спокійно прочитати описи та обрати потрібне покращення. Відображення карток здійснюється через компонент `UpgradeCardView`, який відповідає за встановлення іконки, назви та опису картки у відповідні елементи інтерфейсу.

Після вибору картки `UpgradeManager` застосовує всі покращення, які вона містить. Для цього він звертається до `PlayerStats`, класу, який зберігає всі характеристики гравця та містить методи для їхнього збільшення. Кожен тип покращення має власний метод, наприклад, збільшення максимальної кількості здоров'я, підсилення шкоди від пострілів, зменшення затримки між атаками, підсилення ривка або покращення параметрів відновлення здоров'я. Завдяки цьому система легко розширюється: щоб додати нове покращення, достатньо створити новий тип у переліку `UpgradeType` та реалізувати відповідний метод у `PlayerStats`.

Після застосування покращення панель карток приховується, гра продовжується, а курсор знову блокується. Такий цикл повторюється кожного разу, коли гравець отримує новий рівень, що створює природний ритм розвитку персонажа.

Карткова система прокачки робить кожне проходження унікальним, оскільки набір доступних покращень змінюється випадковим чином. Гравець може комбінувати різні ефекти, створюючи власний стиль гри: робити героя швидким і мобільним, посилювати ближній бій, зосереджуватися на стрільбі або покращувати виживання. Завдяки цьому система залишається простою у використанні, але водночас забезпечує достатню глибину для різних стратегій. Щоб чітко відобразити взаємодію між гравцем, менеджером покращень та системою характеристик, нижче наведено діаграму послідовності, яка демонструє повний цикл обробки вибору картки.

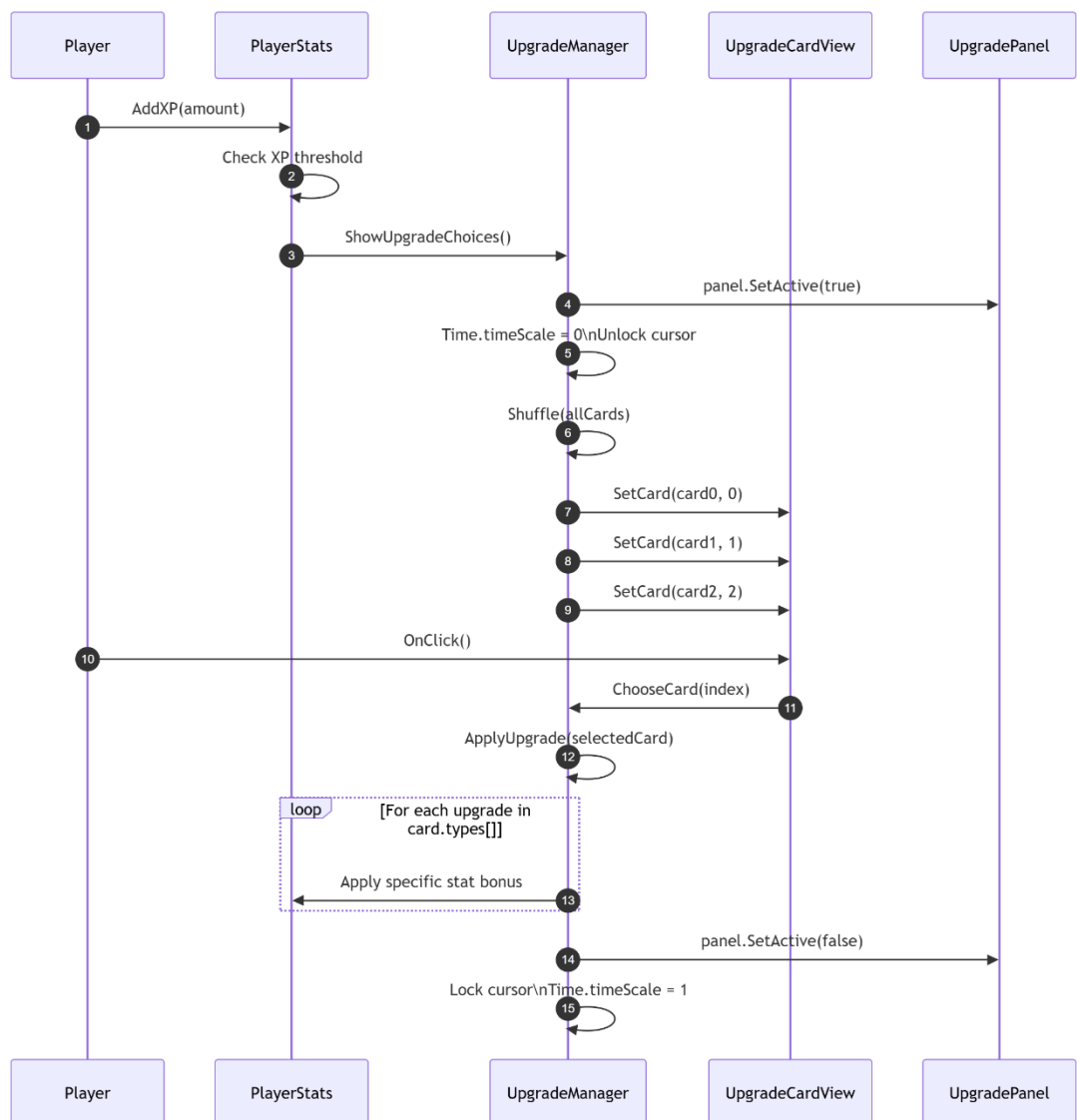


Рис. 3.5 Діаграма послідовності системи покращень

3.3 Проєктування даних

Проєктування даних у грі відіграє ключову роль, оскільки саме від структури та організації даних залежить гнучкість системи, можливість швидкого балансування та простота подальшого розширення проєкту[19, 20]. У грі використовується підхід, заснований на ScriptableObject, спеціальних об'єктах Unity, які дозволяють зберігати конфігураційні параметри окремо від логіки та сцени[21]. Це дає змогу створювати централізовані набори даних, які легко редагувати, повторно використовувати та масштабувати без необхідності змінювати код.

Однією з головних причин використання ScriptableObject є те, що вони існують як окремі файли у проєкті, а не як частина сцени[21]. Це означає, що дані не дублюються між рівнями, не створюють зайвого навантаження на пам'ять і можуть бути використані будь-яким компонентом у будь-якій сцені. Наприклад, один файл EnemyData може визначати характеристики десятків ворогів, які з'являються у різних кімнатах, і будь-яка зміна в цьому файлі автоматично вплине на всі екземпляри ворога. Такий підхід значно спрощує балансування гри, оскільки всі параметри зосереджені в одному місці[19].

У грі використовується кілька основних ScriptableObject-ів, кожен з яких відповідає за окрему групу даних. Першою групою є дані ворогів. Клас EnemyData містить базові характеристики будь-якого противника ближнього бою: швидкість руху, максимальну кількість здоров'я, броню, дальність виявлення гравця, параметри атаки та інші властивості. Це дозволяє створювати різні типи ворогів, змінюючи лише значення у відповідному файлі. Наприклад, у конфігурації «Fat Zombie» (див. рис. 3.1) видно, що цей ворог має підвищену кількість здоров'я, низьку швидкість руху та значну броню, що робить його повільним, але витривалим противником. Усі ці параметри зберігаються у ScriptableObject і можуть бути змінені в будь-який момент без редагування коду.



Рис. 3.1 Конфігурація ворога «Fat Zombie»

Для стріляючих ворогів використовується розширений клас `ShooterZombieData`, який успадковує `EnemyData` та додає параметри дальньої атаки. У цьому `ScriptableObject` зберігається швидкість польоту снаряда, час його існування, а також збільшена дальність атаки. Завдяки цьому стріляючі зомбі можуть мати власну модель поведінки, але при цьому використовувати спільну базу даних ворогів. У конфігурації «Shooter Zombie» (див. рис. 3.2) видно, що цей тип ворога має менше здоров'я, але значно більшу дальність атаки, що робить його небезпечним на відкритих ділянках рівня.

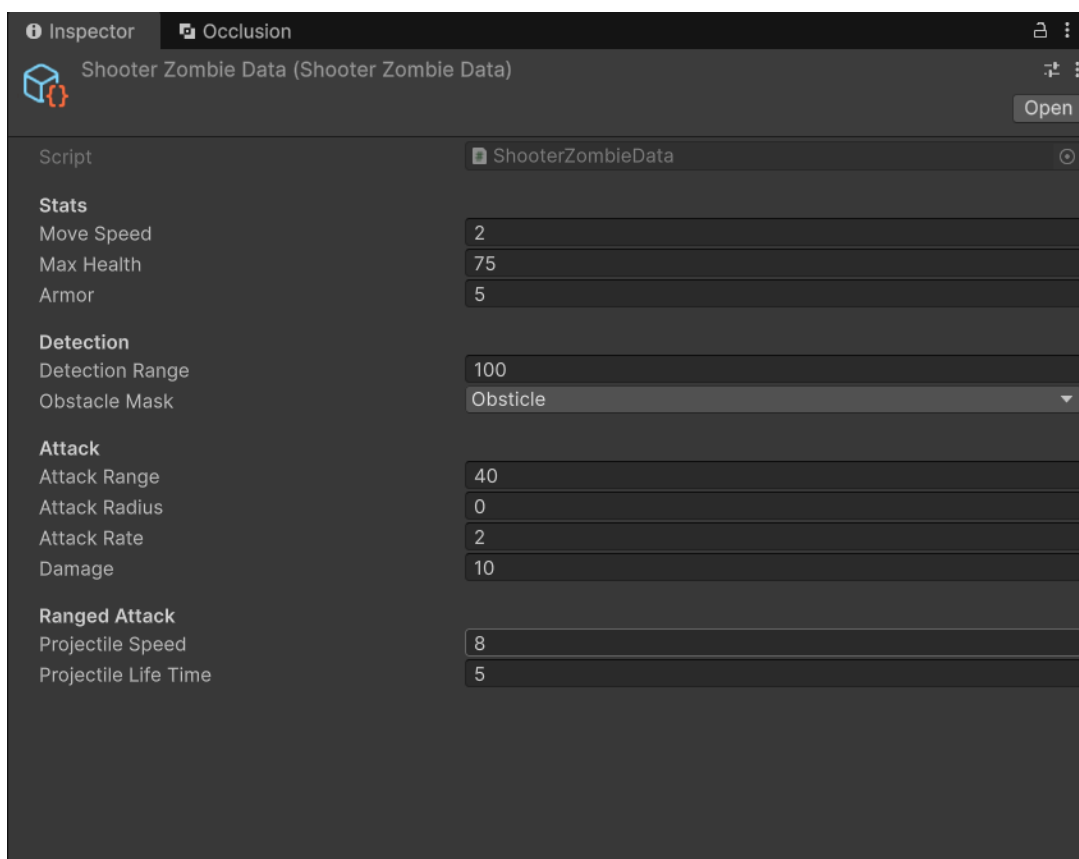


Рис. 3.2 Конфігурація противника «Shooter Zombie»

Окрему групу даних становлять картки покращень, реалізовані у класі UpgradeCard. Кожна картка містить назву, опис, іконку та набір покращень, які вона застосовує. Важливо, що картка може містити кілька ефектів одночасно. Це дозволяє створювати складніші та цікавіші варіанти розвитку персонажа. Наприклад, у картці «Assault Module» (див. рис. 3.3) одночасно підвищується шкода від стрільби та зменшується час перезарядки, що робить її корисною для гравців, які роблять ставку на дальній бій. Усі покращення описуються через перелік типів UpgradeType та відповідні числові значення, що дозволяє легко додавати нові типи ефектів або змінювати існуючі.

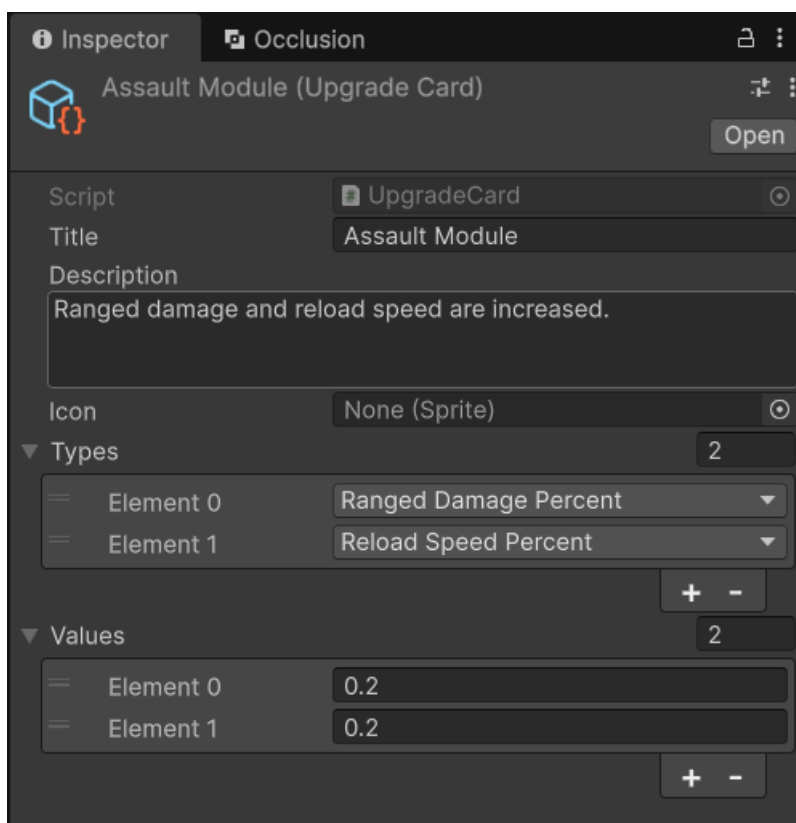


Рис. 3.3 Конфігурація картки «Assault Module»

Ще одним важливим ScriptableObject є WeaponData, який описує характеристики зброї. У цьому файлі зберігаються всі параметри, що визначають поведінку зброї: шкода, швидкість стрільби, розмір магазину, час перезарядки, розльот, віддача, кількість дробин для дробовиків, параметри серійної стрільби та інші властивості. Наприклад, у конфігурації «M4A4» (див. рис. 3.4) видно, що ця зброя має високу швидкість стрільби, середню шкоду та помірну віддачу, що робить її універсальним варіантом для середньої дистанції. Завдяки ScriptableObject усі ці параметри можна змінювати без редагування коду, що значно спрощує процес балансування зброї.

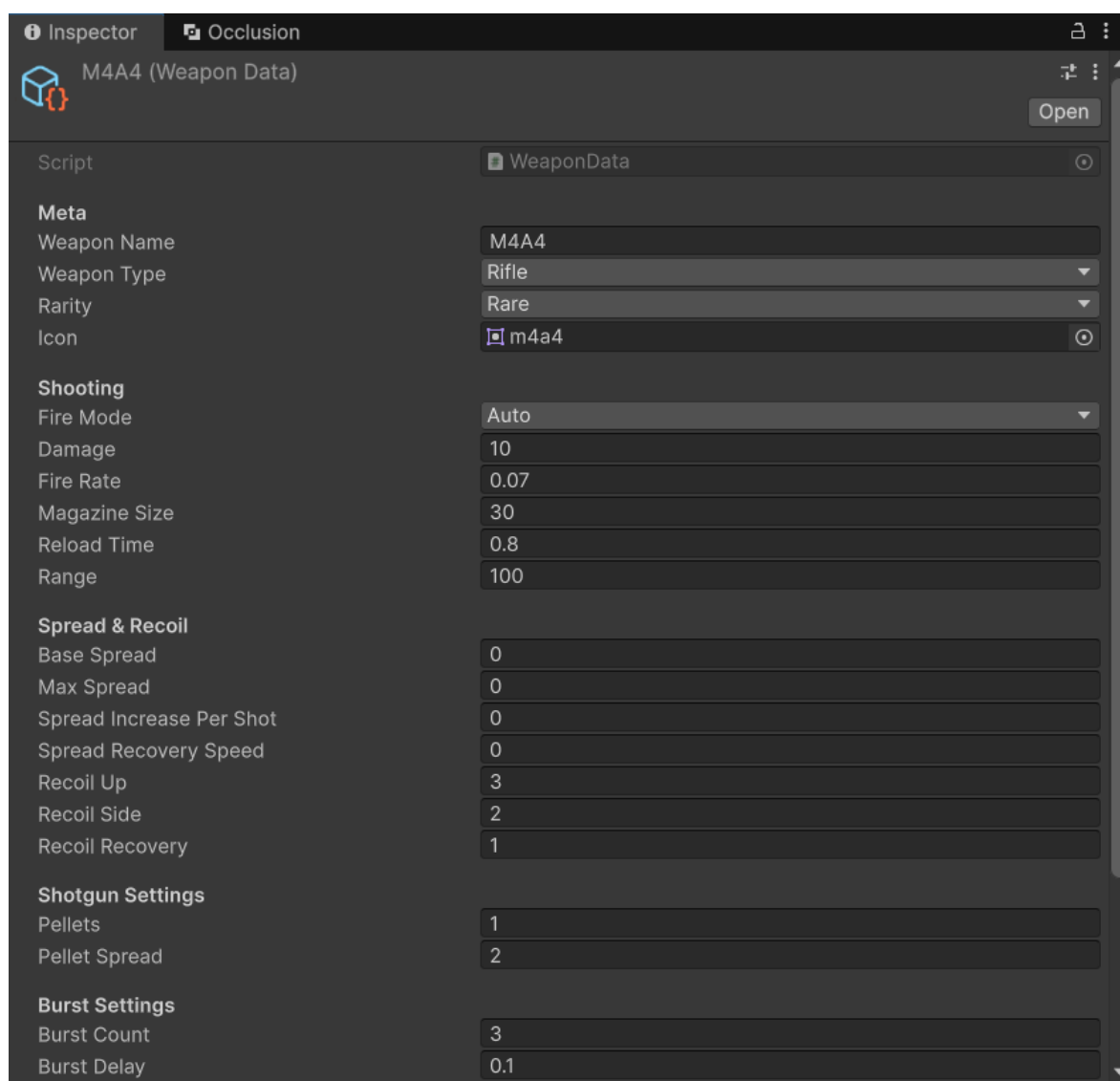


Рис. 3.4 Конфігурації зброї «М4А4»

Використання `ScriptableObject` у ролі контейнерів даних дозволяє створити чітку, логічну та масштабовану структуру проєкту. Усі параметри ворогів, зброї та карток покращень зберігаються у єдиному форматі, що робить систему зрозумілою та легкою для підтримки. Крім того, це дозволяє розробнику швидко експериментувати з балансом, змінюючи значення у конфігураторах без необхідності перекомпіляції коду або зміни логіки поведінки об'єктів.

У підсумку використання `ScriptableObject` стало основою для побудови гнучкої та керованої системи даних у грі. Такий підхід забезпечує зручність редагування, повторне використання параметрів, централізоване керування балансом та можливість швидкого розширення проєкту без ризику порушити

існуючу логіку. Це робить структуру даних не лише ефективною, але й стійкою до змін, що є важливою вимогою для будь-якого сучасного ігрового проєкту[21].

3.4 Інтерфейс користувача

Інтерфейс користувача у грі був спроектований таким чином, щоб забезпечити гравця всією необхідною інформацією, але при цьому не перевантажувати екран зайвими елементами. Оскільки гра має динамічний характер, а основний акцент робиться на бою, ухиленні та швидкому прийнятті рішень, UI виконано у мінімалістичному стилі. На екрані присутні лише ті елементи, які безпосередньо впливають на ігровий процес: приціл, панель здоров'я, панель досвіду, індикатор зброї та боезапасу, підказки взаємодії, панель здоров'я боса та меню вибору карток покращень. Такий підхід дозволяє зберегти чистоту зображення та не відволікати гравця від основної дії.

Почнемо з найпростішого елемента — прицілу. Він розташований у центрі екрана і являє собою невелику точку, яка не перекриває огляд і не заважає прицілюванню. Його можна побачити на рисунку 3.5, де добре видно, що приціл не має анімацій чи декоративних елементів. Це було зроблено свідомо, щоб не відволікати гравця під час стрільби та не створювати зайвого візуального шуму.

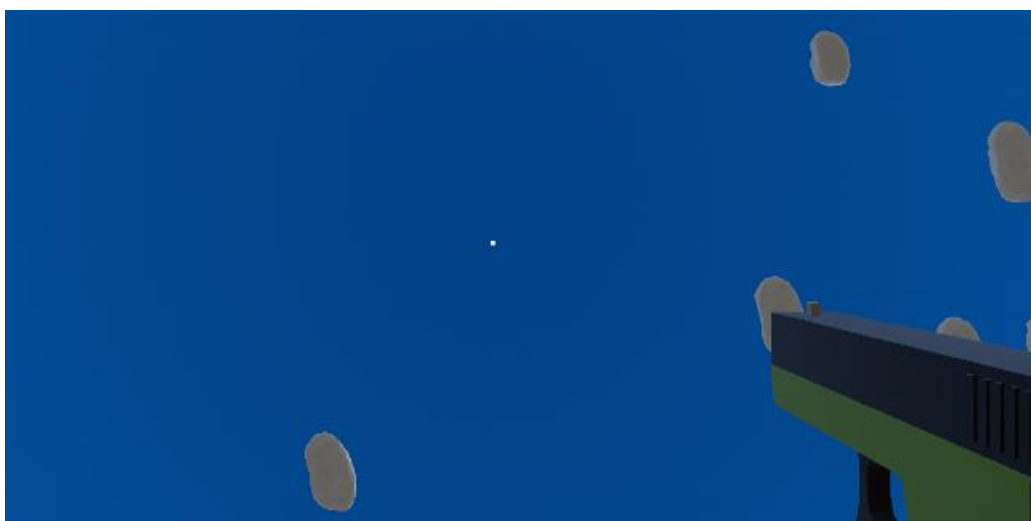


Рис. 3.5 Приціл гравця

У нижній частині екрана, розташована панель здоров'я гравця. Вона відображає поточний стан персонажа та дозволяє швидко оцінити, наскільки гравець у безпеці. На рисунку 3.6 можна побачити червону смугу, яка зменшується при отриманні шкоди. Технічно панель реалізована через компонент Slider, а її оновлення здійснюється скриптом PlayerHealthBar, який підписується на подію зміни здоров'я у PlayerStats. Завдяки цьому панель реагує миттєво, без зайвих викликів у кожному кадрі, що позитивно впливає на продуктивність.

Також над здоров'ям гравця розташована панель досвіду. Вона показує прогрес до наступного рівня та дозволяє гравцю розуміти, коли він отримає доступ до нових покращень. На рисунку 3.6 ще можна побачити жовту смугу, яка заповнюється під час отримання досвіду. Оновлення панелі здійснюється скриптом XPBar, який працює за тим самим принципом, що й панель здоров'я: він підписується на подію зміни досвіду та оновлює значення слайдера лише тоді, коли це необхідно.



Рис. 3.6 Полоси здоров'я та досвіду

Панелі здоров'я та досвіду утворюють єдину систему відображення стану гравця. Обидві працюють на основі подій, що дозволяє уникнути зайвих обчислень у кожному кадрі. Обидві розташовані таким чином, щоб не перекривати огляд, але залишатися завжди помітними. І обидві виконані у мінімалістичному стилі, який відповідає загальній візуальній концепції гри.

У правому нижньому куті екрана розташована панель зброї та боєзапасу. На рисунку 3.7 можна побачити білу іконку зброї та напис “20 / 20”, який відображає кількість патронів у магазині та загальний боєзапас. За оновлення іконки відповідає скрипт WeaponUI, який реагує на зміну зброї у WeaponController. За оновлення

тексту патронів відповідає скрипт AmmoUI, який отримує дані від системи стрільби. Такий поділ дозволяє легко змінювати логіку зброї, не зачіпаючи UI, і навпаки.



Рис. 3.7 Панель з іконкою зброї та боеприпасів

Окремим елементом інтерфейсу є підказки взаємодії. Вони з'являються лише тоді, коли гравець може взаємодіяти з об'єктом, наприклад, підібрати зброю або активувати механізм. На рисунку 3.8 можна побачити невелике текстове повідомлення, яке з'являється над об'єктом. За його роботу відповідає скрипт InteractUI, який показує або приховує підказку залежно від ситуації. Це дозволяє не перевантажувати екран зайвими елементами, але при цьому забезпечує гравця необхідною інформацією.



Рис. 3.8 Підказка для гравця

Під час бою з босом у верхній частині екрана з'являється окрема панель здоров'я боса. На рисунку 3.9 видно широку червону смугу, яка відображає стан здоров'я ворога. Вона активується лише тоді, коли гравець входить у кімнату боса, і приховується після його смерті. За її роботу відповідає скрипт BossHealthUI, який отримує дані від системи босів. Такий підхід дозволяє зробити бій з босом більш зрозумілим і напруженим, оскільки гравець бачить, наскільки близький до перемоги.

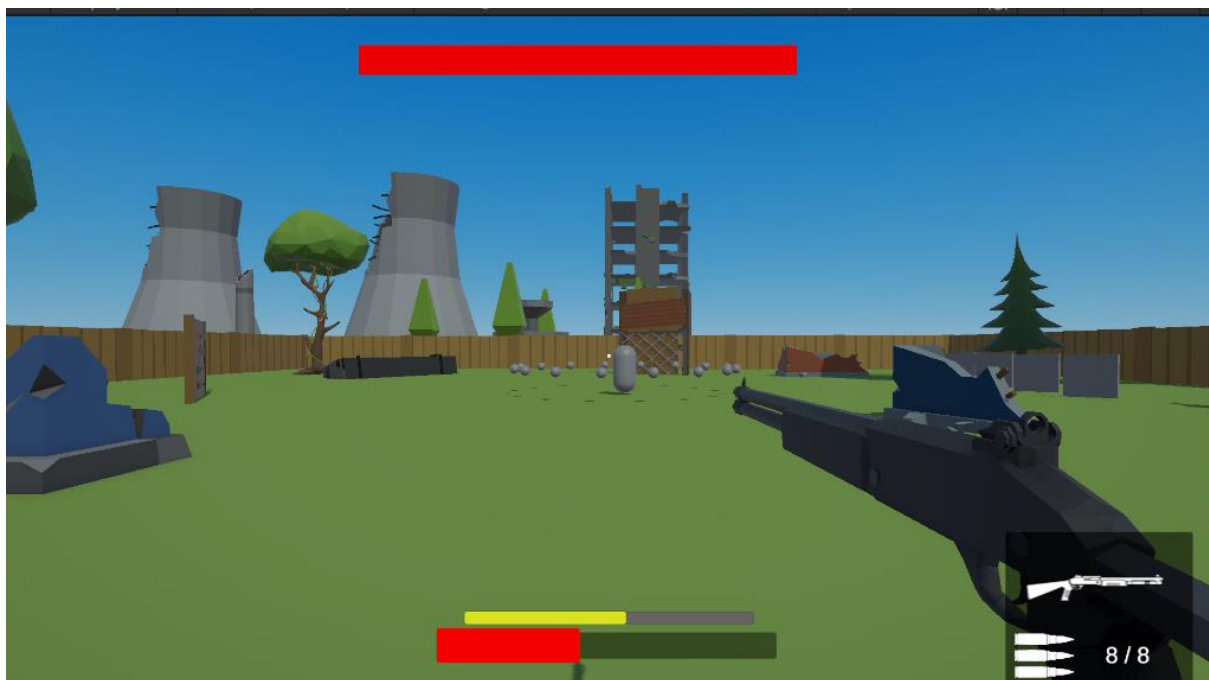


Рис. 3.9 Смушка здоров'я боса

Нарешті, одним із найважливіших елементів інтерфейсу є панель вибору карток покращень. Вона з'являється після отримання нового рівня та дозволяє гравцю обрати один із трьох варіантів розвитку. На рисунку 3.10 можна побачити три картки з назвами, описами та кнопками вибору. Під час появи панелі гра ставиться на паузу, а курсор розблоковується, щоб гравець міг спокійно прочитати описи. За роботу панелі відповідають `UpgradeManager` та `UpgradeCardView`, які заповнюють картки даними та обробляють вибір гравця. Цей елемент інтерфейсу є ключовим для системи прогресії та значно впливає на стиль гри.

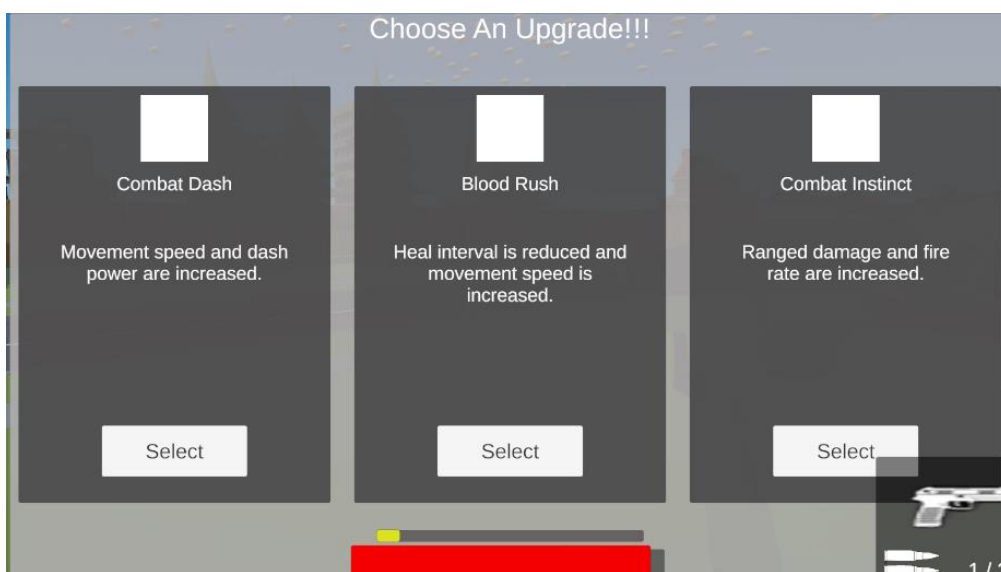


Рис. 3.10 Панель вибору карток покращень

Усі UI-елементи об'єднані кореневим об'єктом `UIRoot`, який не знищується при переході між сценами. Це дозволяє уникнути дублювання інтерфейсу та забезпечує стабільну роботу UI протягом усього проходження.

3.5 Оптимізація гри

Оптимізація стала важливою частиною розробки, оскільки гра містить велику кількість динамічних об'єктів, ворогів, снарядів та елементів інтерфейсу, які активно взаємодіють між собою. Основна мета оптимізації полягала у тому, щоб забезпечити стабільну частоту кадрів навіть у найбільш напружені моменти бою, коли на екрані одночасно присутні десятки ворогів, снарядів та ефектів. Для цього

було застосовано кілька підходів, які дозволили зменшити навантаження на процесор та пам'ять, не погіршуючи якість ігрового процесу.

Одним із ключових рішень стало використання пулів об'єктів для снарядів, атак босів та інших тимчасових елементів. Замість постійного створення та знищення об'єктів, що є дорогими операціями для Unity, гра повторно використовує вже існуючі екземпляри. Це дозволило значно зменшити кількість викликів `Instantiate` та `Destroy`, які зазвичай є одними з головних джерел просідань продуктивності. Завдяки цьому навіть під час інтенсивних перестрілок гра зберігає плавність, а час реакції системи залишається стабільним.

Ще одним напрямом оптимізації стала робота з оновленням логіки ворогів. Оскільки вороги активно пересуваються, атакують та реагують на гравця, їхня поведінка могла б створювати значне навантаження, якби всі обчислення виконувалися щокадру. Тому частина перевірок була перенесена у більш рідкісні інтервали або розподілена між кадрами. Наприклад, перевірка видимості гравця та визначення маршруту пересування не виконуються щомиті, а оновлюються з певною затримкою, що не впливає на якість поведінки, але суттєво зменшує навантаження.

Окрему увагу було приділено оптимізації системи генерації рівнів. Оскільки кімнати створюються процедурно, важливо було уникнути зайвих обчислень під час їхнього формування. Для цього використовуються попередньо підготовлені шаблони кімнат, а логіка розміщення елементів була спрощена таким чином, щоб уникати рекурсивних або надто складних перевірок. Це дозволило забезпечити швидке завантаження рівнів та плавний перехід між кімнатами.

Також було оптимізовано роботу інтерфейсу. Усі UI-елементи оновлюються лише тоді, коли це необхідно, завдяки подієвій моделі. Наприклад, панель здоров'я змінюється лише при отриманні шкоди, а панель досвіду — лише при отриманні ХР. Це дозволило уникнути зайвих викликів у кожному кадрі та зменшити навантаження на систему. Крім того, кореневий об'єкт інтерфейсу не знищується між сценами, що дозволяє уникнути повторного створення UI та зайвих витрат пам'яті.

Загалом оптимізація гри базувалася на поєднанні кількох підходів: повторному використанні об'єктів, розподілі обчислень у часі, спрощенні логіки генерації рівнів та подієвій моделі оновлення інтерфейсу. Усі ці рішення дозволили забезпечити стабільну продуктивність навіть у найскладніших ситуаціях, зберігаючи при цьому якість ігрового процесу та плавність анімацій. Завдяки цьому гра працює стабільно на широкому спектрі пристроїв і забезпечує комфортний досвід для гравця.

4 ТЕСТУВАННЯ

Тестування є важливою складовою процесу розробки відеоігор, оскільки саме на цьому етапі перевіряється коректність роботи всіх механік, стабільність ігрового процесу та відповідність гри функціональним і нефункціональним вимогам. На відміну від класичних програмних продуктів, відеоігри поєднують у собі логіку, фізику, штучний інтелект, анімації, візуальні ефекти та взаємодію з користувачем у реальному часі. Тому тестування в геймдеві охоплює широкий спектр аспектів: від перевірки базових механік руху до оцінки продуктивності під час інтенсивних бойових сцен.

У процесі розробки гри тестування проводилося паралельно з реалізацією функціоналу. Кожна нова система перевірялася окремо, після чого тестувалася у взаємодії з іншими компонентами. Такий підхід дозволив виявляти помилки на ранніх етапах, швидко їх виправляти та забезпечити стабільну роботу гри в цілому. Тестування включало функціональні перевірки, ігрове тестування, тестування продуктивності, а також перевірку коректності процедурної генерації рівнів. Особлива увага приділялася поведінці ворогів, роботі зброї, системі здоров'я, переходам між кімнатами та інтерфейсу користувача.

4.1 Тестові сценарії

Тестовий сценарій - Базова стрільба (рейкаст)

Мета: перевірити коректність роботи механіки стрільби через рейкаст та нанесення шкоди ворогам.

Кроки:

- навести приціл на ворога;
- виконати постріл;
- переконатися, що рейкаст визначив влучання;
- перевірити, чи зменшилося здоров'я ворога;
- перевірити, чи зменшилася кількість патронів.

Очікуваний результат: ворог отримує шкоду відповідно до параметрів зброї, боєзапас зменшується на одиницю.

Тестовий сценарій - Атака ближнього ворога

Мета: перевірити, що ворог ближнього бою завдає шкоди при вході гравця в радіус атаки.

Кроки:

- підійти до ворога на відстань атаки;
- дочекатися моменту, коли ворог виконує удар;
- перевірити зміну здоров'я гравця.

Очікуваний результат: здоров'я гравця зменшується відповідно до параметрів ближньої атаки ворога.

Тестовий сценарій - Атака далекобійного ворога

Мета: перевірити роботу механіки атаки стріляючих ворогів та коректність нанесення шкоди снарядам.

Кроки:

- стати в зоні атаки стріляючого ворога;
- дочекатися моменту випуску снаряда;

- дозволити снаряду влучити в гравця;
- перевірити зміну здоров'я.

Очікуваний результат: снаряд влучає в гравця та завдає шкоди відповідно до параметрів

Тестовий сценарій - Автоматичне відновлення здоров'я

Мета: перевірити, що система природного відновлення здоров'я активується при низькому рівні НР.

Кроки:

- отримати шкоду до рівня нижче порогу активації відновлення;
- залишатися без дій протягом кількох секунд;
- спостерігати за зміною показника здоров'я.

Очікуваний результат: після досягнення критично низького рівня здоров'я запускається механізм поступового природного відновлення, і показник НР починає зростати.

Тестовий сценарій - Зміна зброї

Мета: перевірити коректність роботи системи перемикання зброї.

Кроки:

- підібрати нову зброю;
- перевірити, чи змінилася активна зброя;
- виконати постріл для підтвердження працездатності.

Очікуваний результат: активна зброя змінюється, стрільба працює коректно.

Тестовий сценарій - Оновлення боєзапасу

Мета: перевірити коректність роботи індикатора боєзапасу.

Кроки:

- виконати кілька пострілів;
- перевірити зміну значення боєзапасу;
- виконати перезарядку;

- перевірити відновлення значення до максимального.

Очікуваний результат: індикатор боєзапасу коректно відображає зміни.

Тестовий сценарій - Поява панелі здоров'я боса

Мета: перевірити коректність активації інтерфейсу боса.

Кроки:

- увійти в кімнату боса;
- спостерігати за появою панелі здоров'я;

Очікуваний результат: панель з'являється одразу після входу в кімнату з босом

Тестовий сценарій - Оновлення здоров'я боса

Мета: перевірити коректність зменшення НР боса під час бою.

Кроки:

- завдати босу шкоди;
- спостерігати за зміною значення на панелі;
- повторити кілька разів.

Очікуваний результат: панель коректно відображає зменшення здоров'я.

Тестовий сценарій - Генерація кімнат

Мета: перевірити коректність процедурної генерації рівнів.

Кроки:

- запустити нову гру;
- пройти кілька кімнат;
- перевірити, чи не накладаються кімнати одна на одну;

Очікуваний результат: кімнати генеруються коректно та стабільно.

Тестовий сценарій - Поява ворогів у кімнатах

Мета: перевірити правильність точок появи ворогів.

Кроки:

- увійти в нову кімнату;
- спостерігати за появою ворогів;
- перевірити, чи не з'являються вони у стінах або недоступних місцях.

Очікуваний результат: вороги з'являються у визначених точках, не застрягають у геометрії.

Тестовий сценарій - Поява панелі апгрейдів

Мета: перевірити коректність роботи системи покращень.

Кроки:

- отримати достатньо досвіду для підвищення рівня;
- спостерігати за появою панелі вибору карток;
- перевірити блокування руху гравця.

Очікуваний результат: панель з'являється, гра ставиться на паузу, рух блокується.

Тестовий сценарій - Застосування апгрейду

Мета: перевірити коректність зміни параметрів гравця після вибору картки.

Кроки:

- обрати одну з карток;
- перевірити зміну відповідних характеристик.

Очікуваний результат: ефект картки застосовується коректно.

Тестовий сценарій - Набір досвіду

Мета: перевірити коректність роботи панелі досвіду.

Кроки:

- знищити ворога;
- спостерігати за зміною індикатора досвіду.

Очікуваний результат: індикатор збільшується відповідно до отриманого ХР.

Тестовий сценарій - Перехід на наступний рівень

Мета: перевірити коректність роботи переходів між рівнями.

Кроки:

- активувати переміщення на наступний рівень;
- дочекатися завантаження наступного рівня;
- перевірити правильність позиції гравця після переходу.

Очікуваний результат: перехід відбувається без затримок, гравець з'являється у правильній точці.

Тестовий сценарій - Продуктивність

Кроки:

- запустити гру;
- увійти в кімнату з противниками;
- активно переміщатися, стріляти та взаємодіяти з об'єктами;
- спостерігати за частотою кадрів під час бою;
- перевірити стабільність FPS при великій кількості снарядів;
- оцінити завантаження системних ресурсів;
- перевірити роботу гри на різних налаштуваннях графіки;
- проаналізувати результати тестування.

Очікуваний результат: гра працює стабільно, частота кадрів залишається на прийнятному рівні, відсутні різкі просідання або затримки.

ВИСНОВКИ

Результатом кваліфікаційної роботи є розроблена відеогра, що поєднує в собі жанри FPS та roguelike, завдяки чому підвищується ступінь зацікавленості гравця.

У ході роботи було:

1. Проаналізовано ключові характеристики жанрів FPS та roguelike, визначено їхні спільні та відмінні риси, а також особливості, що формують унікальний ігровий досвід. Це дало змогу окреслити набір механік, необхідних для поєднання швидкісної шутерної динаміки з процедурною непередбачуваністю, що є основою концепції створеної гри.

2. На основі дослідження та аналізу аналогів сформовано перелік функціональних та нефункціональних вимог: система руху та керування персонажем, механіка стрільби, процедурна генерація рівнів, поведінка ворогів і босів, карткова система покращень, інтерфейс користувача, а також продуктивність, стабільність роботи та вимоги до апаратного забезпечення. Це забезпечило чітке технічне бачення майбутнього проєкту.

3. Проведено огляд сучасних ігрових рушіїв: Unity, Unreal Engine та Godot. Порівняння можливостей, продуктивності, доступності інструментів та екосистеми плагінів дозволило обґрунтувати вибір Unity як оптимального середовища для реалізації проєкту. Рушій забезпечив необхідну гнучкість, підтримку 3D графіки, зручну роботу зі скриптами та ефективні засоби для створення процедурної генерації.

4. Реалізовано основні механіки гри: систему керування персонажем із підтримкою ривків, модульну збройову систему зі стрільбою через рейкаст, процедурну генерацію кімнат, поведінку ворогів ближнього та дальнього бою, логіку босів, а також карткову систему покращень характеристик героя. Створено мінімалістичний, але інформативний інтерфейс користувача, що включає панелі здоров'я, досвіду, боєзапасу, зброї, підказки взаємодії та меню вибору покращень.

5. Проведено тестування основних систем гри, включно з перевіркою генерації рівнів, поведінки ворогів, механіки бою, роботи системи прокачки та взаємодії між компонентами. Окремо виконано тестування продуктивності, яке підтвердило стабільну частоту кадрів навіть у складних бойових сценах. Усі функціональні та нефункціональні вимоги були виконані, а гра продемонструвала стабільність і готовність до подальшого розвитку.

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

1. Танчук Д.І., Тренъов М.Г. Розробка системи прогресу персонажа в умовах випадково згенерованого світу Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025, С.412 -414.

2. Танчук Д.І., Тренъов М.Г. Розробка модульної системи зброї для відеогри Rogue Decay на Unity з використанням ScriptableObject та об'єктно-орієнтованої архітектури Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2026, С.157 -159

ПЕРЕЛІК ПОСИЛАНЬ

1. Kushner D. Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture. New York : Random House Trade Paperbacks, 2004. 368 p.
2. Arsenault D. Guitar Hero: Making room for innovation in the retro-FPS genre. Games and Culture. 2009. Vol. 4, no. 2. P. 159–177. URL: <https://doi.org/10.1177/1555412008325481> (дата звернення: 31.05.2026).
3. Craddock D. L. Dungeon Hacks: How NetHack, Angband, and Other Roguelikes Ruled the Video Game World. Canton : Press Start Press, 2015. 238 p.
4. Togelius J., Shaker N., Nelson M. J. Procedural Content Generation in Games. Computational Intelligence in Games. Cham : Springer, 2016. P. 21–39.
5. Juul J. Handmade Pixels: Independent Video Games and the Quest for Authenticity. Cambridge : MIT Press, 2019. 248 p.
6. Rollings A., Adams E. Fundamentals of Game Design. New York : New Riders, 2021. 768 p.
7. Motion Twin. Dead Cells – Design Postmortem. GDC Vault. 2019. URL: <https://www.gdcvault.com/> (дата звернення: 31.05.2026).
8. Dodge Roll. Enter the Gungeon: Bullet Hell Design and Room Generation. GDC Talk. 2017. URL: <https://www.gdcvault.com/> (дата звернення: 31.05.2026).
9. Supergiant Games. Building Hades: Combining Narrative and Roguelite Structure. GDC Vault. 2021. URL: <https://www.gdcvault.com/> (дата звернення: 31.05.2026).
10. Zubek R. Elements of Game Design. Cambridge : MIT Press, 2020. 272 p.
11. O'Donnell C. Developer's Dilemma: The Secret World of Videogame Creators. Cambridge : MIT Press, 2014. 288 p.
12. Hocking C. Advanced Game Design with Unity. Boca Raton : CRC Press, 2021. 340 p.
13. Sanders R. Real-Time Rendering with Unreal Engine. Boca Raton : A K Peters/CRC Press, 2020. 412 p.

14. Godot Engine Documentation Team. Godot Engine: Architecture and Design. Godot Docs. 2023. URL: <https://docs.godotengine.org/> (дата звернення: 31.05.2026).
15. Unity Technologies. Unity Manual: Scripting in C#. Unity Docs. 2023. URL: <https://docs.unity3d.com/> (дата звернення: 31.05.2026).
16. Blender Foundation. Blender 3D: Documentation and Tools. Blender Docs. 2023. URL: <https://docs.blender.org/> (дата звернення: 31.05.2026).
17. Cockos Incorporated. REAPER User Guide. Cockos, 2022. 420 p.
18. Microsoft. C# Programming Guide. Microsoft Learn. 2023. URL: <https://learn.microsoft.com/dotnet/csharp/> (дата звернення: 31.05.2026).
19. Adams E. Fundamentals of Game Design. 3rd ed. New Riders, 2014. 576 p.
20. Nystrom R. Game Programming Patterns. Genever Benning, 2014. 354 p.
21. Unity Technologies. Create modular game architecture with ScriptableObjects. Unity Enterprise Backgrounder. 2020. URL: <https://resources.unity.com/> (дата звернення: 31.05.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



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



Розробка 3D відеогри "Rogue Decay" на рушії Unity в жанрі roguelike

Виконав студент 4 курсу
групи ПД-43

Танчук Денис Ігорович

Активация Керівник роботи

Читайте, чтобы активировать Windows, перейдите в раздел "Параметры".
асистент кафедри Тренъов Микита Григорович

Київ – 2026



МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення рівня зацікавленості гравця у процедурно згенерованих рівнях шляхом оптимізації FPS механік та впровадження динамічних моделей взаємодії, що створюють нетиповий для жанру ігровий досвід. Зокрема імплементувати системи динамічного руху гравця, моделі стрілецької взаємодії, алгоритми послідовної генерації рівнів, механізми прогресії та підсилення персонажа, а також технологічні засоби реалізації цих систем у ігровому рушії Unity.

Об'єкт дослідження: процеси формування ігрового досвіду в FPS грі із послідовною генерацією рівнів.

Предмет дослідження: механіки, методи та технології формування ігрового досвіду в FPS грі з покімнатною roguelike структурою,

Активация Windows
Чтобы активировать Windows, перейдите в раздел "Параметры".



ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести дослідження жанрових особливостей та ігрових механік відеоігор у жанрі FPS та roguelike.
2. Провести аналіз аналогів та ключових особливостей і механік відеоігор у цих жанрах.
3. Сформулювати функціональні та нефункціональні вимоги до програмного забезпечення.
4. Провести огляд існуючих програм для створення відеоігор у жанрі FPS та roguelike.
5. Розробити концепцію та архітектуру гри.
6. Реалізувати основні механіки: переміщення персонажу, збройова система, генерація рівня, вороги та боси, карткова прокачка характеристик героя.
7. Протестувати розроблену відеогру.



3

КОНЦЕПТ ГРИ

Сетинг – постапокаліптичне місто, в якому розповсюдився зомбі вірус

Сюжет – головний герой це детектив який розслідує причини появи зомбі вірусу у місті, приїхавши у самому центрі зарази герой прямує на військову базу, де він планує знайти відповіді на свої питання

Ключові механіки:

1. Стрільба з різних видів озброєння
2. Карткова прокачка характеристик
3. Випадкова генерація рівнів
4. Випадкова зброя зі скринь
5. Боси з фазами

Ключові особливості:

1. Поєднання жанрів FPS та roguelike
2. Карткова система прокачки характеристик героя
3. Випадкова генерація рівня з підготовлених префабів



Активация Windows
Чтобы активировать Windows, перейдите в раздел "Параметры".

4

ПОРІВНЯННЯ З АНАЛОГАМИ

Характеристики	Hades	Dead Cells	Enter the Gungeon	Rogue Decay
Динаміка бою	стильні комбо	швидкість	bullet-hell хаос	Реалістичні FPS-механіки
Структура рівнів	фіксовані біоми	коридори	кімнати, але простіші	3D кімнати
Унікальність жанру	класичний підхід	платформер	top-down шутер	FPS та roguelike
Взаємодія з простором	обмежена ізометрія	2D площина	плоскі арени	Вертикальність, 3D свобода

Активация Windows
Чтобы активировать Windows, перейдите в раздел "Параметры".



ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Забезпечення динамічного переміщення гравця в межах згенерованих покiмнатних локацiй.
2. Реалiзацiя базових та розширених FPS механiк, зокрема системи стрiльби та взаємодiї зi зброєю.
3. Впровадження системи карткової прокачки характеристик, що впливає на iгровий стиль та прогресiю.
4. Розробка механiзмiв збереження та вiдновлення даних гри.
5. Створення системи випадiння зброї зi скрині з урахуванням її рiдкості .
6. Створення алгоритму генерацiї рiвня за допомогою префабiв.

Нефункціональні вимоги:

1. Підтримка ОС не менше Windows 10.
2. Стабільність iгрового процесу з частотою кадрiв 60-100к/с.
- 3.Мінімальні системні вимоги: Процесор: Intel CORE I3-8100/AMD Ryzen 3 3300 X RAM: 8 гб
Відеокарта: NVIDIA GeForce GTX 1050/AMD Radeon RX 570.
- 4.Рекомендовані системні вимоги: Процесор: Intel CORE I5-8700/AMD Ryzen 5 3600 X RAM: 16 гб
Відеокарта: NVIDIA GeForce GTX 1060Ti/ AMD Radeon RX 5700 XT.

Активация Windows
Чтобы активировать Windows, перейдите в раздел "Параметры".

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



Blender



Unity



Visual Studio

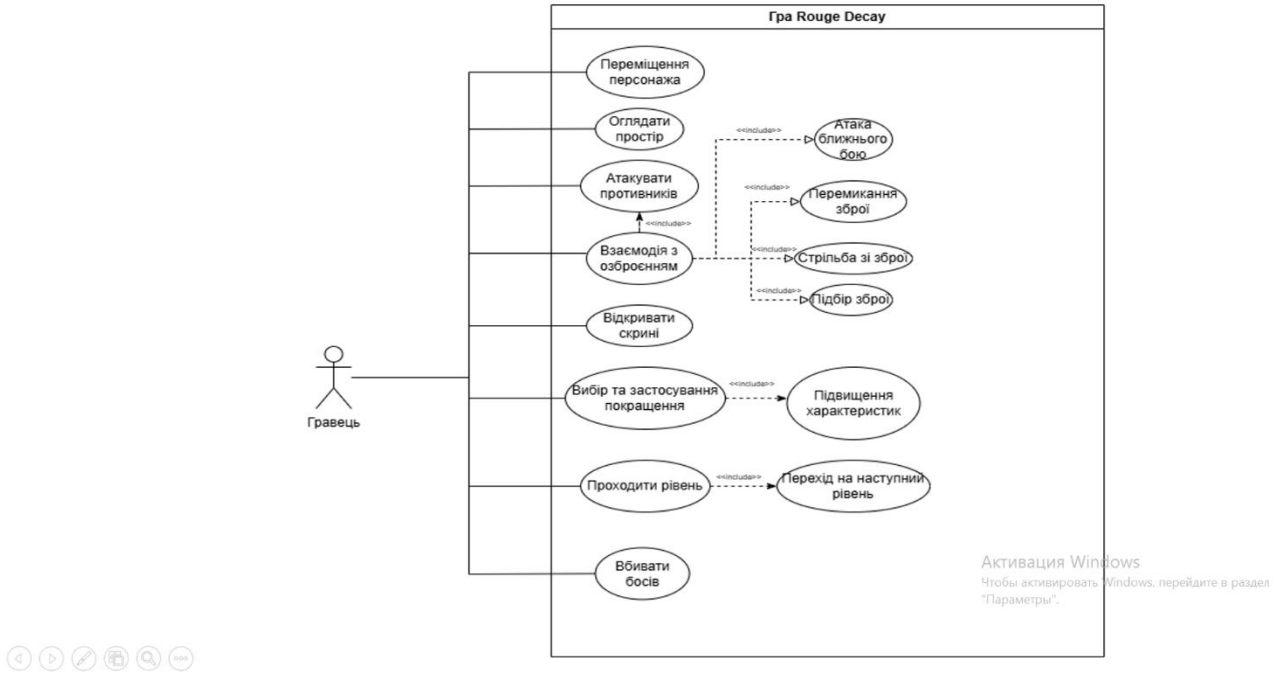


Reaper

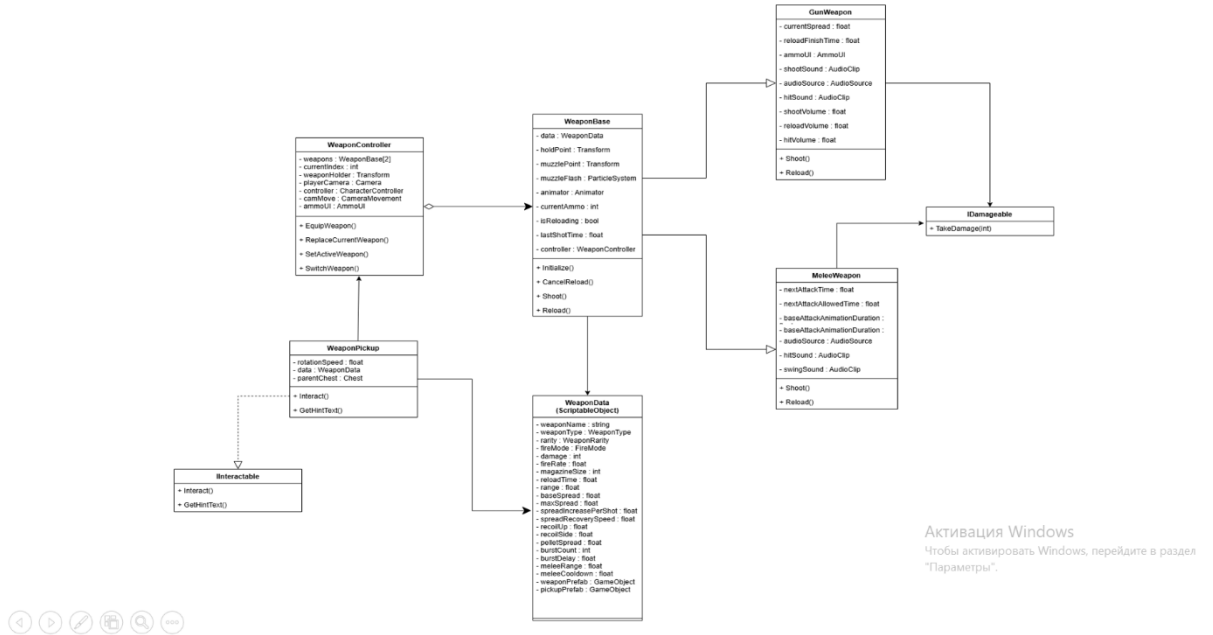
Активация Windows
Чтобы активировать Windows, перейдите в раздел "Параметры".

7

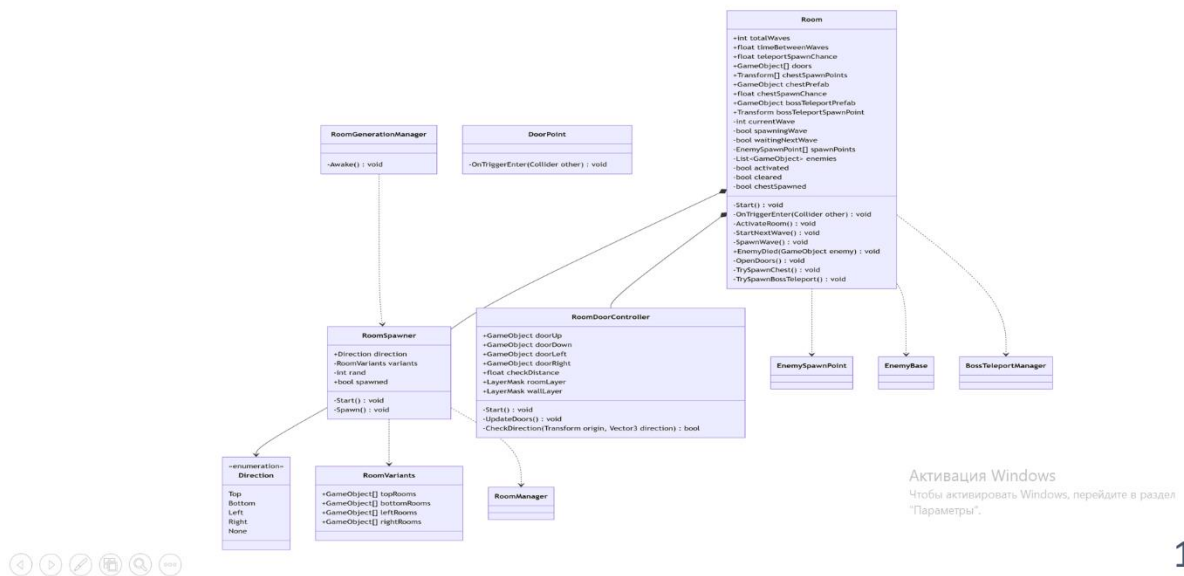
ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



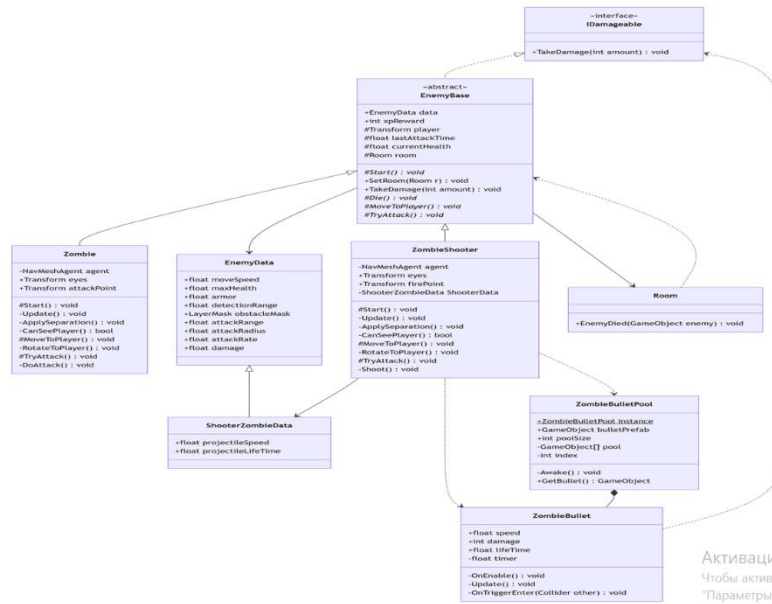
ДІАГРАМА КЛАСІВ СИСТЕМИ ЗБРОЇ



ДІАГРАМА КЛАСІВ ГЕНЕРАЦІЇ РІВНЯ

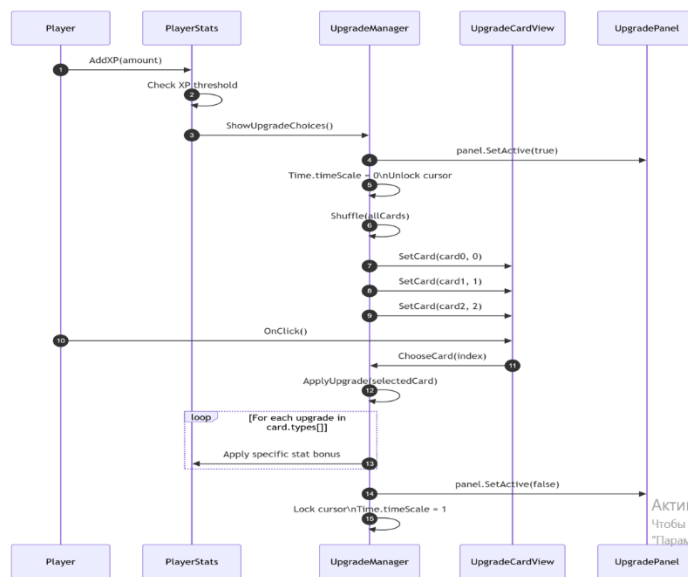


ДІАГРАМА КЛАСІВ ЗОМБІ ВОРОГІВ



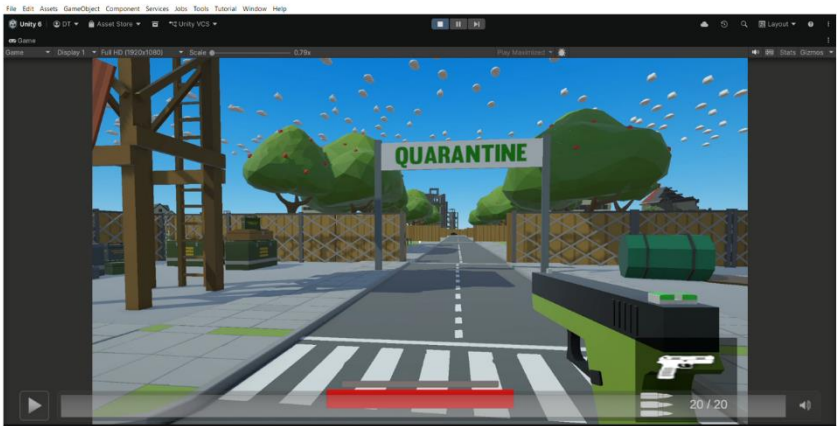
11

ДІАГРАМА ПОСЛІДОВНОСТІ СИСТЕМИ ПОКРАЩЕНЬ



12

ЕКРАННІ ФОРМИ



Вхід у кімнату з противниками

Активация Windows
Чтобы активировать Windows, перейдите в раздел "Параметры".



ЕКРАННІ ФОРМИ

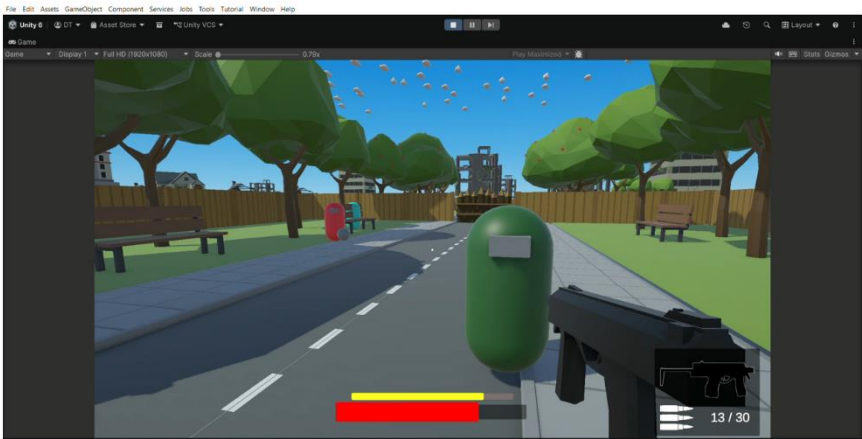


Відкриття кімнати після зачистки та поява скрині зі зброєю

Активация Windows
Чтобы активировать Windows, перейдите в раздел "Параметры".



ЕКРАННІ ФОРМИ



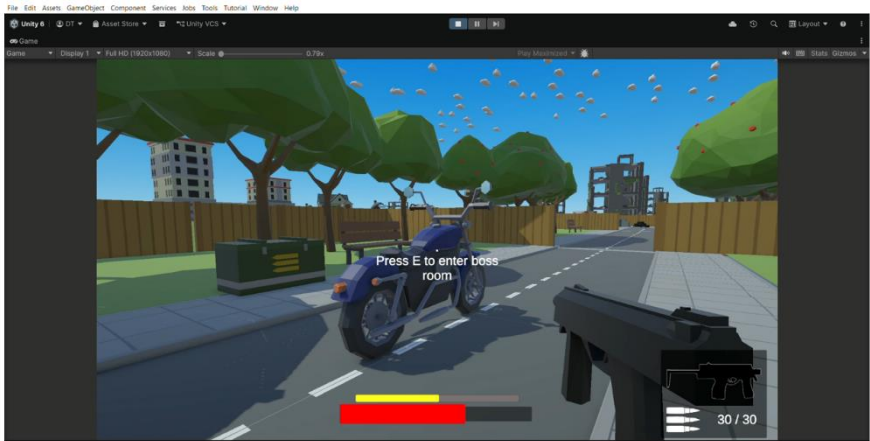
Покращення характеристик гравця через вибір карток з підсиленнями

Windows
Чтобы активировать Windows, перейдите в раздел "Параметры".



15

ЕКРАННІ ФОРМИ



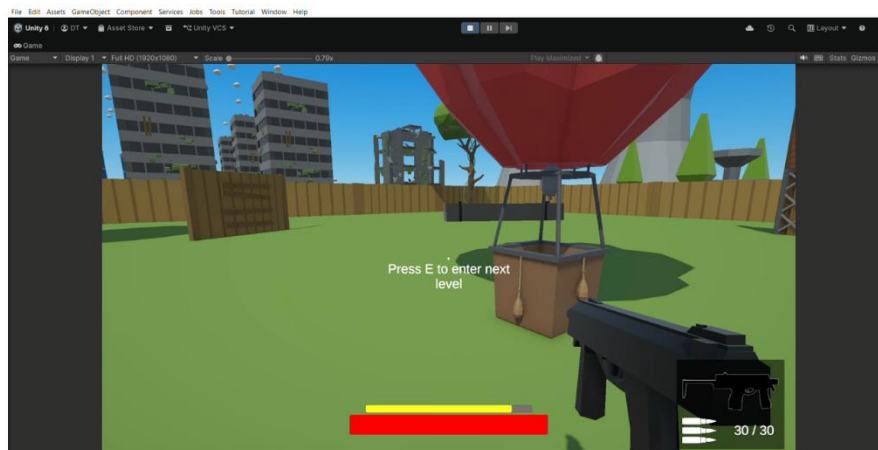
Телепорт до босса та битва з ним

Активация Windows
Чтобы активировать Windows, перейдите в раздел "Параметры".



16

ЕКРАННІ ФОРМИ



Телепорт на наступний рівень після битви з боссом

Активация Windows
Чтобы активировать Windows, перейдите в раздел
"Параметры".



17

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Танчук Д.І., Треньов М.Г. Розробка системи прогресу персонажа в умовах випадково згенерованого світу Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025, С.412 -414.
2. Танчук Д.І., Треньов М.Г. Розробка модульної системи зброї для відеогри Rogue Decay на Unity з використанням ScriptableObject та об'єктно-орієнтованої архітектури Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2026, С.157 -159

Активация Windows
Чтобы активировать Windows, перейдите в раздел
"Параметры".



18

ВИСНОВКИ

1. У ході роботи було проаналізовано ключові характеристики жанрів FPS та roguelike, визначено їхні спільні та відмінні риси, а також особливості, що впливають на формування ігрового досвіду. Це дозволило визначити набір механік, що є основними для поєднання цих жанрів у межах одного проекту.
2. На основі аналізу аналогів було визначено перелік функціональних вимог до гри: система руху, стрільби, генерації рівнів, поведінки ворогів, босів, карткової прокачки тощо. Також сформовано нефункціональні вимоги, що стосуються продуктивності та необхідного апаратного забезпечення.
3. Проаналізовано сучасні ігрові рушії та інструменти, зокрема Unity, Unreal Engine та Godot. На основі порівняння можливостей, продуктивності та доступності було обґрунтовано вибір Unity як оптимального середовища для реалізації проекту.
4. Реалізовано основні механіки гри: систему керування персонажем, збройову систему, генерацію рівнів, поведінку ворогів і босів, а також карткову систему прокачки характеристик героя.
5. Виконано тестування основних систем гри, перевірено коректність роботи генерації рівнів, поведінки ворогів, бою, прокачки та взаємодії між компонентами. Усі функціональні та нефункціональні вимоги були виконані, а гра продемонструвала стабільність і готовність до подальшого розвитку.



ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```

using UnityEngine;
using System.Collections;
using UnityEngine.SceneManagement;

public class GunWeapon : WeaponBase
{
    private float currentSpread;
    private AmmoUI ammoUI;

    private float reloadFinishTime = 0f;

    [Header("Audio")]
    public AudioSource audioSource;
    public AudioClip shootSound;
    public AudioClip reloadSound;
    public AudioClip hitSound;

    [Header("Audio Volumes")]
    [Range(0f, 1f)] public float shootVolume = 1f;
    [Range(0f, 1f)] public float reloadVolume = 1f;
    [Range(0f, 1f)] public float hitVolume = 0.15f;

    private void Start()
    {
        RebindUI();
        if (audioSource == null)
            audioSource = GetComponent<AudioSource>();
    }
    private void OnEnable()
    {
        SceneManager.sceneLoaded += OnSceneLoaded;
        isReloading = false;
        reloadFinishTime = 0f;
        currentSpread = 0f;
        lastShotTime = -GetFinalFireRate();
    }
    private void OnDisable()
    {
        ResetAnimatorState();
        SceneManager.sceneLoaded -= OnSceneLoaded;
    }

    private void OnSceneLoaded(Scene scene,
LoadSceneMode mode)
    {
        RebindUI();
    }

    private void RebindUI()
    {
        ammoUI = FindFirstObjectByType<AmmoUI>();
    }

    private void Update()
    {
        RecoverSpread();
    }

    public override void Shoot()
    {
        if (isReloading) return;

        if (Time.time < reloadFinishTime) return;
        if (Time.time - lastShotTime < GetFinalFireRate())
            return;

        if (currentAmmo <= 0)
        {
            Reload();
            return;
        }

        switch (data.fireMode)
        {
            case FireMode.Single:
            case FireMode.Auto:
                FireSingleShot();
                break;

            case FireMode.Burst:
                StartCoroutine(BurstRoutine());
                break;
        }

        private void FireSingleShot()
        {
            currentAmmo--;
            ammoUI?.UpdateAmmo(currentAmmo,
data.magazineSize);
            lastShotTime = Time.time;
            PlayShootAnimation();
            if (shootSound != null)
                audioSource.PlayOneShot(shootSound,
shootVolume);

            if (muzzleFlash != null)
                muzzleFlash.Play();

            Camera cam = controller.playerCamera;

            if (data.pellets > 1)
                FireShotgun(cam);
            else
                FireRay(cam);

            IncreaseSpread();
            controller.ApplyRecoil(
data.recoilUp,
Random.Range(-data.recoilSide, data.recoilSide)
);
        }

        private IEnumerator BurstRoutine()
        {
            if (Time.time - lastShotTime < GetFinalFireRate())
                yield break;

            int shots = data.burstCount;
            lastShotTime = Time.time;

            for (int i = 0; i < shots; i++)
            {
                if (currentAmmo <= 0) break;
                FireSingleShot();
            }
        }
    }

```

```

yield return new WaitForSeconds(data.burstDelay);
}
}
public override void Reload()
{
    if (isReloading) return;
    if (currentAmmo >= data.magazineSize) return;

    isReloading = true;

    float finalReloadTime = GetFinalReloadTime();
    reloadFinishTime = Time.time + finalReloadTime;

    PlayReloadAnimation();

    if (reloadSound != null)
        audioSource.PlayOneShot(reloadSound,
reloadVolume);

    StartCoroutine(ReloadRoutine());
}

private IEnumerator ReloadRoutine()
{
    yield return new WaitForSeconds(GetFinalReloadTime());

    currentAmmo = data.magazineSize;
    isReloading = false;

    ammoUI?.UpdateAmmo(currentAmmo,
data.magazineSize);
}

private void PlayShootAnimation()
{
    if (animator == null) return;

    float speed = 1f / data.fireRate;
    animator.SetFloat("ShootSpeed", speed);
    animator.SetTrigger("Shoot");
}

private void PlayReloadAnimation()
{
    if (animator == null) return;

    float speed = 1f / data.reloadTime;
    animator.SetFloat("ReloadSpeed", speed);
    animator.SetTrigger("Reload");
}

private void PlayHitSound()
{
    if (hitSound == null || audioSource == null)
        return;

    float originalPitch = audioSource.pitch;

    audioSource.pitch = Random.Range(0.9f, 1.1f);

    audioSource.PlayOneShot(hitSound, hitVolume);

    audioSource.pitch = originalPitch;
}

private void FireShotgun(Camera cam)
{
    for (int i = 0; i < data.pellets; i++)
    {
        Vector3 dir = GetShotgunDirection(cam.transform);
        if (Physics.Raycast(cam.transform.position, dir, out
RaycastHit hit, data.range))
        {
            if (hit.collider.TryGetComponent<IDamageable>(out
var target))
            {
                target.TakeDamage(GetFinalDamage());
                PlayHitSound();
                HitmarkerUI.Instance?.Show();
            }
        }
    }

    private Vector3 GetShotgunDirection(Transform
cam)
    {
        float spread = data.pelletSpread;
        float x = Random.Range(-spread, spread);
        float y = Random.Range(-spread, spread);
        return cam.TransformDirection(new Vector3(x, y,
1f));
    }

    private void FireRay(Camera cam)
    {
        Vector3 dir = GetSpreadDirection(cam.transform);
        if (Physics.Raycast(cam.transform.position, dir, out
RaycastHit hit, data.range))
        {
            if (hit.collider.TryGetComponent<IDamageable>(out
var target))
            {
                target.TakeDamage(GetFinalDamage());
                PlayHitSound();
                HitmarkerUI.Instance?.Show();
            }
        }
    }

    private Vector3 GetSpreadDirection(Transform cam)
    {
        float x = Random.Range(-currentSpread,
currentSpread);
        float y = Random.Range(-currentSpread,
currentSpread);
        return cam.TransformDirection(new Vector3(x, y,
1f));
    }

    private void IncreaseSpread()
    {
        currentSpread += data.spreadIncreasePerShot;
        currentSpread = Mathf.Clamp(currentSpread,
GetFinalSpread(), data.maxSpread);
    }

    private void RecoverSpread()
    {
        currentSpread = Mathf.Lerp(
currentSpread,

```

```

    GetFinalSpread(),
    Time.deltaTime * data.spreadRecoverySpeed
);
}

    private int GetFinalDamage()
    {
        return Mathf.RoundToInt(
            data.damage
            *
            PlayerStats.Instance.rangedDamageMultiplier
        );
    }

    private float GetFinalFireRate()
    {
        return
            data.fireRate
            /
            PlayerStats.Instance.fireRateMultiplier;
    }

    private float GetFinalReloadTime()
    {
        return
            data.reloadTime
            /
            PlayerStats.Instance.reloadSpeedMultiplier;
    }

    private float GetFinalSpread()
    {
        return
            data.baseSpread
            *
            PlayerStats.Instance.spreadMultiplier;
    }
}

public interface IInteractable
{
    void Interact();
    string GetHintText();
}

using UnityEngine;
using UnityEngine.SceneManagement;

public class MeleeWeapon : WeaponBase
{
    private float nextAttackTime = 0f;
    private float nextAttackAllowedTime = 0f;

    [Header("Animation")]
    [SerializeField] private float
baseAttackAnimationDuration = 0.45f;

    [Header("Audio")]
    public AudioSource audioSource;
    public AudioClip swingSound;
    public AudioClip hitSound;

    private void OnEnable()
    {
        nextAttackTime = 0f;
        nextAttackAllowedTime = 0f;
    }
    private void OnDisable()
    {
        ResetAnimatorState();
    }
    public override void Initialize(WeaponController
controller)
    {
        base.Initialize(controller);
        currentAmmo = 1;

        if (audioSource == null)
            audioSource = GetComponent<AudioSource>();
    }

    public override void Shoot()
    {
        if (Time.time < nextAttackAllowedTime)
            return;

        if (Time.time < nextAttackTime)
            return;

        float cooldown = GetMeleeCooldown();
        nextAttackTime = Time.time + cooldown;

        float animDurationFinal =
baseAttackAnimationDuration
            *
            PlayerStats.Instance.meleeAttackSpeedMultiplier;

        nextAttackAllowedTime = Time.time +
            Mathf.Max(cooldown, animDurationFinal);

        PlayAttackAnimation(cooldown);
        PlaySwingSound();
        DoRaycastAttack();
    }

    private void PlayAttackAnimation(float cooldown)
    {
        if (animator == null)
            return;

        float animSpeed = 1f / cooldown;

        animator.SetFloat("AttackSpeed", animSpeed);
        animator.SetTrigger("Attack");
    }

    private void PlaySwingSound()
    {
        if (audioSource != null && swingSound != null)
            audioSource.PlayOneShot(swingSound);
    }

    private void PlayHitSound()
    {
        if (audioSource != null && hitSound != null)
            audioSource.PlayOneShot(hitSound);
    }

    private void DoRaycastAttack()
    {
        float range = GetMeleeRange();

        Vector3 origin =
controller.playerCamera.transform.position;
        Vector3 direction =
controller.playerCamera.transform.forward;

        RaycastHit[] hits = Physics.RaycastAll(origin,
direction, range);

        foreach (var hit in hits)
        {
            if (hit.collider.CompareTag("Player"))
                continue;

```

```

        if (hit.collider.TryGetComponent<IDamageable>(out
var dmg))
        {
            int finalDamage = Mathf.RoundToInt(
                data.damage *
PlayerStats.Instance.meleeDamageMultiplier
            );

            dmg.TakeDamage(finalDamage);
            PlayHitSound();
            HitmarkerUI.Instance?.Show();
            return;
        }
    }

```

```

    }

    public override void Reload() { }
    private float GetMeleeCooldown()
    {
        return data.meleeCooldown /
PlayerStats.Instance.meleeAttackSpeedMultiplier;
    }

    private float GetMeleeRange()
    {
        return data.meleeRange *
PlayerStats.Instance.meleeRangeMultiplier;
    }

```

```

using UnityEngine;

public abstract class WeaponBase : MonoBehaviour
{
    [Header("Data")]
    public WeaponData data;

    [Header("Setup")]
    public Transform holdPoint;
    public Transform muzzlePoint;
    public ParticleSystem muzzleFlash;
    public Animator animator;

    protected int currentAmmo;
    protected bool isReloading;
    protected float lastShotTime;

    protected WeaponController controller;

    public virtual void Initialize(WeaponController controller)
    {
        this.controller = controller;
        currentAmmo = data.magazineSize;
    }

    public virtual void CancelReload()
    {
        isReloading = false;
        StopAllCoroutines();
    }

    protected virtual void OnDisable()
    {
        CancelReload();
    }

    public abstract void Shoot();
    public abstract void Reload();
}

```

```

protected void ResetAnimatorState()
{
    if (animator == null) return
    animator.Rebind()
}

public int CurrentAmmo => currentAmmo;
public int MaxAmmo => data.magazineSize;
public WeaponRarity Rarity => data.rarity;
public WeaponType WeaponType => data.weaponType;
public FireMode FireMode => data.fireMode;

public void SetAmmo(int value)
{
    currentAmmo = Mathf.Clamp(value, 0,
data.magazineSize);
}

using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections;

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

    public CameraMovement camMove;

    [Header("Refs")]
    public Transform weaponHolder;
    public Camera playerCamera;
    public CharacterController controller;

    [Header("Sway Settings")]
    public float swayAmount = 0.05f;
    public float swaySmooth = 8f;

    [Header("Bob Settings")]
    public float bobAmount = 0.05f;
    public float bobSpeed = 8f;

    private float bobTimer;
    private Vector3 initialWeaponPos;

    [Header("Jump Bob Settings")]
    public float jumpBobAmount = 0.2f;
    public float jumpBobSmooth = 6f;
    public float fallBobMultiplier = 1.2f;

    private float jumpBobOffset;
    private bool isGrounded;

    private WeaponBase[] weapons = new WeaponBase[2];
    private int currentIndex = -1;

    private AmmoUI ammoUI;
}

```

```

public System.Action<WeaponBase> onWeaponChanged;
public WeaponBase CurrentWeapon => (currentIndex >= 0 ?
weapons[currentIndex] : null);

private void Awake()
{
    if (Instance == null) Instance = this;
    else Destroy(gameObject);
}

private void Start()
{
    initialWeaponPos = weaponHolder.localPosition;
    RebindUI();
}

private void OnEnable()
{
    SceneManager.sceneLoaded += OnSceneLoaded;
}

private void OnDisable()
{
    SceneManager.sceneLoaded -= OnSceneLoaded;
}

private void OnSceneLoaded(Scene scene, LoadSceneMode
mode)
{
    RebindUI();
    onWeaponChanged?.Invoke(CurrentWeapon);
}

private void RebindUI()
{
    ammoUI = FindFirstObjectByType<AmmoUI>();
}

private void Update()
{
    if (Input.GetKeyDown(KeyCode.Alpha1))
SetActiveWeapon(0);
    if (Input.GetKeyDown(KeyCode.Alpha2))
SetActiveWeapon(1);

    if (currentIndex == -1) return;

    WeaponBase weapon = weapons[currentIndex];
    if (weapon == null) return;

    switch (weapon.FireMode)
    {
        case FireMode.Auto:
            if (Input.GetMouseButton(0)) weapon.Shoot();
            break;

        case FireMode.Single:
        case FireMode.Burst:
        case FireMode.Melee:
            if (Input.GetMouseButtonDown(0)) weapon.Shoot();
            break;
    }

    if (Input.GetKeyDown(KeyCode.R))
        weapon.Reload();

```

```

    if (Input.GetKeyDown(KeyCode.Q))
        SwitchWeapon();

    HandleSway();
    HandleBob();
    HandleJumpBob();
}

private void HandleSway()
{
    float mouseX = Input.GetAxis("Mouse X");
    float mouseY = Input.GetAxis("Mouse Y");

    Vector3 swayOffset = new Vector3(
        -mouseX * swayAmount,
        -mouseY * swayAmount,
        0f
    );

    weaponHolder.localPosition = Vector3.Lerp(
        weaponHolder.localPosition,
        initialWeaponPos + swayOffset,
        Time.deltaTime * swaySmooth
    );
}

private void HandleBob()
{
    float moveX = Input.GetAxis("Horizontal");
    float moveY = Input.GetAxis("Vertical");
    float speed = new Vector2(moveX, moveY).magnitude;

    if (speed > 0.1f)
    {
        bobTimer += Time.deltaTime * bobSpeed;

        float bobX = Mathf.Sin(bobTimer) * bobAmount;
        float bobY = Mathf.Cos(bobTimer * 2f) * bobAmount *
0.5f;

        weaponHolder.localPosition += new Vector3(bobX,
bobY, 0f);
    }
    else
    {
        bobTimer = 0f;
    }
}

private void HandleJumpBob()
{
    isGrounded = controller.isGrounded;

    if (!isGrounded)
    {
        jumpBobOffset = Mathf.Lerp(
            jumpBobOffset,
            -jumpBobAmount * fallBobMultiplier,
            Time.deltaTime * jumpBobSmooth
        );
    }
    else
    {
        jumpBobOffset = Mathf.Lerp(
            jumpBobOffset,
            0f,
            Time.deltaTime * jumpBobSmooth

```

```

    );
}

    weaponHolder.localPosition += new Vector3(0f,
jumpBobOffset, 0f);
}

public void EquipWeapon(GameObject prefab, int slot)
{
    if (slot < 0 || slot > 1) return;

    if (weapons[slot] != null)
        Destroy(weapons[slot].gameObject);

    GameObject instance = Instantiate(prefab, weaponHolder);
    instance.transform.localPosition = Vector3.zero;
    instance.transform.localRotation = Quaternion.identity;

    WeaponBase weapon =
instance.GetComponent<WeaponBase>();
    weapon.Initialize(this);

    weapons[slot] = weapon;

    StartCoroutine(ResetAnimatorNextFrame(weapon));

    if (currentIndex == -1)
        SetActiveWeapon(slot);
    else
        onWeaponChanged?.Invoke(CurrentWeapon);
}

public void ApplyRecoil(float up, float side)
{
    camMove.recoilOffset += new Vector2(up, side);
}

private void SetActiveWeapon(int slot)
{
    if (slot < 0 || slot > 1) return;

    if (currentIndex != -1 && weapons[currentIndex] != null)
        weapons[currentIndex].CancelReload();

    currentIndex = slot;

    WeaponBase weapon = weapons[slot];

    if (weapon != null)
        ammoUI?.UpdateAmmo(weapon.CurrentAmmo,
weapon.MaxAmmo);

    for (int i = 0; i < weapons.Length; i++)
    {
        if (weapons[i] != null)
            weapons[i].gameObject.SetActive(i == slot);
    }

    if (weapon != null)
        StartCoroutine(ResetAnimatorNextFrame(weapon));

    onWeaponChanged?.Invoke(weapon);
}

```

```

private void SwitchWeapon()
{
    int next = (currentIndex == 0) ? 1 : 0;

    if (weapons[currentIndex] != null)
        weapons[currentIndex].CancelReload();

    if (weapons[next] != null)
        SetActiveWeapon(next);
}

public void ReplaceCurrentWeapon(GameObject prefab)
{
    int slot = currentIndex;

    if (weapons[slot] != null)
        Destroy(weapons[slot].gameObject);

    GameObject instance = Instantiate(prefab, weaponHolder);
    instance.transform.localPosition = Vector3.zero;
    instance.transform.localRotation = Quaternion.identity;

    WeaponBase weapon =
instance.GetComponent<WeaponBase>();
    weapon.Initialize(this);

    weapons[slot] = weapon;

    StartCoroutine(ResetAnimatorNextFrame(weapon));

    SetActiveWeapon(slot);
}

public WeaponBase GetWeaponInSlot(int slot)
{
    if (slot < 0 || slot > 1) return null;
    return weapons[slot];
}

public int CurrentSlot => currentIndex;

public void ForceSetActiveWeapon(int slot)
{
    SetActiveWeapon(slot);
}

public void ResetWeapons()
{
    for (int i = 0; i < weapons.Length; i++)
    {
        if (weapons[i] != null)
        {
            Destroy(weapons[i].gameObject);
            weapons[i] = null;
        }
    }

    currentIndex = -1;

    ammoUI?.UpdateAmmo(0, 0);
    onWeaponChanged?.Invoke(null);
}

private IEnumerator ResetAnimatorNextFrame(WeaponBase
weapon)
{
}

```

```

        yield return null;

        if (weapon != null && weapon.animator != null)
        {
            weapon.animator.Rebind();
        }
    }
}

using UnityEngine;

[CreateAssetMenu(fileName = "WeaponData", menuName =
"Game/Weapon Data")]
public class WeaponData : ScriptableObject
{
    [Header("Meta")]
    public string weaponName;
    public WeaponType weaponType;
    public WeaponRarity rarity;
    public Sprite icon;

    [Header("Shooting")]
    public FireMode fireMode;
    public int damage;
    public float fireRate;
    public int magazineSize;
    public float reloadTime;
    public float range = 100f;

    [Header("Spread & Recoil")]
    public float baseSpread;
    public float maxSpread;
    public float spreadIncreasePerShot;
    public float spreadRecoverySpeed;

    public float recoilUp;
    public float recoilSide;
    public float recoilRecovery;

    [Header("Shotgun Settings")]
    public int pellets = 1;
    public float pelletSpread = 2f;

    [Header("Burst Settings")]
    public int burstCount = 3;
    public float burstDelay = 0.1f;

    [Header("Melee")]
    public float meleeRange = 2f;
    public float meleeCooldown = 0.5f;

    [Header("Prefabs")]
    public GameObject weaponPrefab;
    public GameObject pickupPrefab;
}

public enum WeaponRarity
{
    Common,
    Uncommon,
    Rare
}

public enum WeaponType
{
    Melee,
    Pistol,
    Rifle,

```

```

    SMG,
    Shotgun,
    Sniper,
    Special
}

public enum FireMode
{
    Single,
    Auto,
    Burst,
    Melee
}

using UnityEngine;

public class WeaponPickup : MonoBehaviour, IInteractable
{
    public float rotationSpeed = 90f;

    public WeaponData data;
    public Chest parentChest;

    private void Update()
    {
        transform.Rotate(0f, rotationSpeed * Time.deltaTime, 0f);
    }

    public void Interact()
    {
        WeaponController wc = WeaponController.Instance;

        if (wc == null)
            return;

        wc.ReplaceCurrentWeapon(data.weaponPrefab);

        if (parentChest != null)
            Destroy(parentChest.gameObject);

        Destroy(gameObject);
    }

    public string GetHintText()
    {
        return $"Press E to pick up {data.weaponName}";
    }
}

using UnityEngine;
using UnityEngine.SceneManagement;

public class RunManager : MonoBehaviour
{
    public static RunManager Instance;

    [Header("Scenes")]
    public string hubSceneName = "HUB";
    public string[] levelScenes = { "FirstLevel", "SecondLevel"
};

    private int currentLevelIndex = 0;

    private void Awake()
    {
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
            return;
        }
    }
}

```

```

    }

    Instance = this;
    DontDestroyOnLoad(gameObject);
}

public void StartRun()
{
    currentLevelIndex = 0;

SceneManager.LoadScene(levelScenes[currentLevelIndex]);
}

public void LoadNextLevel()
{
    currentLevelIndex++;

    if (currentLevelIndex >= levelScenes.Length)
    {
        SceneManager.LoadScene(hubSceneName);
        return;
    }

SceneManager.LoadScene(levelScenes[currentLevelIndex]);
}

public void HandleDeath()
{
    ResetStats();
    SceneManager.sceneLoaded += SpawnInHub;
    SceneManager.LoadScene(hubSceneName);
}

private void SpawnInHub(Scene scene, LoadSceneMode
mode)
{
    if (scene.name == hubSceneName)
    {
        var spawn =
FindFirstObjectByType<PlayerSpawnPoint>();
        spawn?.ForceSpawn();
        SceneManager.sceneLoaded -= SpawnInHub;
    }
}

private void ResetStats()
{
    var s = PlayerStats.Instance;
    if (s == null) return;

    s.level = 1;
    s.currentXP = 0;
    s.xpToNextLevel = 100;

    s.maxHealth = 100;
    s.currentHealth = 100;

    s.moveSpeed = 6f;

    s.rangedDamageMultiplier = 1f;
    s.fireRateMultiplier = 1f;
    s.reloadSpeedMultiplier = 1f;
    s.spreadMultiplier = 1f;

```

```

    s.meleeDamageMultiplier = 1f;

    s.meleeAttackSpeedMultiplier = 1f;
    s.meleeRangeMultiplier = 1f;

    s.dashPowerMultiplier = 1f;
    s.dashCooldownBonusPercent = 0f;
    s.dashDurationMultiplier = 1f;

    s.armorPercent = 0.2f;

    s.healAmountMultiplier = 1f;
    s.healIntervalMultiplier = 1f;
    s.healDelayMultiplier = 1f;
}

using UnityEngine;

public abstract class EnemyBase : MonoBehaviour,
IDamageable
{
    public EnemyData data;
    public int xpReward = 10;

    protected Transform player;
    protected float lastAttackTime;
    protected float currentHealth;

    protected Room room;

    protected virtual void Start()
    {
        player =
GameObject.FindGameObjectWithTag("Player").transform;
        currentHealth = data.maxHealth;
    }

    public void SetRoom(Room r)
    {
        room = r;
    }

    public virtual void TakeDamage(int amount)
    {
        float reduction = 100f / (100f + data.armor);
        int finalDamage =
Mathf.Max(1,
Mathf.RoundToInt(amount * reduction));

        currentHealth -= finalDamage;

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

    protected virtual void Die()
    {
        PlayerStats.Instance.AddXP(xpReward);

        if (room != null)
            room.EnemyDied(gameObject);

        Destroy(gameObject);
    }

    protected abstract void MoveToPlayer();
    protected abstract void TryAttack();

```

```

}

using UnityEngine;

[CreateAssetMenu(fileName = "EnemyData", menuName =
"Game/Enemy Data")]
public class EnemyData : ScriptableObject
{
    [Header("Stats")]
    public float moveSpeed = 2f;
    public float maxHealth = 50f;
    public float armor = 0f;

    [Header("Detection")]
    public float detectionRange = 15f;
    public LayerMask obstacleMask;

    [Header("Attack")]
    public float attackRange = 1.5f;
    public float attackRadius = 1f;
    public float attackRate = 1f;
    public float damage = 10f;
}
using UnityEngine;
using UnityEngine.AI;

public class Zombie : EnemyBase
{
    private NavMeshAgent agent;

    public Transform eyes;
    public Transform attackPoint;

    protected override void Start()
    {
        base.Start();

        agent = GetComponent<NavMeshAgent>();
        agent.speed = data.moveSpeed;

        agent.updateRotation = false;
        agent.acceleration = 100f;
        agent.angularSpeed = 720f;
        agent.stoppingDistance = data.attackRange * 0.8f;
    }

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

        ApplySeparation();

        if (CanSeePlayer())
        {
            MoveToPlayer();
            RotateToPlayer();
            TryAttack();
        }
        else
        {
            agent.ResetPath();
        }
    }

    private void ApplySeparation()
    {
        float radius = 1.2f;
        float minDistance = 0.55f;
        float maxForce = 1.5f;

```

```

        float baseStrength = 5f;

        Collider[] neighbors =
Physics.OverlapSphere(transform.position, radius);

        Vector3 separation = Vector3.zero;
        int count = 0;

        foreach (Collider col in neighbors)
        {
            if (col == null || col.transform == transform) continue;
            if (!col.CompareTag("Enemy")) continue;

            Vector3 diff = transform.position -
col.transform.position;
            float dist = diff.magnitude;

            if (dist < minDistance)
            {
                float strength = baseStrength * (1f - dist /
minDistance);
                separation += diff.normalized * strength;
                count++;
            }
        }

        if (count > 0)
        {
            separation /= count;
            separation.y = 0;

            separation = Vector3.ClampMagnitude(separation,
maxForce);

            agent.Move(separation * Time.deltaTime);
        }

        private bool CanSeePlayer()
        {
            Vector3 dir = (player.position - eyes.position).normalized;
            float dist = Vector3.Distance(eyes.position,
player.position);

            if (dist > data.detectionRange)
                return false;

            if (Physics.Raycast(eyes.position, dir, dist,
data.obstacleMask))
                return false;

            return true;
        }

        protected override void MoveToPlayer()
        {
            float dist = Vector3.Distance(transform.position,
player.position);

            if (dist > data.attackRange * 0.9f)
                agent.SetDestination(player.position);
            else
                agent.ResetPath();
        }

        private void RotateToPlayer()
        {

```

```

Vector3 dir = player.position - transform.position;
dir.y = 0f;

if (dir.sqrMagnitude < 0.001f)
    return;

Quaternion targetRot = Quaternion.LookRotation(dir);
transform.rotation = Quaternion.Slerp(transform.rotation,
targetRot, Time.deltaTime * 10f);
}

protected override void TryAttack()
{
    if (Time.time - lastAttackTime < data.attackRate)
        return;

    float dist = Vector3.Distance(transform.position,
player.position);
    if (dist > data.attackRange)
        return;

    lastAttackTime = Time.time;
    DoAttack();
}

private void DoAttack()
{
    Collider[] hits =
Physics.OverlapSphere(attackPoint.position,
data.attackRadius);

    foreach (Collider hit in hits)
    {
        if (hit.CompareTag("Player"))
        {
            if (hit.TryGetComponent<IDamageable>(out var
dmg))
                dmg.TakeDamage((int)data.damage);
        }
    }
}

using UnityEngine;

public class ZombieBullet : MonoBehaviour
{
    [Header("Bullet Settings")]
    public float speed = 8f;
    public int damage = 10;
    public float lifeTime = 5f;

    private float timer;

    private void OnEnable()
    {
        timer = 0f;
    }

    private void Update()
    {
        transform.position += transform.forward * speed *
Time.deltaTime;

        timer += Time.deltaTime;
        if (timer >= lifeTime)

```

```

        gameObject.SetActive(false);
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.isTrigger)
            return;

        if (other.CompareTag("Player"))
        {
            IDamageable dmg =
other.GetComponent<IDamageable>();
            if (dmg != null)
                dmg.TakeDamage(damage);
        }

        gameObject.SetActive(false);
    }
}

using UnityEngine;

public class ZombieBulletPool : MonoBehaviour
{
    public static ZombieBulletPool Instance;

    public GameObject bulletPrefab;
    public int poolSize = 50;

    private GameObject[] pool;
    private int index = 0;

    private void Awake()
    {
        Instance = this;

        pool = new GameObject[poolSize];

        for (int i = 0; i < poolSize; i++)
        {
            pool[i] = Instantiate(bulletPrefab);
            pool[i].SetActive(false);
        }
    }

    public GameObject GetBullet()
    {
        for (int i = 0; i < poolSize; i++)
        {
            index = (index + 1) % poolSize;

            if (!pool[index].activeInHierarchy)
            {
                return pool[index];
            }
        }

        GameObject extra = Instantiate(bulletPrefab);
        extra.SetActive(false);
        return extra;
    }
}

using UnityEngine;
using UnityEngine.AI;

public class ZombieShooter : EnemyBase
{

```

```

private NavMeshAgent agent;

public Transform eyes;
public Transform firePoint;

private ShooterZombieData ShooterData =>
(ShooterZombieData)data;

protected override void Start()
{
    base.Start();

    agent = GetComponent<NavMeshAgent>();
    agent.speed = data.moveSpeed;

    agent.updateRotation = false;
    agent.acceleration = 100f;
    agent.angularSpeed = 720f;
    agent.stoppingDistance = data.attackRange * 0.8f;
}

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

    ApplySeparation();

    if (CanSeePlayer())
    {
        MoveToPlayer();
        RotateToPlayer();
        TryAttack();
    }
    else
    {
        agent.ResetPath();
    }
}

private void ApplySeparation()
{
    float radius = 1.2f;
    float minDistance = 0.55f;
    float maxForce = 1.5f;
    float baseStrength = 5f;

    Collider[] neighbors =
Physics.OverlapSphere(transform.position, radius);

    Vector3 separation = Vector3.zero;
    int count = 0;

    foreach (Collider col in neighbors)
    {
        if (col == null || col.transform == transform) continue;
        if (!col.CompareTag("Enemy")) continue;

        Vector3 diff = transform.position -
col.transform.position;
        float dist = diff.magnitude;

        if (dist < minDistance)
        {
            float strength = baseStrength * (1f - dist /
minDistance);
            separation += diff.normalized * strength;
            count++;
        }
    }

    if (count > 0)
    {
        separation /= count;
        separation.y = 0;
        separation = Vector3.ClampMagnitude(separation,
maxForce);
        agent.Move(separation * Time.deltaTime);
    }

    private bool CanSeePlayer()
    {
        Vector3 dir = (player.position - eyes.position).normalized;
        float dist = Vector3.Distance(eyes.position,
player.position);

        if (dist > data.detectionRange)
            return false;

        if (Physics.Raycast(eyes.position, dir, dist,
data.obstacleMask))
            return false;

        return true;
    }

    protected override void MoveToPlayer()
    {
        float dist = Vector3.Distance(transform.position,
player.position);

        if (dist > data.attackRange * 0.9f)
            agent.SetDestination(player.position);
        else
            agent.ResetPath();
    }

    private void RotateToPlayer()
    {
        Vector3 dir = player.position - transform.position;
        dir.y = 0f;

        if (dir.sqrMagnitude < 0.001f)
            return;

        Quaternion targetRot = Quaternion.LookRotation(dir);
        transform.rotation = Quaternion.Slerp(transform.rotation,
targetRot, Time.deltaTime * 10f);
    }

    protected override void TryAttack()
    {
        if (Time.time - lastAttackTime < data.attackRate)
            return;

        float dist = Vector3.Distance(transform.position,
player.position);
        if (dist > data.attackRange)
            return;

        lastAttackTime = Time.time;
        Shoot();
    }

    private void Shoot()
    {
        GameObject bulletObj =
ZombieBulletPool.Instance.GetBullet();

```

```

        bulletObj.transform.position = firePoint.position;
        bulletObj.transform.rotation = firePoint.rotation;
        bulletObj.SetActive(true);

        ZombieBullet bullet =
        bulletObj.GetComponent<ZombieBullet>();
        bullet.damage = (int)data.damage;
        bullet.speed = ShooterData.projectileSpeed;
        bullet.lifeTime = ShooterData.projectileLifeTime;
    }
}

using UnityEngine;

public class BurstShooter : HumanEnemyBase
{
    public Transform firePoint;

    [Header("Burst Settings")]
    public int bulletsPerBurst = 5;
    public float timeBetweenBullets = 0.08f;
    public float burstCooldown = 1.2f;
    public float spread = 4f;
    public float bulletSpeed = 26f;

    private int bulletsShot = 0;
    private float nextShootTime = 0f;
    private bool isBursting = false;

    protected override void MoveToPlayer()
    {
        float dist = Vector3.Distance(transform.position,
        player.position);

        if (dist > data.attackRange)
        {
            agent.SetDestination(player.position);
        }
        else if (dist < data.attackRange * 0.7f)
        {
            Vector3 dir = (transform.position -
            player.position).normalized;
            agent.SetDestination(transform.position + dir * 1f);
        }
        else
        {
            agent.SetDestination(transform.position);
        }

        Vector3 lookDir = player.position - transform.position;
        lookDir.y = 0;

        if (lookDir.sqrMagnitude > 0.01f)
        {
            transform.rotation = Quaternion.Slerp(
                transform.rotation,
                Quaternion.LookRotation(lookDir),
                Time.deltaTime * 10f
            );
        }
    }
}

```

```

    protected override void TryAttack()
    {
        float dist = Vector3.Distance(transform.position,
        player.position);
        if (dist > data.attackRange)
            return;

        if (Time.time >= nextShootTime)
        {
            if (!isBursting)
            {
                isBursting = true;
                bulletsShot = 0;
            }

            Shoot();
            bulletsShot++;

            if (bulletsShot >= bulletsPerBurst)
            {
                isBursting = false;
                nextShootTime = Time.time + burstCooldown;
            }
            else
            {
                nextShootTime = Time.time + timeBetweenBullets;
            }
        }
    }

    private void Shoot()
    {
        Quaternion rot = firePoint.rotation * Quaternion.Euler(
            Random.Range(-spread, spread),
            Random.Range(-spread, spread),
            0f
        );

        GameObject obj = BulletPool.Instance.GetBullet();
        obj.transform.position = firePoint.position;
        obj.transform.rotation = rot;

        RaiderBullet b = obj.GetComponent<RaiderBullet>();
        b.damage = (int)data.damage;
        b.speed = bulletSpeed;
        b.lifeTime = 3f;
    }
}

using UnityEngine;
using UnityEngine.AI;

public abstract class HumanEnemyBase : EnemyBase
{
    protected NavMeshAgent agent;
    public Transform eyes;

    protected override void Start()
    {
        base.Start();

        agent = GetComponent<NavMeshAgent>();
        agent.speed = data.moveSpeed;
        agent.acceleration = 100f;
        agent.angularSpeed = 720f;
    }
}

```

```

    agent.updateRotation = false;
}

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

    if (CanSeePlayer())
    {
        MoveToPlayer();
        TryAttack();
    }
    else
    {
        agent.SetDestination(transform.position);
    }
}

protected bool CanSeePlayer()
{
    Vector3 dir = (player.position - eyes.position).normalized;
    float dist = Vector3.Distance(eyes.position,
player.position);

    if (dist > data.detectionRange)
        return false;

    if (Physics.Raycast(eyes.position, dir, dist,
data.obstacleMask))
        return false;

    return true;
}

```

```
using UnityEngine;
```

```

public class RaiderBullet : MonoBehaviour
{
    public float speed = 25f;
    public int damage = 10;
    public float lifeTime = 3f;

    private float timer;

    private void OnEnable()
    {
        timer = 0f;
    }

    private void Update()
    {
        transform.position += transform.forward * speed *
Time.deltaTime;

        timer += Time.deltaTime;
        if (timer >= lifeTime)
            BulletPool.Instance.ReturnBullet(gameObject);
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.isTrigger) return;

        if (other.CompareTag("Player"))
        {
            IDamageable dmg =
other.GetComponent<IDamageable>();

```

```

        if (dmg != null)
            dmg.TakeDamage(damage);
        }

        BulletPool.Instance.ReturnBullet(gameObject);
    }
}

using UnityEngine;

public class RaiderMelee : HumanEnemyBase
{
    public Transform attackPoint;

    protected override void MoveToPlayer()
    {
        if (agent == null) return;

        agent.SetDestination(player.position);

        Vector3 dir = player.position - transform.position;
        dir.y = 0;

        if (dir.sqrMagnitude > 0.01f)
        {
            Quaternion rot = Quaternion.LookRotation(dir);
            transform.rotation =
Quaternion.Slerp(transform.rotation, rot, Time.deltaTime *
10f);
        }
    }

    protected override void TryAttack()
    {
        if (Time.time - lastAttackTime < data.attackRate)
            return;

        float dist = Vector3.Distance(transform.position,
player.position);
        if (dist > data.attackRange)
            return;

        lastAttackTime = Time.time;

        Collider[] hits =
Physics.OverlapSphere(attackPoint.position,
data.attackRadius);

        foreach (Collider hit in hits)
        {
            if (hit.CompareTag("Player"))
            {
                IDamageable dmg =
hit.GetComponent<IDamageable>();
                if (dmg != null)
                    dmg.TakeDamage((int)data.damage);
            }
        }
    }
}

using UnityEngine;

public class RaiderPistol : HumanEnemyBase
{

```

```

public Transform firePoint;
public float spread = 2f;

protected override void MoveToPlayer()
{
    float dist = Vector3.Distance(transform.position,
player.position);

    if (dist > data.attackRange)
    {
        agent.SetDestination(player.position);
    }
    else if (dist < data.attackRange * 0.7f)
    {
        Vector3 dir = (transform.position -
player.position).normalized;
        agent.SetDestination(transform.position + dir * 1f);
    }
    else
    {
        agent.SetDestination(transform.position);
    }

    Vector3 lookDir = player.position - transform.position;
    lookDir.y = 0;

    if (lookDir.sqrMagnitude > 0.01f)
    {
        transform.rotation = Quaternion.Slerp(
            transform.rotation,
            Quaternion.LookRotation(lookDir),
            Time.deltaTime * 10f
        );
    }
}

protected override void TryAttack()
{
    if (Time.time - lastAttackTime < data.attackRate)
        return;

    float dist = Vector3.Distance(transform.position,
player.position);
    if (dist > data.attackRange)
        return;

    lastAttackTime = Time.time;
    Shoot();
}

private void Shoot()
{
    Quaternion rot = firePoint.rotation * Quaternion.Euler(
        Random.Range(-spread, spread),
        Random.Range(-spread, spread),
        0f
    );

    GameObject obj = BulletPool.Instance.GetBullet();
    obj.transform.position = firePoint.position;
    obj.transform.rotation = rot;

```

```

    RaiderBullet b = obj.GetComponent<RaiderBullet>();
    b.damage = (int)data.damage;
    b.speed = 30f;
    b.lifeTime = 3f;
}
}

using UnityEngine;

public class BossRoom : Room
{
    public BossSpawnPoint spawnPoint;
    [Header("Teleport After Boss")]
    public GameObject levelTeleportPrefab;
    public Transform levelTeleportSpawnPoint;

    private bool activated = false;
    private GameObject bossInstance;
    private IBoss boss;

    private void OnTriggerEnter(Collider other)
    {
        if (activated) return;
        if (!other.CompareTag("Player")) return;

        ActivateRoom();
    }

    private void ActivateRoom()
    {
        activated = true;

        bossInstance = spawnPoint.SpawnBoss();

        boss = bossInstance.GetComponent<IBoss>();
        if (boss == null)
        {
            Debug.LogError("BossRoom: Spawned boss does NOT
implement IBoss!");
            return;
        }

        BossHealthUI.Instance.Show(boss.MaxHealth);

        boss.SetRoom(this);
    }

    public void BossDied()
    {
        BossHealthUI.Instance.Hide();

        if (levelTeleportPrefab != null && levelTeleportSpawnPoint
!= null)
        {
            Instantiate(levelTeleportPrefab,
                levelTeleportSpawnPoint.position,
                levelTeleportSpawnPoint.rotation);
        }
    }
}

using UnityEngine;
using UnityEngine.AI;

public class FatZombieBoss : MonoBehaviour, IDamageable,
IBoss

```

```

{
    private BossRoom room;

    public void SetRoom(BossRoom r) => room = r;

    [Header("Base stats")]
    public int maxHealth = 300;
    public int currentHealth;

    public int MaxHealth => maxHealth;
    public int CurrentHealth => currentHealth;

    [Header("Phase thresholds")]
    [Range(0f, 1f)] public float phase2Start = 0.7f;
    [Range(0f, 1f)] public float phase3Start = 0.4f;

    private int currentPhase = 1;

    [Header("Refs")]
    public Transform firePoint;
    private Transform player;
    public NavMeshAgent agent;

    [Header("Bullet settings")]
    public int bulletsPerRing = 16;
    public float bulletSpeed = 8f;
    public int bulletDamage = 10;
    public float bulletLifeTime = 4f;
    public float ringFireCooldownPhase1 = 3f;
    public float ringFireCooldownPhase2 = 2f;
    public float ringFireCooldownPhase3 = 1f;
    public float ringRotationStep = 10f;

    private float lastRingTime;
    private float currentRingAngle;

    [Header("Summon settings")]
    public bool canSummon = true;
    public float summonCooldownPhase2 = 10f;
    public float summonCooldownPhase3 = 6f;
    public int summonCountPhase2 = 2;
    public int summonCountPhase3 = 4;
    public float summonRadius = 5f;
    public GameObject[] zombiePrefabs;

    private float lastSummonTime;

    [Header("Movement phase 3")]
    public float phase3MoveSpeed = 1.5f;

    private void Start()
    {
        currentHealth = maxHealth;
        BossHealthUI.Instance.Show(maxHealth);

        GameObject p =
        GameObject.FindGameObjectWithTag("Player");
        if (p != null) player = p.transform;

        if (agent != null) agent.isStopped = true;
    }

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

        UpdatePhase();
        HandleShooting();
        HandleSummoning();

```

```

        HandleMovement();
    }

    private void UpdatePhase()
    {
        float hpPercent = (float)currentHealth / maxHealth;

        if (hpPercent <= phase3Start && currentPhase != 3)
        {
            currentPhase = 3;
            agent.isStopped = false;
            agent.speed = phase3MoveSpeed;
        }
        else if (hpPercent <= phase2Start && currentPhase == 1)
        {
            currentPhase = 2;
        }
    }

    private void HandleShooting()
    {
        float cooldown = currentPhase switch
        {
            1 => ringFireCooldownPhase1,
            2 => ringFireCooldownPhase2,
            3 => ringFireCooldownPhase3,
            _ => ringFireCooldownPhase1
        };

        if (Time.time - lastRingTime < cooldown) return;

        ShootBulletRing();
        lastRingTime = Time.time;
    }

    private void ShootBulletRing()
    {
        currentRingAngle += ringRotationStep;

        for (int i = 0; i < bulletsPerRing; i++)
        {
            float angle = currentRingAngle + i * (360f /
            bulletsPerRing);
            Vector3 dir = Quaternion.Euler(0f, angle, 0f) *
            Vector3.forward;

            GameObject obj = BossBulletPool.Instance.GetBullet();
            obj.transform.position = firePoint.position;

            BossBullet bullet = obj.GetComponent<BossBullet>();
            bullet.Init(dir, bulletSpeed, bulletDamage,
            bulletLifeTime);
        }
    }

    private void HandleSummoning()
    {
        if (!canSummon) return;
        if (currentPhase < 2) return;

        float cooldown = currentPhase == 2 ?
        summonCooldownPhase2 : summonCooldownPhase3;
        int count = currentPhase == 2 ? summonCountPhase2 :
        summonCountPhase3;

        if (Time.time - lastSummonTime < cooldown) return;

        SummonZombies(count);
        lastSummonTime = Time.time;
    }

```

```

    }

    private void SummonZombies(int count)
    {
        for (int i = 0; i < count; i++)
        {
            Vector2 circle = Random.insideUnitCircle *
            summonRadius;
            Vector3 pos = transform.position + new
            Vector3(circle.x, 0f, circle.y);

            GameObject prefab = zombiePrefabs[Random.Range(0,
            zombiePrefabs.Length)];
            Instantiate(prefab, pos, Quaternion.identity);
        }
    }

    private void HandleMovement()
    {
        if (currentPhase < 3) return;
        agent.SetDestination(player.position);
    }

    public void TakeDamage(int amount)
    {
        currentHealth -= amount;
        currentHealth = Mathf.Max(0, currentHealth);

        BossHealthUI.Instance.UpdateHP(currentHealth);

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

    private void Die()
    {
        BossHealthUI.Instance.Hide();
        room?.BossDied();
        Destroy(gameObject);
    }
}

using UnityEngine;
using UnityEngine.AI;

public class ArmoredBanditBoss : MonoBehaviour,
IDamageable, IBoss
{
    private BossRoom room;

    public void SetRoom(BossRoom r) => room = r;

    [Header("Base stats")]
    public int maxHealth = 800;
    public int currentHealth;

    public int MaxHealth => maxHealth;
    public int CurrentHealth => currentHealth;

    [Range(0f, 0.95f)]
    [Header("Armor (0-0.95)")]
    public float armorPercent = 0.7f;

    [Header("Movement")]
    public float moveSpeed = 1.5f;
    public float stoppingDistance = 2.5f;

    [Header("Melee attack")]
    public int meleeDamage = 20;
    public float attackRange = 3f;

```

```

    public float attackCooldown = 2f;

    [Header("Summoning")]
    public bool canSummon = true;
    public float summonCooldown = 10f;
    public int summonCount = 3;
    public float summonRadius = 6f;
    public GameObject[] minionPrefabs;

    [Header("Refs")]
    public NavMeshAgent agent;
    private Transform player;

    private float lastAttackTime;
    private float lastSummonTime;

    private void Start()
    {
        currentHealth = maxHealth;
        BossHealthUI.Instance.Show(maxHealth);

        GameObject p =
        GameObject.FindGameObjectWithTag("Player");
        if (p != null) player = p.transform;

        if (agent == null) agent =
        GetComponent<NavMeshAgent>();

        agent.speed = moveSpeed;
        agent.stoppingDistance = stoppingDistance;
    }

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

        HandleMovement();
        HandleAttack();
        HandleSummoning();
    }

    private void HandleMovement()
    {
        agent.SetDestination(player.position);
    }

    private void HandleAttack()
    {
        if (Time.time - lastAttackTime < attackCooldown) return;

        float dist = Vector3.Distance(transform.position,
        player.position);
        if (dist > attackRange) return;

        IDamageable dmg =
        player.GetComponent<IDamageable>();
        if (dmg != null) dmg.TakeDamage(meleeDamage);

        lastAttackTime = Time.time;
    }

    private void HandleSummoning()
    {
        if (!canSummon) return;
        if (Time.time - lastSummonTime < summonCooldown)
        return;

        SummonMinions(summonCount);
        lastSummonTime = Time.time;
    }

```

```

    }

    private void SummonMinions(int count)
    {
        for (int i = 0; i < count; i++)
        {
            Vector2 circle = Random.insideUnitCircle *
summonRadius;
            Vector3 pos = transform.position + new
Vector3(circle.x, 0f, circle.y);

            GameObject prefab = minionPrefabs[Random.Range(0,
minionPrefabs.Length)];
            Instantiate(prefab, pos, Quaternion.identity);
        }
    }

    public void TakeDamage(int amount)
    {
        float multiplier = Mathf.Clamp01(1f - armorPercent);
        multiplier = Mathf.Max(multiplier, 0.05f);

        int finalDamage = Mathf.Max(1,
Mathf.RoundToInt(amount * multiplier));

        currentHealth -= finalDamage;
        currentHealth = Mathf.Max(0, currentHealth);

        BossHealthUI.Instance.UpdateHP(currentHealth);

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

    private void Die()
    {
        BossHealthUI.Instance.Hide();
        room?.BossDied();
        Destroy(gameObject);
    }
}

using UnityEngine;

public class MilitaryTurretBoss : MonoBehaviour,
IDamageable, IBoss
{
    private BossRoom room;

    public void SetRoom(BossRoom r) => room = r;

    [Header("Base stats")]
    public int maxHealth = 500;
    public int currentHealth;

    public int MaxHealth => maxHealth;
    public int CurrentHealth => currentHealth;

    [Header("Armor")]
    [Range(0f, 0.95f)]
    public float armorPercent = 0.6f;
    [Header("Machine Model")]
    public Transform machineModel;
    public Vector3 machineRotationOffset;

    [Header("Rotation")]
    public Transform turretModel;
    public float rotationSpeed = 4f;

    [Header("Rotation Offset")]

```

```

    public Vector3 rotationOffset;

    [Header("Shooting (MilitaryAK style)")]
    public Transform firePoint;

    public int bulletsPerMiniBurst = 3;
    public int miniBurstsPerAttack = 8;
    public float timeBetweenBullets = 0.07f;
    public float timeBetweenMiniBursts = 0.25f;
    public float burstCooldown = 1.2f;

    public float spread = 2f;
    public float bulletSpeed = 20f;
    public int bulletDamage = 10;
    public float bulletLifeTime = 3f;

    private int bulletsShot = 0;
    private int miniBurstsShot = 0;
    private float nextShootTime = 0f;
    private bool isBursting = false;

    [Header("Overheat System")]
    public int burstsBeforeOverheat = 3;
    public float overheatDuration = 4f;
    private int burstCounter = 0;
    private bool isOverheated = false;
    private float overheatTimer = 0f;

    [Header("Summoning")]
    public bool canSummon = true;
    public float summonCooldown = 12f;
    public int summonCount = 3;
    public float summonRadius = 6f;
    public GameObject[] minionPrefabs;

    private float lastSummonTime;

    private Transform player;

    private void Start()
    {
        currentHealth = maxHealth;
        BossHealthUI.Instance.Show(maxHealth);

        if (machineModel != null)
            machineModel.localRotation =
Quaternion.Euler(machineRotationOffset);

        GameObject p =
GameObject.FindGameObjectWithTag("Player");
        if (p != null)
            player = p.transform;
    }

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

        HandleRotation();
        HandleOverheat();
        HandleShooting();
        HandleSummoning();
    }

    private void HandleRotation()
    {

```

```

Vector3 dir = (player.position - turretModel.position);
dir.y = 0f;

if (dir.sqrMagnitude > 0.1f)
{
    Quaternion targetRot = Quaternion.LookRotation(dir);

    targetRot *= Quaternion.Euler(rotationOffset);

    turretModel.rotation = Quaternion.Lerp(
        turretModel.rotation,
        targetRot,
        rotationSpeed * Time.deltaTime
    );
}

private void HandleOverheat()
{
    if (!isOverheated) return;

    overheatTimer -= Time.deltaTime;

    if (overheatTimer <= 0f)
    {
        isOverheated = false;
        burstCounter = 0;
    }
}

private void HandleShooting()
{
    if (isOverheated) return;

    if (Time.time >= nextShootTime)
    {
        if (!isBursting)
        {
            isBursting = true;
            bulletsShot = 0;
            miniBurstsShot = 0;
        }

        ShootOneBullet();
        bulletsShot++;

        if (bulletsShot >= bulletsPerMiniBurst)
        {
            miniBurstsShot++;
            bulletsShot = 0;

            if (miniBurstsShot >= miniBurstsPerAttack)
            {
                isBursting = false;
                nextShootTime = Time.time + burstCooldown;

                burstCounter++;
                if (burstCounter >= burstsBeforeOverheat)
                {
                    isOverheated = true;
                    overheatTimer = overheatDuration;
                }
            }
        }
        else
        {

```

```

            nextShootTime = Time.time +
            timeBetweenMiniBursts;
        }
    }
    else
    {
        nextShootTime = Time.time + timeBetweenBullets;
    }
}

private void ShootOneBullet()
{
    Vector3 dir = (player.position -
    firePoint.position).normalized;

    dir = Quaternion.Euler(
        Random.Range(-spread, spread),
        Random.Range(-spread, spread),
        0f
    ) * dir;

    GameObject obj = BossBulletPool.Instance.GetBullet();
    obj.transform.position = firePoint.position;

    BossBullet bullet = obj.GetComponent<BossBullet>();
    bullet.Init(dir, bulletSpeed, bulletDamage,
    bulletLifeTime);
}

private void HandleSummoning()
{
    if (!canSummon) return;
    if (Time.time - lastSummonTime < summonCooldown)
    return;

    SummonMinions(summonCount);
    lastSummonTime = Time.time;
}

private void SummonMinions(int count)
{
    for (int i = 0; i < count; i++)
    {
        Vector2 circle = Random.insideUnitCircle *
        summonRadius;
        Vector3 pos = transform.position + new
        Vector3(circle.x, 0f, circle.y);

        GameObject prefab = minionPrefabs[Random.Range(0,
        minionPrefabs.Length)];
        Instantiate(prefab, pos, Quaternion.identity);
    }
}

public void TakeDamage(int amount)
{
    if (!isOverheated)
    return;

    float multiplier = 1f - armorPercent;
    multiplier = Mathf.Clamp(multiplier, 0.05f, 1f);

    int finalDamage = Mathf.RoundToInt(amount *
    multiplier);

    currentHealth -= finalDamage;

```

```

        currentHealth = Mathf.Max(0, currentHealth);

        BossHealthUI.Instance.UpdateHP(currentHealth);

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

    private void Die()
    {
        BossHealthUI.Instance.Hide();
        room?.BossDied();
        Destroy(gameObject);
    }
}

using UnityEngine;
using System.Collections;

public class PlayerMovment : MonoBehaviour
{
    public CharacterController characterController;
    public PlayerStats stats;

    [Header("Movement")]
    Vector3 moveVelocity;
    float verticalVelocity;

    [Header("Ground Check")]
    public Transform groundCheck;
    public float groundDistance = 0.4f;
    public LayerMask whatIsGround;
    bool isGrounded;
    bool wasGroundedLastFrame = true;

    [Header("Jump")]
    public float jumpHeight = 3f;
    public float gravity = -9.81f;

    [Header("Dash Base")]
    public float dashTime = 0.2f;
    public float dashCooldown = 0.75f;

    bool canDash = true;
    Vector3 dashVelocity = Vector3.zero;

    [Header("Audio Source")]
    public AudioSource audioSource;

    [Header("Footstep")]
    public AudioClip footstepSound;
    [Range(0f, 1f)] public float footstepVolume = 0.7f;
    public float footstepInterval = 0.4f;
    float footstepTimer = 0f;

    [Header("Jump / Land")]
    public AudioClip jumpSound;
    [Range(0f, 1f)] public float jumpVolume = 1f;

    public AudioClip landSound;
    [Range(0f, 1f)] public float landVolume = 0.8f;

    [Header("Dash")]
    public AudioClip dashSound;
    [Range(0f, 1f)] public float dashVolume = 0.9f;

    void Update()
    {
        GroundCheck();
        Move();
        Jump();
        ApplyGravity();

        HandleFootsteps();
        HandleLandingSound();

        if (Input.GetKeyDown(KeyCode.LeftShift) && canDash)
            StartCoroutine(Dash());

        Vector3 finalVelocity = moveVelocity + dashVelocity;
        finalVelocity.y = verticalVelocity;

        characterController.Move(finalVelocity *
            Time.deltaTime);
    }

    void Move()
    {
        float x = Input.GetAxis("Horizontal");
        float z = Input.GetAxis("Vertical");

        Vector3 move = transform.right * x + transform.forward *
            z;
        moveVelocity = move * stats.moveSpeed;
    }

    void Jump()
    {
        if (isGrounded && Input.GetButtonDown("Jump"))
        {
            verticalVelocity = Mathf.Sqrt(jumpHeight * -2f *
                gravity);

            if (jumpSound != null)
                audioSource.PlayOneShot(jumpSound,
                    jumpVolume);
        }
    }

    void GroundCheck()
    {
        isGrounded =
            Physics.CheckSphere(groundCheck.position, groundDistance,
                whatIsGround);

        if (isGrounded && verticalVelocity < 0)
            verticalVelocity = -2f;
    }

    void ApplyGravity()
    {
        verticalVelocity += gravity * Time.deltaTime;
    }

    IEnumerator Dash()
    {
        canDash = false;

        if (dashSound != null)
            audioSource.PlayOneShot(dashSound, dashVolume);

        Vector3 dashDirection;

        if (moveVelocity.sqrMagnitude > 0.1f)
            dashDirection = moveVelocity.normalized;
    }
}

```

```

else
    dashDirection = transform.forward;

    float dashPowerFinal = stats.dashPower *
stats.dashPowerMultiplier;
    float dashDurationFinal = dashTime *
stats.dashDurationMultiplier;
    float dashCooldownFinal = dashCooldown / (1f +
stats.dashCooldownBonusPercent);

    dashVelocity = dashDirection * dashPowerFinal;

    yield return new WaitForSeconds(dashDurationFinal);

    dashVelocity = Vector3.zero;

    yield return new WaitForSeconds(dashCooldownFinal);

    canDash = true;
}

void HandleFootsteps()
{
    if (!isGrounded)
        return;

    Vector3 horizontalVelocity = new
Vector3(moveVelocity.x, 0, moveVelocity.z);

    if (horizontalVelocity.magnitude > 0.1f)
    {
        footstepTimer -= Time.deltaTime;

        if (footstepTimer <= 0f)
        {
            PlayFootstep();
            footstepTimer = footstepInterval / stats.moveSpeed;
        }
    }
    else
    {
        footstepTimer = 0f;
    }
}

void PlayFootstep()
{
    if (footstepSound == null || audioSource == null)
        return;

    audioSource.pitch = Random.Range(0.9f, 1.1f);
    audioSource.PlayOneShot(footstepSound,
footstepVolume);
}

void HandleLandingSound()
{
    if (!wasGroundedLastFrame && isGrounded)
    {
        if (landSound != null)
            audioSource.PlayOneShot(landSound, landVolume);
    }

    wasGroundedLastFrame = isGrounded;
}
}

```

```

using UnityEngine;
using UnityEngine.SceneManagement;

public class PlayerInteract : MonoBehaviour
{
    public Camera cam;
    public float distance = 4f;
    public LayerMask interactMask;
    public KeyCode key = KeyCode.E;

    private IInteractable currentInteractable;
    private InteractUI interactUI;

    private void Start()
    {
        RebindUI();
        RebindCamera();
    }

    private void OnEnable()
    {
        SceneManager.sceneLoaded += OnSceneLoaded;
    }

    private void OnDisable()
    {
        SceneManager.sceneLoaded -= OnSceneLoaded;
    }

    private void OnSceneLoaded(Scene scene, LoadSceneMode
mode)
    {
        RebindUI();
        RebindCamera();
    }

    private void RebindUI()
    {
        interactUI = FindFirstObjectByType<InteractUI>();
    }

    void Update()
    {
        CheckInteractable();

        if (currentInteractable != null &&
Input.GetKeyDown(key))
        {
            currentInteractable.Interact();
        }
    }

    private void RebindCamera()
    {
        cam = Camera.main;
    }

    void CheckInteractable()
    {
        Ray ray = new Ray(cam.transform.position,
cam.transform.forward);

        if (Physics.Raycast(ray, out RaycastHit hit, distance,
interactMask))
        {
            IInteractable interactable =
hit.collider.GetComponentInParent<IInteractable>();

            if (interactable != null)
            {

```

```

        if (currentInteractable != interactable)
        {
            currentInteractable = interactable;

            if (interactUI != null)
            interactUI.ShowHint(interactable.GetHintText());
        }

        return;
    }
}

if (currentInteractable != null)
{
    currentInteractable = null;

    if (interactUI != null)
        interactUI.HideHint();
}
}

using UnityEngine;

public class CameraMovment : MonoBehaviour
{
    public float mouseSensitivity = 100f;
    public Transform playerBody;

    float xRotation;

    public Vector2 recoilOffset;
    Vector2 recoilVelocity;
    public float recoilRecoverySpeed = 10f;

    void Start()
    {
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }

    void Update()
    {
        float mouseX = Input.GetAxis("Mouse X") *
mouseSensitivity * Time.deltaTime;
        float mouseY = Input.GetAxis("Mouse Y") *
mouseSensitivity * Time.deltaTime;

        xRotation -= mouseY;
        xRotation = Mathf.Clamp(xRotation, -90f, 90f);

        playerBody.Rotate(Vector3.up * mouseX);

        recoilOffset = Vector2.SmoothDamp(
            recoilOffset,
            Vector2.zero,
            ref recoilVelocity,
            1f / recoilRecoverySpeed
        );
    }
}

```

```

        transform.localRotation = Quaternion.Euler(
            xRotation - recoilOffset.x,
            recoilOffset.y,
            0f
        );
    }
}

using UnityEngine;
using UnityEngine.SceneManagement;

public class PlayerStats : MonoBehaviour, IDamageable
{
    public static PlayerStats Instance;

    private void Awake()
    {
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
            return;
        }
        Instance = this;
    }

    public event System.Action<float, float> OnHealthChanged;
    public event System.Action<int, int> OnXPChanged;

    [Header("Health")]
    public int maxHealth = 100;
    public int currentHealth;
    [Range(0f, 0.95f)]
    public float armorPercent = 0.2f;

    [Header("Auto Heal (Out of Combat)")]
    public float healDelay = 5f;
    public int healAmount = 5;
    public float healInterval = 1f;

    private float lastDamageTime = -999f;
    private bool isHealing = false;

    [Header("Heal Upgrades")]
    public float healAmountMultiplier = 1f;
    public float healIntervalMultiplier = 1f;
    public float healDelayMultiplier = 1f;

    [Header("Movement")]
    public float moveSpeed = 6f;

    [Header("Dash")]
    public float dashPower = 20f;
    public float dashPowerMultiplier = 1f;
    public float dashCooldownBonusPercent = 0f;
    public float dashDurationMultiplier = 1f;

    [Header("Weapon Multipliers")]
    public float rangedDamageMultiplier = 1f;
    public float fireRateMultiplier = 1f;
    public float reloadSpeedMultiplier = 1f;
    public float spreadMultiplier = 1f;
    public float meleeDamageMultiplier = 1f;
}

```

```

[Header("Melee Extra")]
public float meleeAttackSpeedMultiplier = 1f;
public float meleeRangeMultiplier = 1f;

[Header("Experience")]
public int level = 1;
public int currentXP = 0;
public int xpToNextLevel = 100;

private XPBar xpBar;
private PlayerHealthBar healthBar;
private UpgradeManager upgradeManager;

private void Start()
{
    currentHealth = maxHealth;
    RebindUI();
    OnHealthChanged?.Invoke(currentHealth, maxHealth);
}

private void Update()
{
    TryStartAutoHeal();
}

private void OnEnable()
{
    SceneManager.sceneLoaded += OnSceneLoaded;
}

private void OnDisable()
{
    SceneManager.sceneLoaded -= OnSceneLoaded;
}

private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    RebindUI();
}

private void RebindUI()
{
    xpBar = FindFirstObjectByType<XPBar>();
    healthBar = FindFirstObjectByType<PlayerHealthBar>();
    upgradeManager = FindFirstObjectByType<UpgradeManager>();

    if (healthBar != null)
        OnHealthChanged?.Invoke(currentHealth, maxHealth);

    if (xpBar != null)
        OnXPChanged?.Invoke(currentXP, xpToNextLevel);
}

public void TakeDamage(int amount)
{
    float multiplier = 1f - armorPercent;
    multiplier = Mathf.Clamp(multiplier, 0.05f, 1f);

    int finalDamage = Mathf.RoundToInt(amount * multiplier);

    currentHealth -= finalDamage;

```

```

currentHealth = Mathf.Max(0, currentHealth);

lastDamageTime = Time.time;
isHealing = false;

OnHealthChanged?.Invoke(currentHealth, maxHealth);

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

private void TryStartAutoHeal()
{
    if (currentHealth <= 0) return;
    if (currentHealth >= maxHealth) return;

    if (currentHealth >= 30 && !isHealing) return;

    float finalDelay = healDelay * healDelayMultiplier;

    if (Time.time - lastDamageTime >= finalDelay)
    {
        if (!isHealing)
            StartCoroutine(AutoHealRoutine());
    }
}

private System.Collections.IEnumerator AutoHealRoutine()
{
    isHealing = true;

    while (currentHealth > 0 && currentHealth < maxHealth)
    {
        if (Time.time - lastDamageTime < healDelay * healDelayMultiplier)
        {
            isHealing = false;
            yield break;
        }

        int finalHeal = Mathf.RoundToInt(healAmount * healAmountMultiplier);
        currentHealth += finalHeal;
        currentHealth = Mathf.Min(currentHealth, maxHealth);

        OnHealthChanged?.Invoke(currentHealth, maxHealth);

        float finalInterval = healInterval * healIntervalMultiplier;
        yield return new WaitForSeconds(finalInterval);
    }

    isHealing = false;
}

public void Heal(int amount)
{
    currentHealth = Mathf.Min(maxHealth, currentHealth + amount);
    OnHealthChanged?.Invoke(currentHealth, maxHealth);
}

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

```

```

        Destroy(PlayerRoot.Instance.gameObject);
        SceneManager.LoadScene("GameOverScene");
    }

    public void AddXP(int amount)
    {
        if (xpBar == null)
            RebindUI();

        currentXP += amount;
        OnXPChanged?.Invoke(currentXP, xpToNextLevel);

        while (currentXP >= xpToNextLevel)
        {
            currentXP -= xpToNextLevel;
            LevelUp();
            OnXPChanged?.Invoke(currentXP, xpToNextLevel);
        }
    }

    public void LevelUp()
    {
        level++;
        xpToNextLevel = Mathf.RoundToInt(xpToNextLevel *
1.2f);

        if (upgradeManager == null)
            upgradeManager =
FindFirstObjectByType<UpgradeManager>();

        upgradeManager?.ShowUpgradeChoices();
    }

    public void AddMaxHealthPercent(float percent)
    {
        int add = Mathf.RoundToInt(maxHealth * percent);
        maxHealth += add;
        currentHealth += add;

        OnHealthChanged?.Invoke(currentHealth, maxHealth);
    }

    public void AddMoveSpeedPercent(float percent) =>
moveSpeed *= 1f + percent;
    public void AddRangedDamagePercent(float percent) =>
rangedDamageMultiplier *= 1f + percent;
    public void AddFireRatePercent(float percent) =>
fireRateMultiplier *= 1f + percent;
    public void AddReloadSpeedPercent(float percent) =>
reloadSpeedMultiplier *= 1f + percent;
    public void AddSpreadPercent(float percent) =>
spreadMultiplier *= 1f + percent;
    public void AddMeleeDamagePercent(float percent) =>
meleeDamageMultiplier *= 1f + percent;
    public void AddArmorPercent(float percent) =>
armorPercent = Mathf.Clamp01(armorPercent + percent);
    public void AddDashPowerPercent(float percent) =>
dashPowerMultiplier *= 1f + percent;
    public void AddDashCooldownPercent(float percent) =>
dashCooldownBonusPercent += percent;
    public void AddDashDurationPercent(float percent) =>
dashDurationMultiplier *= 1f + percent;
    public void AddMeleeAttackSpeedPercent(float percent) =>
meleeAttackSpeedMultiplier *= 1f + percent;
    public void AddMeleeRangePercent(float percent) =>
meleeRangeMultiplier *= 1f + percent;

    public void AddHealAmountPercent(float percent)
    {
        healAmountMultiplier *= 1f + percent;
    }

    public void AddHealIntervalPercent(float percent)
    {
        healIntervalMultiplier *= 1f - percent;
        healIntervalMultiplier = Mathf.Max(0.1f,
healIntervalMultiplier);
    }

    public void AddHealDelayPercent(float percent)
    {
        healDelayMultiplier *= 1f - percent;
        healDelayMultiplier = Mathf.Max(0.1f,
healDelayMultiplier);
    }
}

using UnityEngine;

public interface IDamageable
{
    void TakeDamage(int amount);
}

using UnityEngine;
using System.Collections.Generic;

public class Room : MonoBehaviour
{
    [Header("Wave Settings")]
    public int totalWaves = 3;
    public float timeBetweenWaves = 2f;

    private int currentWave = 0;
    private bool spawningWave = false;
    private bool waitingNextWave = false;

    [Header("Teleport Settings")]
    [Range(0f, 1f)]
    public float teleportSpawnChance = 0.3f;

    public GameObject[] doors;

    public Transform[] chestSpawnPoints;
    public GameObject chestPrefab;

    [Range(0f, 1f)]
    public float chestSpawnChance = 0.3f;

    public GameObject bossTeleportPrefab;
    public Transform bossTeleportSpawnPoint;

    private EnemySpawnPoint[] spawnPoints;
    private List<GameObject> enemies = new
List<GameObject>();

    private bool activated = false;
    private bool cleared = false;
    private bool chestSpawned = false;

    private void Start()
    {
        spawnPoints =
GetComponentInChildren<EnemySpawnPoint>();
    }
}

```

```

private void OnTriggerEnter(Collider other)
{
    if (activated || cleared)
        return;

    if (other.CompareTag("Player"))
    {
        ActivateRoom();
    }
}

private void ActivateRoom()
{
    activated = true;

    foreach (var door in doors)
        door.SetActive(true);

    StartNextWave();
}

private void StartNextWave()
{
    waitingNextWave = false;

    currentWave++;

    if (currentWave > totalWaves)
    {
        OpenDoors();
        return;
    }

    spawningWave = true;
    SpawnWave();
}

private void SpawnWave()
{
    enemies.Clear();

    foreach (var point in spawnPoints)
    {
        GameObject enemy = point.TrySpawn();

        if (enemy != null)
        {
            enemies.Add(enemy);
        }
    }

    enemy.GetComponent<EnemyBase>().SetRoom(this);
}

    spawningWave = false;

    if (enemies.Count == 0 && !waitingNextWave)
    {
        waitingNextWave = true;
        Invoke(nameof(StartNextWave), timeBetweenWaves);
    }
}

public void EnemyDied(GameObject enemy)
{
    enemies.Remove(enemy);

    if (enemies.Count == 0 && !spawningWave &&
!waitingNextWave)
{
    waitingNextWave = true;
    Invoke(nameof(StartNextWave), timeBetweenWaves);
}

private void OpenDoors()
{
    if (cleared) return;

    cleared = true;

    foreach (var door in doors)
        door.SetActive(false);

    TrySpawnChest();
    TrySpawnBossTeleport();
}

private void TrySpawnChest()
{
    if (chestSpawned) return;
    chestSpawned = true;

    if (Random.value > chestSpawnChance)
        return;

    if (chestSpawnPoints.Length == 0)
        return;

    int rand = Random.Range(0, chestSpawnPoints.Length);

    Instantiate(
        chestPrefab,
        chestSpawnPoints[rand].position,
        chestSpawnPoints[rand].rotation
    );
}

private void TrySpawnBossTeleport()
{
    if (spawnPoints.Length == 0)
        return;

    if (!BossTeleportManager.CanSpawnTeleport)
        return;

    if (Random.value > teleportSpawnChance)
        return;

    if (bossTeleportPrefab == null || bossTeleportSpawnPoint
== null)
        return;

    Instantiate(
        bossTeleportPrefab,
        bossTeleportSpawnPoint.position,
        bossTeleportSpawnPoint.rotation
    );

    BossTeleportManager.MarkTeleportSpawned();
}

}

using UnityEngine;

public class RoomDoorController : MonoBehaviour
{

```

```

public GameObject doorUp;
public GameObject doorDown;
public GameObject doorLeft;
public GameObject doorRight;

public float checkDistance = 4f;

public LayerMask roomLayer;
public LayerMask wallLayer;

private void Start()
{
    UpdateDoors();
}

private void UpdateDoors()
{
    if (doorUp != null)
    {
        bool hasRoom = CheckDirection(doorUp.transform,
Vector3.forward);
        doorUp.SetActive(!hasRoom);
    }

    if (doorDown != null)
    {
        bool hasRoom = CheckDirection(doorDown.transform,
Vector3.back);
        doorDown.SetActive(!hasRoom);
    }

    if (doorLeft != null)
    {
        bool hasRoom = CheckDirection(doorLeft.transform,
Vector3.left);
        doorLeft.SetActive(!hasRoom);
    }

    if (doorRight != null)
    {
        bool hasRoom = CheckDirection(doorRight.transform,
Vector3.right);
        doorRight.SetActive(!hasRoom);
    }
}

private bool CheckDirection(Transform origin, Vector3
direction)
{
    RaycastHit hit;

    int mask = roomLayer | wallLayer;

    if (Physics.Raycast(origin.position, direction, out hit,
checkDistance, mask))
    {
        if (((1 << hit.collider.gameObject.layer) & roomLayer)
!= 0)
            return true;

        if (((1 << hit.collider.gameObject.layer) & wallLayer) !=
0)
            Destroy(hit.collider.gameObject);
    }
}

```

```

        return false;
    }
}
using UnityEngine;

public class RoomSpawner : MonoBehaviour
{
    public Direction direction;

    public enum Direction
    {
        Top,
        Bottom,
        Left,
        Right,
        None
    }

    private RoomVariants variants;
    private int rand;
    public bool spawned = false;

    private void Start()
    {
        variants =
GameObject.FindGameObjectWithTag("Rooms").GetComponent<RoomVariants>();

        Spawn();
    }

    private void Spawn()
    {
        if (spawned) return;

        Vector3 pos = transform.position;

        if (RoomManager.occupiedPositions.Contains(pos))
        {
            spawned = true;
            return;
        }

        GameObject roomToSpawn = null;

        switch (direction)
        {
            case Direction.Top:
                roomToSpawn =
variants.topRooms[Random.Range(0,
variants.topRooms.Length)];
                break;
            case Direction.Bottom:
                roomToSpawn =
variants.bottomRooms[Random.Range(0,
variants.bottomRooms.Length)];
                break;
            case Direction.Left:
                roomToSpawn =
variants.leftRooms[Random.Range(0,
variants.leftRooms.Length)];
                break;
            case Direction.Right:
                roomToSpawn =
variants.rightRooms[Random.Range(0,
variants.rightRooms.Length)];
                break;
        }
    }
}

```

```

        break;
    }

    Instantiate(roomToSpawn,
roomToSpawn.transform.rotation);

    RoomManager.occupiedPositions.Add(pos);

    spawned = true;
}

using UnityEngine;

public class RoomVariants : MonoBehaviour
{
    public GameObject[] topRooms;
    public GameObject[] bottomRooms;
    public GameObject[] leftRooms;
    public GameObject[] rightRooms;

}

using UnityEngine;

public class EnemySpawnPoint : MonoBehaviour
{
    public GameObject[] possibleEnemies;

    [Range(0f, 1f)]
    public float spawnChance = 0.7f;

    public GameObject TrySpawn()
    {
        if (Random.value > spawnChance)
            return null;

        if (possibleEnemies.Length == 0)
            return null;

        int rand = Random.Range(0, possibleEnemies.Length);
        return Instantiate(possibleEnemies[rand],
transform.position, Quaternion.identity);
    }
}

using UnityEngine;

public class DoorPoint : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Wall"))
        {

            Destroy(other.gameObject);
        }
    }
}

using System.Collections.Generic;
using UnityEngine;

public class RoomManager : MonoBehaviour
{
    public static HashSet<Vector3> occupiedPositions = new
HashSet<Vector3>();

```

```

public Transform startRoom;

private void Awake()
{
    occupiedPositions.Clear();

    if (startRoom != null)
    {
        occupiedPositions.Add(startRoom.position);
    }
    else
    {
        Debug.LogWarning("RoomManager: StartRoom не
призначено!");
    }
}

```