МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інженерії
програмного забезпечення

_____ Євген ДАВИДЕНКО

«___» _____ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА
НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

Комп'ютерна 3D-гра у жанрі PVE Shooter на рушії Unity

Спеціальність 121 Інженерія програмного забезпечення
Освітня програма «Інженерія програмного забезпечення»

Здобувач _____ **Антон ПАВЛОВ**

«__» _____ 2026 р.

Керівник роботи _____ Марина ФАЛЕНКОВА
ст.викладачка

«__» _____ 2026 р.

Миколаїв – 2026

Завдання на виконання кваліфікаційної роботи

Чорноморський національний університет імені Петра Могили

Факультет	Комп’ютерних наук
Кафедра	Інженерії програмного забезпечення
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	121 Інженерія програмного забезпечення
Освітня програма	Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри інженерії
програмного забезпечення
_____ Євген ДАВИДЕНКО
«___» _____ 2026 р.

ЗАВДАННЯ

на кваліфікаційну магістерську роботу здобувача

Павлова Антон Дмитровича

(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи
Комп'ютерна 3D-гра у жанрі PVE Shooter на рушії Unity

Затверджена наказом ЧНУ ім. Петра Могили від «_26_» _грудня_ 2025 р. №_349_

2. Строк представлення кваліфікаційної роботи «___»_____2026 р.

3. Очікуваний результат роботи та початкові дані якщо такі потрібні

1. _____Очікуваним результатом є створення комп’ютерної 3D-гри у жанрі PVE Shooter на рушії Unity, яка поєднуватиме динамічний бойовий процес, систему розвитку персонажа, різноманітні типи ворогів та механіки покращення зброї.

2. Перелік питань, що підлягають розробці:

Провести аналіз предметної області та існуючих аналогів ігор жанру PVE Shooter, обрати програмні засоби та технології для розробки гри, спроєктувати структуру ігрових рівнів та основні механіки гри, реалізувати систему керування персонажем та механіку стрільби, розробити систему ворогів та бойову взаємодію між гравцем і супротивниками, розробити користувацький інтерфейс гри та головне меню.

3. Перелік графічних матеріалів

Презентація

4. Консультанти:

Консультант	Кафедра (організація)	Частина роботи

Дата видачі завдання «04» січень 2026 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: Комп'ютерна 3D-гра у жанрі PVE Shooter на рушії Unity

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Розробка та затвердження завдання на виконання КБР	26.12.2025	04.01.2026	виконано
2.	Огляд літератури за темою роботи	10.01.2026	19.01.2026	виконано
3.	Складання календарного плану КБР	20.01.2026	21.01.2026	виконано
4.	Аналіз предметної області	22.01.2026	10.02.2026	виконано
5.	Розробка проєктних рішень	11.02.2026	05.03.2026	виконано
6.	Моделювання та конструювання ПЗ	06.03.2026	25.03.2026	виконано
7.	Кодування, тестування та апробація розробленого ПЗ, аналіз результатів тестування, розробка керівництва користувача	10.04.2026	24.05.2026	виконано
8.	Відгук керівника КБР	25.05.2026	25.05.2026	виконано
9.	Оформлення КБР та презентації	20.05.2026	25.05.2026	виконано
10.	Попередній захист	27.05.2026	27.05.2026	виконано
11.	Рецензування			
12.	Завершення оформлення КБР та презентації			
13.	Захист кваліфікаційної роботи	.		

Здобувач _____ Антон ПАВЛОВ

«___» _____ 20___ р.

Керівник роботи
ст. викладачка

_____ Марина ФАЛЕНКОВА

«___» _____ 20___ р.

АНОТАЦІЯ

до кваліфікаційної бакалаврської роботи

«Комп'ютерна 3D-гра у жанрі PVE Shooter на рушії Unity»

Здобувач 409 гр.: Павлов Антон

Керівник: ст. викладачка Фаленкова Марина

Актуальність роботи обумовлена популярністю комп’ютерних 3D-ігор та активним розвитком ігрової індустрії. Сучасні користувачі очікують від ігор якісної графіки, динамічного геймплею та великої кількості інтерактивних механік. Особливу популярність сьогодні мають ігри жанру PVE Shooter, у яких гравець проходить бойові локації, знищує ворогів, покращує персонажа та отримує нові можливості для зброї.

Мета роботи полягає у розробці комп’ютерної 3D-гри на платформі Unity з реалізацією динамічного бойового процесу, системи розвитку персонажа та механік покращення зброї.

Об’єктом дослідження є процес розробки комп’ютерної 3D-гри у жанрі PVE Shooter на рушії Unity.

Предметом дослідження є методи, технології та програмні засоби створення комп’ютерних 3D-ігор, реалізація FPS-механік, системи прокачування персонажа та модифікацій зброї.

Кваліфікаційна робота складається із вступу, 4 розділів, висновків та переліку джерел посилання.

У вступі обґрунтовано актуальність обраної теми, визначено мету, основні завдання, об'єкт і предмет дослідження, а також розкрито практичне значення отриманих результатів.

У першому розділі розглядається актуальність створення ігор жанру **PVE Shooter**, виконується аналіз наявних аналогів, визначаються основні вимоги до майбутнього проєкту та формується технічне завдання для його розробки.

Другий розділ присвячено обґрунтуванню вибору засобів розробки, зокрема інструментів, фреймворків, мови програмування та програмного забезпечення, необхідного для моделювання й реалізації гри.

У третьому розділі здійснюється моделювання ключових ігрових процесів. У межах цього розділу розробляються UML-діаграми, діаграми станів, переходів і компонентів, а також описуються алгоритми реалізації основних ігрових механік.

Четвертий розділ охоплює програмну реалізацію гри, зокрема створення елементів користувацького інтерфейсу, розробку механіки рівнів, інтеграцію елементів штучного інтелекту та тестування готового прототипу.

КБР викладена на 74 сторінках, вона містить 4 розділів, 32 ілюстрацій, 7 таблиць, 15 джерел у переліку посилань.

Ключові слова: *3D-гра, FPS, PVE Shooter, Unity, геймплей, ігровий рушій, комп'ютерна гра, модифікації зброї, розробка ігор, система прокачування.*

ABSTRACT

to the bachelor's qualification work

"Computer 3D game in the PVE shooter on the Unity engine"

Student of group 409: Pavlov Anton

Supervisor: Senior Lecturer Falenkova Maryna

The relevance of the work is determined by the popularity of computer 3D games and the active development of the game industry. Modern users expect games to have high-quality graphics, dynamic gameplay, and a large number of interactive mechanics. Games in the PVE Shooter genre are especially popular today, where the player completes combat locations, destroys enemies, improves the character, and gains new weapon capabilities.

The purpose of the work is to develop a computer 3D game on the Unity platform with the implementation of dynamic combat gameplay, a character development system, and weapon upgrade mechanics.

The object of the study is the process of developing a computer 3D game in the PVE Shooter genre on the Unity engine.

The subject of the study includes methods, technologies, and software tools for creating computer 3D games, as well as the implementation of FPS mechanics, a character progression system, and weapon modification mechanics.

The bachelor's qualification work consists of an introduction, 4 chapters, conclusions, and a list of references.

The introduction substantiates the relevance of the chosen topic, defines the purpose, main tasks, object and subject of the study, and also reveals the practical significance of the obtained results.

The first chapter examines the relevance of creating games in the PVE Shooter genre, analyzes existing analogues, defines the main requirements for the future project, and forms the technical specification for its development.

The second chapter is devoted to justifying the choice of development tools, including instruments, frameworks, programming language, and software required for modeling and implementing the game.

The third chapter presents the modeling of key game processes. Within this chapter, UML diagrams, state diagrams, transition diagrams, component diagrams, and algorithms for implementing the main game mechanics are developed.

The fourth chapter covers the software implementation of the game, including the creation of user interface elements, development of level mechanics, integration of artificial intelligence elements, and testing of the completed prototype.

The bachelor's qualification work is presented on 74 pages and contains 4 chapters, 32 illustrations, 7 tables, and 15 sources in the list of references.

Keywords: 3D game, computer game, FPS, game development, game engine, gameplay, progression system, PVE Shooter, Unity, weapon modifications.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ **4**

ВСТУП **5**

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ **7**

1.1 Актуальність предметної сфери **7**

1.2 Огляд аналогів **8**

1.3 Визначення технічного завдання для ігрового застосунку **12**

1.4 Опис загального алгоритму виконання проєкту **14**

Висновки до розділу 1 ..... **16**

2 АНАЛІЗ ФРЕЙМВОРКІВ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ **17**

2.1 Вибір рушія для розробки ігрового застосунку **17**

2.2 Вибір мови програмування для розробки **22**

2.3 Вибір застосунку для моделювання **24**

2.4 Специфікація вимог **27**

Висновки до розділу 2 ..... **29**

3 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ **30**

3.1 Написання usecase **30**

3.2 Створення діаграми використання **34**

3.3 Побудова діаграм взаємодії (послідовності та кооперації) **37**

3.4 Діаграми станів та переходів **39**

3.5 Діаграма діяльності **42**

3.6 Розробка діаграм компонентів та розгортання **46**

Висновки до розділу 3 ..... **49**

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ **50**

4.1 Створення дизайну ігрового застосунку **50**

4.2 Програмна реалізація UI-елементів до застосунку **55**

4.3 Програмна реалізація основних механік **58**

ВИСНОВКИ **72**

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 73
ДОДАТОК А 75
ДОДАТОК Б 77
ДОДАТОК В 79
ДОДАТОК Г 83
ДОДАТОК Д 89

ПЕРЕЛІК СКОРОЧЕНЬ

ЛКМ – Ліва кнопка миші

ОС – операційна система

ПЗ – програмне забезпечення

ПК – персональний комп’ютер

ПКМ – Права кнопка миші

GDD – Game Design Document

NPC – Non-player character

PVE – Player versus Environment

ВСТУП

Актуальність теми полягає в тому, що в сучасному світі комп'ютерні 3D-ігри займають важливе місце в індустрії розваг і програмного забезпечення. Завдяки розвитку технологій та збільшенню потужності комп'ютерних систем, сучасні ігри стають дедалі більш реалістичними та динамічними. Особливо популярними серед користувачів залишаються ігри жанру PVE Shooter, у яких гравець проходить бойові локації, бореться з великою кількістю ворогів та покращує власного персонажа.

Ігри жанру PVE Shooter поєднують швидкий темп геймплею, систему розвитку персонажа, динамічний бойовий процес та використання різноманітної зброї. Такі проєкти дозволяють користувачам отримувати новий ігровий досвід завдяки прокачуванню персонажа, відкриттю нових можливостей та покращенню характеристик зброї.

Розробка 3D-ігор на платформі Unity стала одним із найпопулярніших напрямків серед незалежних розробників та ігрових студій. Unity надає потужний набір інструментів для створення 3D-графіки, фізики, анімацій, систем частинок та користувацьких інтерфейсів. Крім того, рушій дозволяє створювати кросплатформні проєкти для ПК та інших платформ.

Особливістю майбутнього проєкту є система отримання досвіду за знищення ворогів, прокачування характеристик персонажа та отримання спеціальних модифікацій зброї після перемоги над босами. Наприклад, зброя може отримувати ефекти замороження, отрути або вогняних куль.

Мета: створення 3D-гри на платформі Unity з реалізацією динамічного бойового процесу, системи розвитку персонажа та механік покращення зброї.

Для досягнення поставленої цілі необхідно виконати наступні **завдання:**

- провести аналіз сучасних технологій у сфері розробки ігор;
- огляд аналогів жанру PVE Shooter;
- визначити основні функції майбутньої гри;
- оформлення специфікацій вимог до ПЗ;

- створення загального алгоритму розробки гри.

Об'єкт роботи: розробка комп'ютерної 3D-гри у жанрі PVE Shooter на рушії Unity.

Предмет роботи: аналіз технологій та інструментів для створення комп'ютерної 3D-гри жанру PVE Shooter.

Сфера застосування: комп'ютерні ігри жанру PVE Shooter на платформі Unity надають гравцям можливість зануритися у динамічний бойовий процес, де важливу роль відіграють швидка реакція, точність та розвиток персонажа. Важливим аспектом таких ігор є не лише створення якісного геймплею, але й реалізація систем прогресії та покращення зброї.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність предметної сфери

Комп'ютерні 3D-ігри вже багато років залишаються однією з найпопулярніших форм цифрових розваг. Завдяки високій продуктивності сучасних ПК та потужним графічним процесорам ігрова індустрія продовжує активно розвиватися. Сучасні користувачі очікують від ігор високоякісної графіки, динамічного геймплею та великої кількості інтерактивних механік.

Особливу популярність сьогодні мають ігри жанру PVE Shooter. Основною особливістю таких ігор є проходження бойових локацій, боротьба з ворогами та система розвитку персонажа. Гравець повинен швидко реагувати на небезпеку, використовувати різні типи зброї та адаптуватися до різних типів супротивників.

У сучасних PVE Shooter іграх широко використовуються механіки прокачування. За знищення ворогів гравець отримує досвід, який можна використовувати для покращення характеристик персонажа, збільшення запасу здоров'я, швидкості пересування або сили зброї.

Також у таких іграх популярними є системи модифікацій зброї. Після перемоги над босами користувач може отримувати спеціальні ефекти для зброї, наприклад замороження ворогів, отруйні кулі або ефекти підпалювання.

Unity є одним із найпопулярніших рушіїв для створення таких проєктів. Він дозволяє реалізовувати FPS-механіки, фізику куль, систему частинок, анімації зброї та різноманітні візуальні ефекти [1].

Створення комп’ютерної 3D-ігри

Створення комп’ютерних 3D-ігор в наші часи є одним із найпопулярніших напрямків у сфері розробки програмного забезпечення та цифрових розваг. Завдяки розвитку сучасних технологій розробники отримують можливість створювати великі та деталізовані ігрові світи з великою кількістю механік та інтерактивних елементів. Саме 3D-ігри дозволяють користувачам отримати більш реалістичний та захоплюючий ігровий досвід завдяки якісній графіці, анімаціям та динамічному геймплею.

Однією з найпопулярніших платформ для створення 3D-ігор є Unity. Даний рушій дозволяє реалізовувати складні ігрові механіки, працювати з фізикою, освітленням, анімаціями та системами частинок. Крім того, Unity має велику кількість готових інструментів і ресурсів, що значно спрощує процес розробки гри.

Під час створення комп’ютерної 3D-гри важливу роль відіграє не тільки графічна складова, але й сам ігровий процес. Для жанру PVE Shooter особливо важливими є динамічні бойові сцени, система ворогів, різноманітність зброї та система прокачування персонажа. Саме ці елементи формують основний ігровий процес та впливають на зацікавленість користувача під час проходження гри.

У сучасних PVE Shooter іграх також активно використовуються системи модифікації зброї та покращення характеристик персонажа. Наприклад, після перемоги над босами гравець може отримувати спеціальні ефекти для зброї, які змінюють стиль проходження гри та роблять геймплей більш різноманітним.

Окрім створення основних механік гри, важливим етапом є оптимізація проєкту. Гра повинна стабільно працювати навіть під час великої кількості ворогів, ефектів та об’єктів на сцені. Саме тому під час розробки важливо правильно налаштовувати графіку, освітлення та інші елементи для забезпечення комфортного ігрового процесу.

Отже, створення комп’ютерної 3D-гри на платформі Unity є актуальним та перспективним напрямком, який дозволяє реалізовувати сучасні різноманітні ігрові механіки, створювати цікавий та динамічний геймплей для гравців та забезпечувати користувачам багатий цікавий ігровий досвід.

1.2 Огляд аналогів

Left 4 Dead 2 [2]

Left 4 Dead 2 – це кооперативний шутер від першої особи, у якому гравці проходять різноманітні рівні та борються проти великої кількості заражених ворогів.

Таблиця 1.1 – Опис «Left 4 Dead 2»

Назва	Left 4 Dead 2
Виробник	Valve
Мова реалізації	C++
Функції	1. кооперативний режим проходження; 2. велика кількість ворогів; 3. різноманітна зброя та предмети.
Переваги	1. динамічний геймплей; 2. висока реіграбельність; 3. якісний штучний інтелект.
Недоліки	1. застаріла графіка; 2. обмежена система розвитку персонажа.

Рисунок 1.1 – Зображення геймплею «Left 4 Dead 2»

Killing Floor 2 [3]

Killing Floor 2 – це кооперативний шутер від першої особи, де гравці повинні виживати проти хвиль мутантів. Гра містить систему класів, прокачування та велику кількість модифікацій зброї.

Таблиця 1.2 – Опис «Killing Floor 2»

Назва	Killing Floor 2
Виробник	Tripwire Interactive
Мова реалізації	C++
Функції	1. система хвиль ворогів; 2. прокачування персонажа; 3. великий вибір зброї; 4. битви з босами.
Переваги	1. динамічний бойовий процес; 2. велика кількість зброї; 3. хороша оптимізація.
Недоліки	1. одноманітність деяких рівнів; 2. висока складність для новачків.

Рисунок 1.2 – Зображення геймплею «Killing Floor 2»

Warhammer: Vermintide 2 [4]

Warhammer: Vermintide 2 – це PVE Action FPS гра, в якій гравці проходять різноманітні рівні та завдання де борються проти великої кількості ворогів, використовуючи ближній та дальній бій.

Таблиця 1.3 – Опис «Warhammer: Vermintide 2»

Назва	Warhammer: Vermintide 2
Виробник	Fatshark
Мова реалізації	C++
Функції	1. кооперативне проходження; 2. система класів; 3. велика кількість ворогів; 4. система покращення спорядження.
Переваги	1. атмосферний світ; 2. якісна бойова система; 3. різноманітні локації.
Недоліки	1. високі системні вимоги; 2. складність балансування класів.

Рисунок 1.3 – Зображення геймплею «Warhammer: Vermintide 2»

Проведений аналіз аналогів показав що жанр PVE Shooter і зараз залишається досить популярним серед гравців. Це пов’язано з тим що він поєднує динамічні бойові сцени, розвиток персонажа та різні ігрові механіки.

Такі ігри приваблюють користувачів швидким темпом геймплею, великою кількістю зброї та можливістю поступово покращувати свого персонажа під час проходження.

Кожна з розглянутих ігор має свої особливості та по різному впливає на розвиток жанру. Деякі більше орієнтовані на кооператив, інші – на різноманітність ворогів або унікальні механіки зброї. Але в цілому у всіх них є спільні риси активні бойові дії, велика кількість супротивників, система прокачування та використання різних типів зброї.

Отже можна зробити висновок що жанр PVE Shooter не втрачає своєї актуальності. Більше того він продовжує розвиватися з’являються нові механіки покращується графіка та загалом ігровий процес стає більш цікавим для гравців.

1.3 Визначення технічного завдання для ігрового застосунку

Технічне завдання полягає у створенні та реалізації комп’ютерної 3D-гри у жанрі PVE Shooter, яка забезпечить динамічний, захопливий та тривалий ігровий процес для користувачів. Застосунок повинен відповідати наступним вимогам [5]:

1. меню при запуску – після запуску гри повинно з'являтися меню, що дозволяє користувачам підготуватись до проходження гри;
2. перегляд та запуск збережених етапів гри – користувач повинен мати можливість переглядати інформацію про збережені сесії, завантажувати їх та видаляти;
3. розташування кнопок – розташування функціональних кнопок інтерфейсу повинно забезпечувати зручність використання для користувача та коректне відображення на комп’ютерах із різними розмірами екранів;
4. вид від першої особи – основне відображення ігрового процесу повинно бути реалізовано у вигляді камери від першої особи (FPS), що забезпечує більш поглиблене відчуття гравця у ігровому середовищі;
5. системні вимоги – ігровий застосунок повинен працювати на ПК з операційною системою Windows версії не нижче 10.

Перш ніж почати розробку, необхідно обрати платформу для розробки гри, яка буде відповідати наступним критеріям:

– підтримка комп'ютерних платформ – платформа повинна підтримувати розробку ігор для комп'ютерних систем;

– створення 3D ігор – можливість розробки 3D ігор повинна бути обов'язковою;

– зручність використання – інтерфейс для розробки має бути зрозумілим та зручним;

– якісна документація – наявність детальної та різносторонньої документації є важливим фактором для швидкого освоєння та використання;

– **безкоштовне розповсюдження** – платформа повинна пропонувати безкоштовну версію або індивідуальний план, що дозволяє створювати ігри без додаткових витрат.

Unity було обрано як одну з найпопулярніших платформ для розробки ігор, оскільки вона повністю відповідає основним вимогам, необхідним для створення сучасного ігрового проєкту [6]:

− **підтримка різних операційних систем** – Unity дозволяє розробляти ігри для різних платформ, зокрема Windows, macOS та Linux, що дає можливість запускати готовий продукт на різних комп’ютерах;

− **потужні можливості для 3D–розробки** – середовище надає широкий набір інструментів для роботи з 3D–графікою, включаючи моделювання, анімацію, освітлення та інші важливі елементи створення гри;

− **зручний та зрозумілий інтерфейс** – Unity має досить простий у використанні інтерфейс, що дозволяє швидше освоїти середовище та більше уваги приділяти розробці самої гри;

− **наявність великої кількості матеріалів** – платформа підтримується великою спільнотою, а також має офіційну документацію, навчальні курси та інші ресурси, які допомагають вирішувати проблеми під час розробки;

– **безкоштовність** – Unity має безкоштовну версію, яка містить в собі усі основні можливості для створення ігор, а також дозволяє без додаткових витрат експортувати проєкти на різні платформи.

Unity має зручний та зрозумілий інтерфейс, який складається з кількох основних вікон. Наприклад, вікно Scene(Сцена) використовується для перегляду і редагування ігрової сцени, Inspector(Інспектор) – для налаштування різних властивостей об’єктів, а у вкладці Project(Проект) зберігаються всі файли та ресурси самого проєкту. Крім цього, Unity надає вбудовані сервіси, які можуть використовуватися для роботи з гравцями, зокрема для їх залучення, утримання та монетизації.

Окрім інтерфейсу, середовище містить власний редактор, що дозволяє працювати з 3D-об’єктами та анімаціями безпосередньо в самому рушії. Для створення логіки гри використовується мова програмування C#, яка є досить зрозумілою та має велику кількість навчальних матеріалів. Також Unity надає можливість експортувати готову гру на різні платформи, такі як Windows, macOS та Linux. Більшість параметрів і налаштувань можна змінювати прямо в редакторі, що значно спрощує процес розробки та тестування гри.

1.4 Опис загального алгоритму виконання проєкту

Для успішної розробки гри необхідно заздалегідь продумати всі етапи роботи та визначити приблизні терміни виконання. Час створення проєкту може суттєво відрізнятися залежно від його складності та обсягу. У цілому, розробка гри є досить складним і багатоетапним процесом, де на кожному етапі реалізуються окремі компоненти, які згодом об’єднуються в єдиний завершений продукт.

Перший етап – створення меню гри. Меню є основним візуальним елементом враження від якого отримує гравець ще до початку гри,тому необхідно визначитись із дизайном майбутньої гри, обирається палітра кольорів, також планується майбутній функціонал и навігація кнопок меню.

Другий етап – проєктування рівнів. Воно є важливою частиною розробки гри, оскільки саме вони визначають, що та де саме буде робити гравець під час проходження гри. На цьому етапі продумується загальна концепція рівнів, їхній дизайн, рівень складності, а також формуються основні цілі та завдання, які необхідно виконати в процесі гри.

Третій етап – моделювання. На цьому етапі вже створюються 3D-моделі персонажів та моделі NPC, об'єктів, ландшафтів та інших елементів гри. Також за необхідністю знаходять та завантажують деякі моделі які є в доступі спільноти. Для цих цілей зазвичай використовуються спеціалізовані програми, такі як Blender, Maya або 3DS Max. А для готових моделей Unity Assets Store.

Четвертий етап – розробка. Етап розробки передбачає безпосереднє створення програмної частини гри, де реалізуються всі основні функції, геймплей та інші важливі елементи. Саме на цьому етапі відбувається написання коду, створення правил, логіки, взаємодії гравця з ігровим світом та інтеграція різних компонентів гри. Для цього розробники зазвичай використовують сучасні рушії, такі як Unity, Unreal Engine або GameMaker Studio.

Завершальний стан розробки є тестування гри. На цьому етапі вона проходить ряд перевірок, під час яких оцінюється стабільність роботи, коректність функцій та загальний ігровий процес. Це дуже важливий та вирішальний етап, оскільки саме тут можна виявити різні помилки, баги та інші недоліки різної складності ще до випуску гри для користувачів.

Паралельно з підготовкою до розробки формується GDD – документ ігрового дизайну. У ньому описуються основні елементи гри, такі як жанр, загальна концепція, сюжет, персонажі, рівні, складність та особливості геймплею. У процесі розробки цей документ може доповнюватися та змінюватися.

Створення GDD допомагає краще структурувати проєкт, зменшити кількість помилок і непорозумінь під час роботи, за необхідністю розробники можуть інтегрувати у розробку інших спеціалістів яким цей документ значно допоможе прискорити розуміння в якій частині є розробка, та що вже було

зроблено, ну а також зробити кінцевий продукт більш цілісним. Крім цього, перед фінальним релізом гра може бути випущена у вигляді бета-версії, щоб виявити та виправити ті недоліки, які могли залишитися непоміченими на попередніх етапах розробки.

Висновки до розділу 1

Майбутнє інформаційних технологій тісно пов’язане з розвитком комп’ютерних 3D-ігор, які з кожним роком стають все більш популярними серед користувачів. Завдяки високій продуктивності сучасних ПК з’являється можливість створювати складні та візуально привабливі ігрові проєкти, що забезпечують гравцям захоплюючий ігровий досвід. Ігри жанру PVE Shooter мають велику кількість аналогів, тому під час розробки важливо реалізувати унікальні елементи, які дозволять проєкту виділятися серед інших та подарувати користувачу особливий досвід. Однією з основних переваг майбутньої гри є поєднання динамічного бойового процесу з новою сюжетною лінією, системи прокачування персонажа та різноманітних модифікацій зброї. Це дозволяє зробити геймплей більш цікавим і різноманітним для користувача. Разом з тим, під час розробки можуть виникати певні труднощі, зокрема оптимізація продуктивності гри, балансування складності, а також адаптація проєкту під різні конфігурації комп’ютерів. Вирішення цих завдань є важливим для створення якісного продукту, оскільки це може зацікавити гравців та допоможе зайняти грі, своє місце серед інших ігор даного жанру.

2 АНАЛІЗ ФРЕЙМВОРКІВ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ

2.1 Вибір рушія для розробки ігрового застосунку

Unity

Unity – це універсальне середовище для розробки ігор, яке дає можливість створювати проєкти для різних платформ, зокрема Windows, macOS, Linux, Android, iOS, а також консолей, таких як Xbox і PlayStation. Завдяки цьому розроблений продукт можна адаптувати під широку аудиторію користувачів. Середовище має досить зручний і зрозумілий інтерфейс, що дозволяє працювати як із візуальною частиною гри, так і з програмним кодом. Це робить Unity зручним інструментом як для програмістів, так і для дизайнерів. За допомогою цього рушія можна створювати 3D-об’єкти, налаштовувати їхню взаємодію, а також застосовувати різноманітні ефекти для покращення якості гри.

Крім того, Unity має велику кількість вбудованих інструментів і компонентів, які спрощують процес розробки. У деяких випадках це дозволяє реалізовувати функціонал навіть без значного обсягу коду, що в результаті допомагає зекономити час під час створення проєкту.

Однією з переваг Unity є підтримка різних мов програмування, зокрема C#, що дає розробникам можливість працювати у зручному для них середовищі. Це значно спрощує процес створення ігрової логіки та взаємодії між об’єктами. Крім того, Unity активно використовується для розробки проєктів у сфері віртуальної та доповненої реальності (VR та AR), що робить його ще більш актуальним серед сучасних розробників.

Також варто відзначити велику спільноту користувачів Unity. Завдяки цьому можна легко знайти навчальні матеріали, приклади коду та готові рішення для реалізації різних ігрових механік. Наявність великої кількості документації та ресурсів суттєво полегшує вирішення технічних проблем і дозволяє пришвидшити процес розробки гри.

Unreal Engine

Unreal Engine – це потужний рушій для розробки ігор та різноманітних візуальних проєктів, який був створений компанією Epic Games. Спочатку він з’явився у 1998 році та використовувався переважно для створення шутерів від першої особи, але з часом значно розширив свої можливості. Сьогодні Unreal Engine застосовується не тільки в ігровій індустрії, а й у таких сферах, як архітектурна візуалізація, кіноіндустрія та віртуальна реальність.

Цей рушій надає розробникам широкий набір інструментів для роботи з фізикою, анімацією, освітленням та іншими важливими складовими гри. Завдяки цьому можна створювати проєкти з високим рівнем деталізації та більш реалістичним ігровим середовищем [7].

В Unreal Engine основною мовою програмування є C++, що дозволяє розробникам отримати високий рівень контролю над проєктом та ефективно оптимізувати гру. Разом з цим рушій підтримує і більш доступні інструменти, зокрема систему візуального програмування Blueprint, а також мови Python і Lua. Це дає можливість працювати як досвідченим програмістам, так і тим, хто тільки починає знайомство з розробкою ігор. Однією з важливих особливостей Unreal Engine є підтримка рендерингу в реальному часі, що дозволяє досягати високої якості графіки та ефектів.

Крім цього, Unreal Engine має власний маркетплейс, де доступна велика кількість готових ресурсів: 3D-моделі, текстури, звуки та інші елементи, які можна використовувати під час розробки. Це значно спрощує процес створення гри та економить час.

Водночас варто зазначити, що Unreal Engine орієнтований на високоякісну графіку, через що він є більш вимогливим до ресурсів комп’ютера та складнішим у використанні. Розробка на цьому рушії часто потребує глибших технічних знань і більше часу на освоєння. Для невеликих проєктів, таких як індивідуальна розробка PVE Shooter гри, це може ускладнювати процес і збільшувати терміни створення продукту. Саме тому для подібних задач частіше обирають Unity, який є більш простим і зручним у використанні.

CryEngine

CryEngine – це потужний ігровий рушій, розроблений компанією Crytek, який відомий своїми можливостями у створенні максимально реалістичної графіки та якісних фізичних симуляцій. Його використовують не лише для розробки ігор, а й у сферах архітектурної та промислової візуалізації.

Серед основних можливостей CryEngine можна виділити сучасну систему рендерингу, реалістичну фізику об’єктів та підтримку технологій віртуальної реальності. Завдяки цьому рушій дозволяє створювати детально опрацьовані ігрові світи з точним відображенням освітлення, тіней та інших графічних ефектів [8].

CryEngine підтримує розробку ігор для різних платформ, зокрема ПК, консолей та мобільних пристроїв. Проте, у порівнянні з іншими рушіями, він вважається більш складним у використанні, що може ускладнити роботу для початківців. Крім цього, CryEngine має досить високі вимоги до апаратного забезпечення як на етапі розробки, так і для кінцевих користувачів, що може бути проблемою при створенні проєктів, орієнтованих на слабші комп’ютери.

Ще одним недоліком є відносно менша активність спільноти розробників у порівнянні з Unity або Unreal Engine. Через це іноді складніше знайти готові рішення, приклади або швидку допомогу під час виникнення технічних проблем.

Також варто враховувати, що ігровий рушій CryEngine, як і Unreal Engine, має досить високу складність освоєння і вимагає хорошого рівня технічних знань для ефективного використання. Для розробки комп’ютерного застосунку у жанрі PVE Shooter гри це може бути не зовсім оптимальним варіантом, оскільки основний акцент у таких проєктах робиться не лише на графіці, а й на геймплеї, оптимізації та швидкості розробки. Крім того, через меншу популярність рушія доступність навчальних матеріалів і підтримки також є більш обмеженою.

Таблиця переваг та недоліків для порівняння платформ розробки (табл.2.1-2.3): Unity, Unreal Engine, CryEngine.

Таблиця 2.1 – Переваги та недоліки Unity

№	**Unity**	
	Переваги	*Недоліки*
1	Простий та інтуїтивний інтерфейс	Обмежені можливості для розробки графіки, залежно від версії
2	Велика спільнота користувачів та безкоштовні ресурси, що допомагають в навчанні та розробці проєктів	Обмежена можливість контролювання рендеринга та обробки даних
3	Широке застосування і підтримка різних платформ	Іноді можуть виникати проблеми з оптимізацією та продуктивністю
4	Безкоштовний	

Таблиця 2.2 – Переваги та недоліки Unreal Engine

№	Unreal Engine	
	Переваги	*Недоліки*
1	Висока якість графіки та візуальних ефектів	Висока вартість підписки на платформу
2	Широкі можливості для розробки ігор, включаючи VR та AR проєкти	Складніше вивчення та розробка порівняно з Unity
3	Є можливість використовувати скриптові мови, такі як C++, Python, а також власні мови програмування	Через свої потужні особливості до графіки, має великі вимоги до обладнання
		Потребує великого досвіду у розумінні C++

Таблиця 2.3 – Переваги та недоліки CryEngine

№	CryEngine	
	Переваги	*Недоліки*
1	CryEngine має досить зручний та потужний інтерфейс для розробників, що дозволяє швидко створювати та редагувати ігрові об'єкти	CryEngine вимагає від розробника високого рівня знань та навичок в програмуванні та редагуванні ігрових об'єктів
2	CryEngine надає готові рішення для створення ігрових механік, таких як фотореалізм та розумне освітлення	Для роботи з CryEngine потрібна потужна комп'ютерна система, що може бути проблемою для деяких користувачів
3	CryEngine має вражаючу графіку, яка дозволяє створювати дуже деталізовані ігрові світи	У порівнянні з Unity та Unreal Engine, спільнота CryEngine є досить маленькою, що може ускладнити пошук допомоги
4	Може працювати з дуже великими ігровими світами, що дозволяє створювати масштабні проєкти	CryEngine підтримує лише деякі платформи, такі як Windows та Xbox, що може бути обмеженням для деяких розробників

Unity було обрано як найбільш оптимальну платформу для розробки гри у жанрі PVE Shooter завдяки її зручності у використанні, гнучкості та швидкості розробки. Рушій надає всі необхідні інструменти для реалізації FPS-механік, системи стрільби, роботи з анімаціями та іншими важливими елементами ігрового процесу, а також дозволяє без проблем адаптувати проєкт під різні платформи.

Unreal Engine та CryEngine, незважаючи на свої потужні можливості в області графіки та фізики, є більш складними у використанні та мають вищі вимоги до ресурсів. Через це їх застосування для розробки подібного проєкту може бути менш зручним, особливо у випадку індивідуальної розробки або роботи невеликої команди.

2.2 Вибір мови програмування для розробки

У процесі розробки ігор на платформі Unity найчастіше використовуються такі мови програмування, як C#, C++ та Java. Кожна з них має свої особливості, тому вибір мови залежить від вимог проєкту, його складності та досвіду розробника.

Наприклад, C++ зазвичай застосовується у високопродуктивних проєктах, де важлива максимальна ефективність роботи з графікою та обчисленнями. Java також є досить поширеною мовою завдяки своїй кросплатформенності, проте у контексті Unity вона використовується значно рідше.

Найбільш популярною мовою для роботи з Unity є C#, оскільки саме вона є основною в цьому рушії. Вона має відносно простий синтаксис, добре підходить для реалізації ігрової логіки та має велику кількість навчальних матеріалів, що значно спрощує процес розробки [9].

Оскільки Unity тісно інтегрований з мовою програмування C#, для розробки 3D-гри було обрано саме цю мову. C# є однією з найпоширеніших серед Unity-розробників, адже вона має зрозумілий синтаксис, високий рівень абстракції та дозволяє досить швидко реалізовувати різні ігрові механіки.

Однією з важливих переваг C# є зручність у створенні інтерфейсу, а також інтеграції з анімаціями та візуальними ефектами. Це дає можливість створювати більш насичене ігрове середовище, реалізовувати взаємодію між об’єктами та працювати з поведінкою персонажів. Крім того, мова входить до платформи .NET, що відкриває доступ до великої кількості бібліотек, які значно спрощують розробку.

Також C# підтримує багатопоточність, що дозволяє ефективніше виконувати складні обчислення та розподіляти навантаження між процесами. Це важливо для ігор, де одночасно присутня велика кількість об’єктів, наприклад ворогів або різних елементів оточення, які взаємодіють між собою під час ігрового процесу.

Ще однією перевагою є відносно високий рівень надійності та контроль за помилками. Завдяки статичній типізації багато проблем можна виявити ще на етапі написання коду, що зменшує кількість критичних помилок під час виконання гри. Також це допомагає зробити програму більш стабільною в цілому.

Для розробки було обрано середовище Visual Studio, яке має зручні інструменти для написання коду, його перевірки та налагодження. Воно добре інтегрується з Unity, що дозволяє швидко вносити зміни в код і одразу перевіряти їх у самому проєкті. Це значно спрощує процес розробки та дозволяє швидше знаходити і виправляти помилки [10].

Visual Studio добре інтегрується з Unity, що дозволяє автоматично застосовувати зміни в коді без необхідності виконувати додаткові дії вручну. Це значно спрощує роботу, особливо коли проєкт стає більшим і містить багато різних компонентів. Завдяки цьому розробник може швидко додавати нові елементи гри, наприклад механіки, ефекти або інші частини, які впливають на геймплей.

Окрім цього, середовище має зручні інструменти для керування проєктом, а також можливості для тестування і налагодження. Це дозволяє швидко знаходити помилки в коді та виправляти їх у процесі розробки, що робить сам процес більш ефективним.

Таким чином, використання мови програмування C# разом із середовищем Visual Studio є більш доцільним вибором для розробки 3D-ігор на Unity. Таке поєднання забезпечує зручність у роботі, достатню продуктивність та стабільність під час створення ігрового проєкту. Крім того, це дозволяє ефективно реалізовувати основні ігрові механіки та спрощує процес розробки.

Також варто ще відзначити, що така зв’язка інструментів підтримує можливість створення кросплатформних застосунків, що робить її більш популярною серед розробників сучасних ігор.

2.3 Вибір застосунку для моделювання

Візуальна складова відіграє дуже важливу роль у створенні комп’ютерних 3D-ігор, особливо при розробці на Unity. Усе, що бачить гравець – персонажі, оточення, ефекти – створюється розробниками та дизайнерами і напряму впливає на загальне враження від гри. Саме тому кожна деталь має значення, адже від неї залежить атмосфера та сприйняття ігрового процесу.

Вибір інструментів для 3D-моделювання також є одним із ключових етапів розробки, оскільки від цього залежить якість графіки, деталізація об’єктів та можливості їх використання в ігровому середовищі.

Для порівняння програм, які використовуються для створення 3D-моделей у Unity, доцільно розглянути два популярні рішення: Cinema 4D та Blender.

Cinema 4D

Cinema 4D – це програмний продукт компанії Maxon, який використовується для створення 3D-моделей, анімації та візуалізації. Його активно застосовують не тільки в кіноіндустрії та рекламі, а й під час розробки комп’ютерних ігор, зокрема 3D-проєктів на Unity. За допомогою Cinema 4D розробники можуть створювати деталізовані моделі та різноманітні візуальні ефекти, які потім легко імпортуються в Unity для подальшого використання в ігрових сценах і при створенні персонажів [11].

Однією з головних переваг Cinema 4D є зручний і зрозумілий інтерфейс, завдяки якому програму можуть використовувати як новачки, так і більш досвідчені розробники. Також вона підтримує велику кількість форматів файлів, що дозволяє без проблем переносити створені моделі в інші програми або безпосередньо в Unity. Це робить процес розробки більш гнучким, оскільки можна ефективно використовувати вже готові ресурси. Окрім цього, у програмі є вбудована бібліотека моделей і матеріалів, що допомагає значно пришвидшити створення ігрового середовища.

Cinema 4D також має широкі можливості для роботи з анімацією. У ній доступні інструменти, такі як ієрархія об’єктів та скелетна анімація, які

дозволяють детально налаштовувати рухи персонажів і різних об’єктів. Це можна використовувати, наприклад, для створення анімацій персонажів у грі на Unity, де важливо правильно передати рухи та взаємодію з оточенням. Крім того, програма дозволяє створювати тривимірний текст, що може бути корисним при розробці інтерфейсу гри.

Ще однією сильною стороною Cinema 4D є потужний рендеринг, який дає змогу створювати якісні та реалістичні зображення. Це важливо для 3D-ігор, оскільки саме візуальна частина сильно впливає на сприйняття гри. Програма підтримує як власний рендер, так і сторонні рішення, наприклад Arnold, Octane або Redshift, що дозволяє обрати найбільш підходящий варіант для конкретного проєкту. Під час роботи можна використовувати сучасні методи освітлення та текстурування, щоб покращити вигляд сцени.

Окремо варто відзначити систему MoGraph, яка призначена для створення складних анімацій та динамічних ефектів. За її допомогою можна реалізовувати різні візуальні елементи, що змінюються залежно від дій гравця або подій у грі, що робить ігровий процес більш інтерактивним.

Також Cinema 4D добре інтегрується з іншими програмами, наприклад Adobe After Effects та Adobe Illustrator. Це дає можливість переносити моделі та анімації для подальшої обробки або використання у відео та рекламних матеріалах. Підтримка різних форматів файлів дозволяє легко обмінюватися ресурсами між різними програмами, що є важливим під час розробки ігор.

Отже, Cinema 4D можна вважати досить потужним інструментом для створення 3D-ігор на Unity, оскільки він має широкі можливості для моделювання, анімації та роботи з візуальними ефектами.

Blender

Blender – це потужна і при цьому безкоштовна програма для 3D-моделювання, створення анімацій та візуалізації. Вона активно розвивається завдяки відкритій спільноті користувачів, серед яких є як ентузіасти, так і професійні розробники. Постійні оновлення дозволяють підтримувати сучасні

технології та інструменти, що є великою перевагою при створенні 3D-ігор на Unity.

Завдяки цьому Blender часто використовується як один із основних інструментів у процесі розробки, особливо коли потрібно створювати якісні моделі, анімації та інші візуальні елементи для ігрових проєктів [12].

Однією з головних переваг Blender є його зручний та продуманий інтерфейс, який значно полегшує процес навчання і подальшого використання програми. Це особливо важливо при розробці ігор на Unity, де часто потрібно швидко створювати та редагувати візуальний контент. Крім цього, Blender підтримує розширення на основі мови Python, що дає можливість створювати власні скрипти та автоматизувати окремі процеси, наприклад генерацію об’єктів або спрощення роботи зі сценами.

Програма має великий набір інструментів для 3D-моделювання. Розробники можуть створювати як прості об’єкти, так і більш складні моделі з високим рівнем деталізації. Також у Blender є інструменти для анімації, зокрема система ключових кадрів і кривих руху, що дозволяє створювати динамічні сцени та анімувати персонажів або об’єкти для подальшого використання в Unity. Це корисно при розробці ігрових елементів, де важлива взаємодія між об’єктами та плавність рухів.

За допомогою вбудованих інструментів можна створювати досить реалістичні сцени. Також підтримуються фізичні симуляції, наприклад рідин або твердих тіл, що дозволяє відтворювати більш природну поведінку об’єктів у грі.

Blender також підтримує роботу з різними форматами файлів, що значно спрощує інтеграцію з іншими програмами та рушіями, зокрема з Unity. Це дозволяє без проблем переносити моделі, текстури та анімації між різними середовищами розробки.

На основі порівняння Blender та Cinema 4D можна зробити висновок, що Blender є більш вигідним варіантом для розробки 3D-ігор на Unity, особливо для індивідуальних розробників або невеликих команд. Це пов’язано з тим, що програма є безкоштовною, має широкий функціонал і постійно оновлюється.

Крім того, відкритий вихідний код дозволяє розширювати можливості програми та адаптувати її під конкретні задачі, що робить Blender зручним і ефективним інструментом у процесі розробки гри.

2.4 Специфікація вимог

ВЛАСТИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Доступність

Запуск і стабільна робота гри мають бути забезпечені на ПК, що відповідають встановленим технічним вимогам.

Переносимість

Програмне забезпечення має корректно працювати на операційних системах Windows 10.

Продуктивність

Продуктивність гри залежатиме від технічних характеристик комп'ютера користувача.

Надійність

Дані користувачів є приватними і кожен має свій власний ігровий процес.

ПРИЗНАЧЕННЯ ЗАСТОСУНКУ

Призначення застосунку, для якої розробляється програмне забезпечення

ПЗ розроблене для забезпечення користувачам розважального досвіду в жанрі PVE Shooter.

Погодження, що ухвалені в програмній документації

Для створення ПЗ та забезпечення його стабільної роботи будуть використовуватись сторонні ассети, власної розробки або модифіковані моделі та бібліотеки Unity.

ЗАГАЛЬНИЙ ОПИС

Сфера застосування

Застосунок розроблено для використання користувачами з метою відпочинку та розваги.

Характеристики користувачів

Основні характеристики користувачів – наявність ПК який відповідає технічним характеристикам ПЗ.

ВИМОГИ ДО ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

Джерела і зміст вхідної інформації (даних)

У цьому ПЗ основним джерелом вхідної інформації є користувач, який самостійно вводить дані, такі як ім'я персонажу.

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Вимоги до програмного забезпечення можна описати через кілька ключових характеристик, які відображають якість і надійність роботи програми.

Коректність

Ця характеристика відображає здатність програми реалізувати алгоритми без помилок. Коректність охоплює всі етапи розробки – від специфікацій та проєктування до реалізації алгоритмів та структури даних.

Стійкість

Ця характеристика визначає здатність ПЗ здійснювати обробку інформації, зберігаючи при цьому відповідність визначеним специфікаціям.

Відновлюваність

Ця характеристика описує можливість програми адаптуватися до виявлених помилок та їх усунення.

Надійність

Надійність ПЗ включає в себе кілька важливих характеристик, серед яких можна виділити цілісність, живучість, завершеність та загальну працездатність системи. Під цілісністю розуміється здатність програми стабільно працювати та протидіяти можливим збоїм. Надійність визначає, наскільки правильно програма обробляє вхідні дані під час виконання. Завершеність означає, що готовий продукт не містить критичних помилок і був якісно протестований. Працездатність, у свою чергу, характеризує можливість програми відновлювати свою роботу після виникнення збоїв або непередбачених ситуацій.

ВИМОГИ ДО ТЕХНІЧНОГО ЗАБЕЗПЕЧЕННЯ

1. Операційна система: Windows 10 64bit
2. Процесор: 2.4 Ghz i5 or greater or AMD equivalent
3. Оперативна пам'ять: 4 ГБ RAM
4. Відеокарт: Graphic card with 2 GB VRAM or higher
5. Вільне місце: 5 ГБ
6. DirectX: версії 11

Висновки до розділу 2

У процесі виконання другого розділу було розглянуто різні ігрові рушії, проаналізовано їх основні особливості та відмінності, а також досліджено програмні засоби, які використовуються для створення 3D-графіки та моделей для гри. Як основний рушій для розробки 3D-гри було обрано Unity. Це пояснюється тим, що він є досить зручним у використанні, має велику кількість готових інструментів і ресурсів, а також підтримує роботу на різних платформах, що робить його універсальним рішенням для створення ігор. У якості мови програмування було обрано C#, оскільки вона є основною для роботи з Unity і широко використовується у розробці ігор. Вона дозволяє ефективно реалізовувати ігрову логіку, працювати з інтерфейсом. Для створення 3D-моделей було обрано Blender. Ця програма є хорошим варіантом як для невеликих команд, так і для індивідуальних розробників, оскільки вона безкоштовна, має широкий набір інструментів і підтримується великою спільнотою, що сприяє її постійному розвитку та вдосконаленню.

3 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ

У проєктуванні використано мову графічного моделювання UML (Unified Modeling Language), яка є універсальним інструментом для візуалізації та створення моделей об’єктів програмного забезпечення. UML – діаграми мають важливе значення під час створення проєктів програмного забезпечення, оскільки дозволяють детально описати логіку роботи системи, виявляти можливі помилки на ранніх етапах розробки та підвищувати ефективність процесу [13].

Завдяки своїй універсальності UML широко використовується фахівцями в IT-галузі – програмістами, аналітиками, менеджерами проєктів і навіть власниками компаній, які прагнуть зрозуміти структуру майбутніх рішень. Хоча UML не є мовою програмування, її моделі можуть слугувати основою для автоматичної генерації коду. Для побудови UML-діаграм у цьому проєкті застосовано програмне забезпечення StarUML, яке забезпечує інтуїтивно зрозумілий інтерфейс і потужні засоби моделювання.

3.1 Написання usecase

Use Case – це текстовий опис різних сценаріїв взаємодії користувача із системою, які можуть закінчитися як успішно, так і невдало, залежно від виконання певних цілей. Сценарій являє собою послідовність дій, які виконує користувач для реалізації конкретних операцій у системі [14].

Існують три основні форми написання*:*

− коротка форма – це стислий опис одного зі сценаріїв, зазвичай успішного, у вигляді одного абзацу. Використовується на етапі початкового аналізу вимог до системи;

− поверхнева форма – це детальніший опис всіх сценаріїв у вільній формі (основного та альтернативних) одного з варіантів використання. Використовується на етапі первинного аналізу вимог до системи;

− повна форма – це детальний опис всіх кроків та дій, включаючи попередні та постумови виконання use case. Використовується на етапі вибору з повного списку варіантів використання важливих для роботи системи.

Короткий use case

Користувач має будувати власну економіку та збройні сили, щоб атакувати базу противника й досягти перемоги. Важливим елементом гри є ефективне управління ресурсами, що видобуваються робітниками. Користувач отримує контроль над єдиним доступним рівнем і має виконувати всі дії, спираючись на механіку реального часу. Успіх залежить від вміння грамотно розподіляти ресурси та ухвалювати стратегічні рішення [14].

Поверхневий use case

Користувач керує робітниками для видобутку ресурсів, будує й покращує споруди та військові підрозділи, а також розробляє стратегії для атаки бази противника. Всі дії відбуваються в межах одного рівня, і користувач змагається з штучним інтелектом. Гра має офлайн-режим, що дозволяє насолоджуватися нею без необхідності підключення до інтернету. Для досягнення перемоги користувач повинен знищити базу суперника, ефективно використовуючи доступні ресурси.

Альтернативний сценарій:

– користувач не зміг захистити свою базу й програв матч;

– користувач збудував недостатньо сильну армію, що не дозволило успішно атакувати базу противника;

– користувач не використав ресурси ефективно, що призвело до сповільнення темпів розвитку та поразки;

– користувач припустився помилки у стратегії, через що програв.

Таблиця 3.1 – Повний use case

Primary Actor	Гравець
Scope	Комп’ютерна 3D-гра жанру PVE Shooter
Level	Проходження ігрового рівня
Preconditions	Гравець запустив гру
Stakeholders and interests	1. програміст – зацікавлений у якісному виконанні поставленого завдання, коректній реалізації функціоналу та стабільній роботі розроблених ігрових механік. 2. тестувальник – зацікавлений у виявленні помилок, перевірці працездатності гри та передачі знайдених недоліків програмісту для їх подальшого усунення. 3. гравець – зацікавлений у проходженні гри, досягненні перемоги, покращенні власних навичок та використанні різних ігрових можливостей для ефективного подолання рівнів.
Main Success Scenario:	1. гравець встановлює гру; 2. гравець запускає гру; 3. гравець обирає початок проходження рівня; 4. гравець аналізує ситуацію на бойовій локації та обирає тактику дій; 5. гравець переміщується рівнем, використовує зброю та знищує ворогів; 6. гравець збирає ресурси, отримує досвід і покращує персонажа або зброю; гравець досягає перемоги шляхом виконання цілі рівня та знищення противників.
Result	Гравець отримує задоволення від успішного проходження рівня, знищення противників та покращення власних навичок у бойовому процесі
Extensions:	
1.	1. Гравець не задоволений ігровим процесом: 1.1. гравець може залишити відгук або звернутися до розробників через систему зворотного зв’язку для подання пропозицій чи скарг; 1.2. негативний досвід може бути пов’язаний із такими причинами: 1.2a. занадто низька складність проходження рівнів; 1.2b. надмірна складність противників або штучного інтелекту; 1.2c. недостатньо зрозумілий інтерфейс користувача; 1.2d. технічні помилки, просідання продуктивності або нестабільна робота гри; 1.2e. некоректний баланс між силою ворогів, зброєю та можливостями персонажа.

Продовження таблиці 3.1

2.	2. Тестувльник знаходить помилку у грі: 2.1a. якщо помилка не є критичною, вона фіксується у звіті тестувальника; 2.1b. програміст аналізує причину помилки та вносить необхідні зміни до коду; 2.1c. після виправлення тестувальник повторно перевіряє відповідну механіку гри 2.2a. якщо помилка є критичною і заважає проходженню гри, її виправлення отримує високий пріоритет; 2.2b. програміст виконує детальний аналіз проблеми, виправляє помилку та проводить додаткове тестування.
3.	3. Гравець не може пройти рівень: 3.1. причиною може бути надмірна кількість ворогів, нестача боєприпасів або неправильний баланс складності; 3.2. гравець може змінити тактику проходження, використати іншу зброю або покращити характеристики персонажа; 3.3. у разі системної проблеми розробник коригує баланс рівня, кількість ресурсів або параметри противників.
Special Requirements	1) гра повинна мати зрозумілий та зручний інтерфейс, який дозволяє гравцю швидко орієнтуватися в основних елементах керування, стані здоров'я, боєприпасах і доступних покращеннях. 2) ігровий процес має бути оптимізований для стабільної роботи на персональних комп'ютерах із різними технічними характеристиками. 3) система штучного інтелекту повинна забезпечувати реалістичну поведінку противників: виявлення гравця, переслідування, атаку, ухилення та реакцію на зміну ситуації на рівні. 4) система прокачування персонажа та модифікації зброї має бути зрозумілою для користувача й безпосередньо впливати на ефективність проходження гри. 5) гра повинна підтримувати автономний режим роботи без необхідності постійного підключення до інтернету.
Frequency of Occurrence	Основні функціональні процеси гри виконуються під час кожного запуску та проходження рівнів. До них належать переміщення гравця, використання зброї, взаємодія з противниками, отримання досвіду, покращення персонажа та завершення бойових локацій. Гра розрахована переважно на автономний режим роботи, тому більшість функцій доступні без постійного підключення до мережі Інтернет.

Після розгляду основних і додаткових сценаріїв використання гри можна краще зрозуміти, як саме гравець буде взаємодіяти з ігровою системою. Такий опис допомагає побачити, які дії виконує користувач під час гри: запускає рівень,

пересувається локацією, бореться з ворогами, використовує зброю, покращує персонажа та намагається досягти перемоги.

Опис use case також дає змогу заздалегідь помітити можливі проблеми, які можуть виникнути під час розробки або проходження гри. Наприклад, рівень може бути занадто складним, вороги можуть поводитися неправильно, інтерфейс може бути незручним, або гра може працювати нестабільно. Якщо врахувати такі моменти ще на етапі проєктування, то готова гра буде більш зручною, зрозумілою та цікавою для гравця.

3.2 Створення діаграми використання

Діаграми варіантів використання (Use Case diagrams) допомагають показати, як користувачі взаємодіють із системою. За їх допомогою можна зрозуміти, які дії виконують різні учасники процесу та які можливості повинна мати система.

Такі діаграми не показують, як система влаштована всередині, а описують її роботу з точки зору користувача. Тобто вони допомагають побачити, що саме може робити гравець, тестувальник або розробник під час взаємодії з грою.

Use Case діаграми зручні на етапі проєктування, тому що дозволяють заздалегідь визначити основні функції гри та уникнути непорозумінь під час розробки. Завдяки цьому простіше узгодити вимоги до майбутнього проєкту і краще зрозуміти, як повинна працювати гра для кінцевого користувача [15].

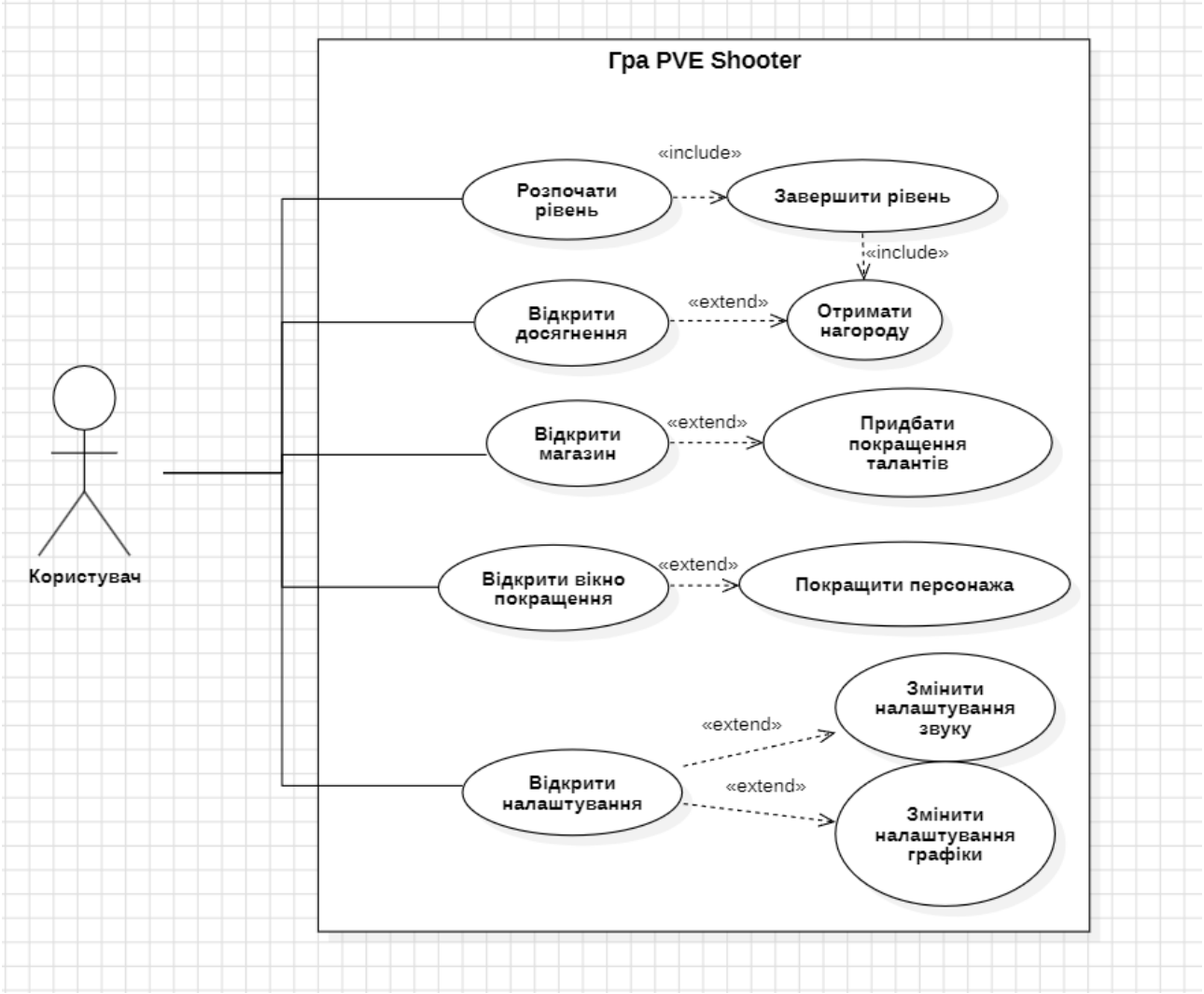

Рисунок 3.1 – Діаграма варіантів використання

Діаграма показує основні варіанти використання гри PVE Shooter та взаємодію гравця з її функціями. Головною дійовою особою є користувач, який може запускати ігровий процес, розпочинати рівень, відкривати досягнення, магазин, вікно покращення персонажа та налаштування.

У межах ігрового процесу гравець може розпочати рівень і після його проходження завершити його. Після завершення рівня система передбачає можливість отримання нагороди. Також гравець може відкривати досягнення, за які в окремих випадках отримує додаткову винагороду.

Окремо на діаграмі показано взаємодію з магазином та системою розвитку персонажа. Через магазин користувач може придбати покращення талантів, а через вікно покращення – підвищити характеристики персонажа. Це дозволяє

зробити ігровий процес більш різноманітним і дає гравцю можливість поступово посилювати свого героя.

Також у грі передбачено налаштування, через які користувач може змінити параметри звуку та графіки. Це робить гру зручнішою для різних гравців і дозволяє адаптувати її під можливості комп’ютера або особисті вподобання.

Зв’язки include та extend показують залежність між окремими діями. Наприклад, отримання нагороди пов’язане із завершенням рівня, а придбання покращень або зміна налаштувань є додатковими можливостями, які розширюють основний сценарій використання гри. Таким чином, діаграма допомагає наочно показати, які основні функції повинні бути реалізовані в грі для забезпечення повноцінної взаємодії з користувачем.

Компоненти діаграми варіантів використання

Діаграма варіантів використання складається з кількох основних елементів: актора, варіанта використання, системи та зв’язків між ними.

Актор – це користувач або інший учасник, який взаємодіє із системою, але не є її частиною. У межах даної гри основним актором є **гравець**, який запускає гру, проходить рівні, відкриває магазин, покращує персонажа та змінює налаштування.

Варіант використання (прецедент) – це певна дія, яку може виконати користувач у системі. На діаграмі він зображується у вигляді еліпса. Наприклад, для гри PVE Shooter такими варіантами є розпочати рівень, завершити рівень, відкрити магазин, покращити персонажа, відкрити налаштування, змінити звук або змінити графіку.

Система – це межі самої програми або гри. На діаграмі вона зображується прямокутником із назвою. У цьому випадку системою є **гра PVE Shooter**, всередині якої розміщені всі основні функції, доступні гравцю.

Зв’язок – показує, як актор взаємодіє з варіантами використання. Наприклад, гравець може напряму відкрити магазин, почати рівень або перейти до налаштувань.

У діаграмах варіантів використання можуть застосовуватися різні типи зв’язків:

1. **Асоціація (Association) –** це звичайний зв’язок між актором і дією, яку він може виконати. Наприклад, гравець пов’язаний із дією розпочати рівень.

2. **Розширення (Extend) –** показує додаткову або необов’язкову дію. Наприклад, після відкриття магазину гравець може додатково придбати покращення талантів. Тобто ця дія не завжди обов’язкова, але вона розширює основний сценарій.

3. **Включення (Include) –** показує дію, яка є частиною іншої дії або обов’язково з нею пов’язана. Наприклад, після завершення рівня гравець може отримати нагороду, оскільки це є логічним результатом проходження рівня.

4. **Генералізація (Generalization) –** використовується тоді, коли один елемент є різновидом іншого. Наприклад, у складнішій системі можна було б розділити користувачів на звичайного гравця, тестувальника або адміністратора. У межах даної діаграми головним актором виступає саме гравець.

3.3 Побудова діаграм взаємодії (послідовності та кооперації)

Діаграми взаємодії використовуються для того, щоб показати, як різні частини гри взаємодіють між собою під час виконання певної дії. Вони допомагають краще зрозуміти, що відбувається всередині гри, коли гравець виконує певну дію – починає рівень, атакує ворога, отримує нагороду або покращує персонажа.

До основних діаграм взаємодії належать діаграма послідовності та діаграма кооперації.

Коли потрібно деталізувати зв'язки між об'єктами, доцільно використовувати діаграми кооперації. Для аналізу порядку виконання дій найкраще підходять діаграми послідовності.

1) **Діаграми послідовності (Sequence Diagrams) [15]**

Діаграма послідовності дозволяє показати, як саме відбувається взаємодія між об’єктами гри у певному сценарії. Вона відображає послідовність дій від початку до завершення процесу.

Наприклад, для гри PVE Shooter можна описати процес проходження рівня. Спочатку гравець запускає рівень, після чого система завантажує локацію. Далі на сцені з’являються вороги, гравець переміщується картою, використовує зброю та атакує противників. Після знищення ворогів система нараховує досвід або нагороду, а потім рівень вважається завершеним.

Ключові елементи:

− об’єкти – учасники процесу, наприклад гравець, ігрова система, ворог, зброя або система нагород;

− повідомлення – дії або команди, які передаються між об’єктами;

− лінія життя – показує, скільки часу об’єкт бере участь у процесі;

− фокус управління – показує момент, коли об’єкт активно виконує певну дію.

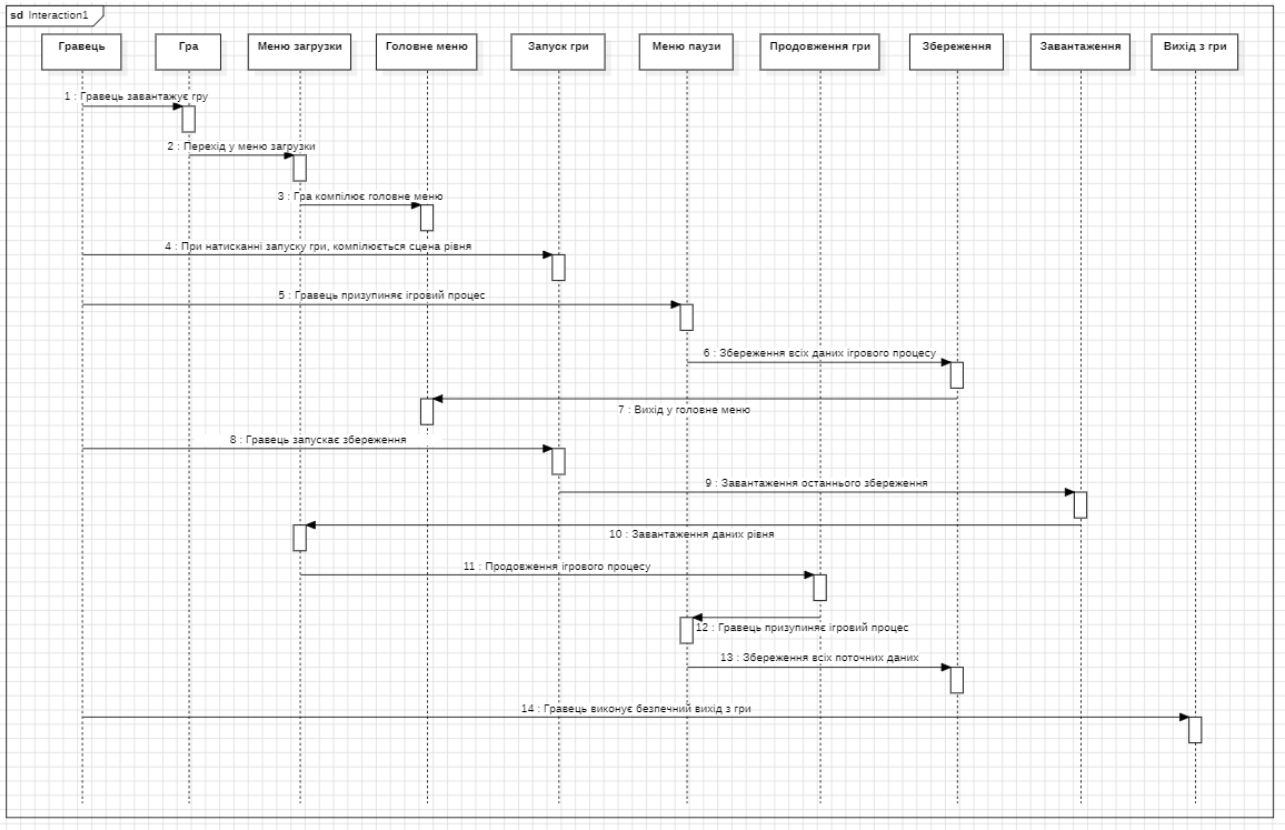

Рисунок 3.2 – Діаграма послідовності

Ця діаграма показує послідовність дій, які відбуваються під час взаємодії гравця з грою. На ній видно, як користувач запускає гру, переходить до головного меню, обирає початок гри та вибирає потрібний рівень для проходження.

Після вибору рівня система завантажує ігрову локацію, після чого гравець починає проходження – пересувається рівнем, бореться з ворогами та виконує основні ігрові дії. Також на діаграмі показано можливість поставити гру на паузу, зберегти поточний прогрес і продовжити проходження з того самого місця.

Окремо відображено завершення рівня. Після успішного проходження гравець отримує нагороду, а потім може перейти до вікна покращень і використати отримані ресурси для посилення персонажа. Наприкінці сценарію показано безпечний вихід з гри.

Діаграма допомагає зрозуміти загальний порядок роботи гри від запуску до проходження рівня, отримання нагороди, покращення персонажа та завершення ігрової сесії. Вона наочно показує, які елементи системи беруть участь у цьому процесі та як вони взаємодіють між собою.

3.4 Діаграми станів та переходів

Діаграми станів є одним із графічних інструментів UML, що застосовується для опису поведінки об’єктів або систем у процесі їхнього функціонування. Вони відображають, у яких станах може перебувати об’єкт протягом свого життєвого циклу, а також показують події або умови, які викликають перехід з одного стану до іншого.

Такі діаграми особливо доцільно використовувати для моделювання реактивних об’єктів, поведінка яких змінюється під впливом зовнішніх або внутрішніх факторів [15].

Завдяки діаграмам станів розробники можуть детальніше проаналізувати динаміку роботи системи, зрозуміти реакцію її компонентів на певні події та сформувати зрозумілу модель поведінки. Вони використовуються під час

дослідження поведінки об’єктів, сценаріїв використання, окремих операцій або функціональних частин системи.

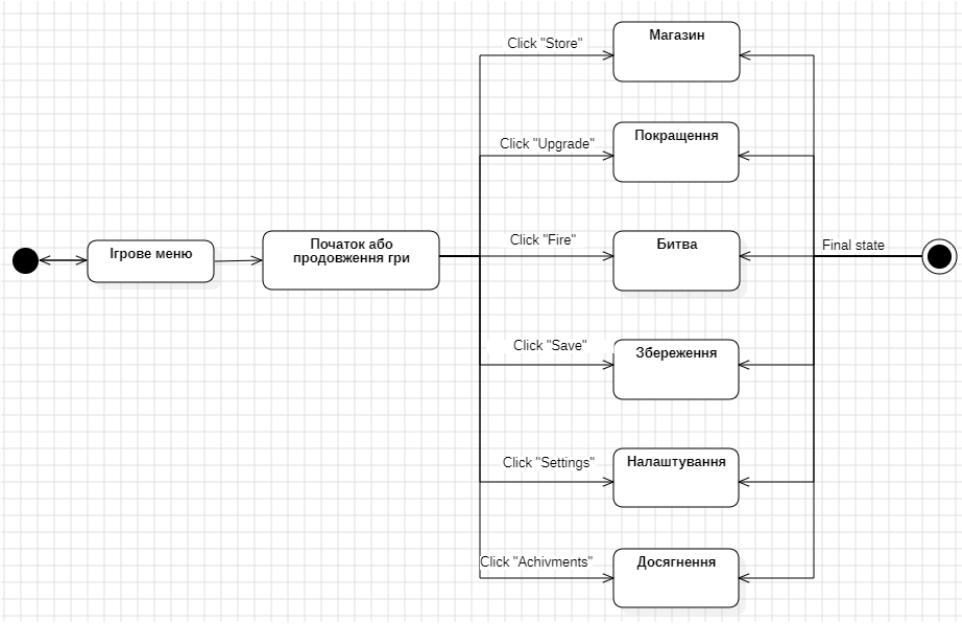

Рисунок 3.3 – Діаграма станів для всієї гри

Ця діаграма показує основні стани, у яких може перебувати гра, та можливі переходи між ними. Вона допомагає краще зрозуміти, як змінюється робота гри на різних етапах від запуску й головного меню до проходження рівня, паузи, завершення гри або виходу.

Кожен стан на діаграмі зображено окремим блоком, а стрілки між ними показують, за яких умов відбувається перехід до іншого стану. Такі переходи можуть бути викликані діями гравця, наприклад натисканням кнопки, вибором пункту меню, завершенням рівня або внутрішніми подіями самої гри.

Завдяки цій діаграмі можна наочно побачити загальну логіку роботи гри та зрозуміти, як користувач переходить між її основними частинами.

Основні компоненти діаграм станів та переходів [15].

1. **Стан (State) –** Певне положення або ситуація, у якій об’єкт перебуває протягом визначеного часу. Стан може бути пов’язаний із виконанням дій або очікуванням події.

2. **Подія (Event) –** Дія або зміна, яка запускає перехід об’єкта з одного стану до іншого. Подією може бути команда користувача, зміна умов середовища або сигнал від іншої частини системи.

3. **Перехід (Transition) –** Зв’язок між двома станами. Він показує, за якої події та за яких умов об’єкт переходить із поточного стану до нового.

4. **Дія (Action) –** Коротка операція, яка виконується під час переходу між станами. Наприклад, зміна значення, оновлення даних або передача повідомлення іншому компоненту.

5. **Діяльність (Activity) –** Процес, який триває, поки об’єкт перебуває у певному стані. На відміну від дії, діяльність може займати більше часу та включати кілька операцій.

6. **Початковий та кінцевий стани (Initial and Final States) –** Початковий стан позначає момент, з якого починається робота або життєвий цикл об’єкта. Кінцевий стан показує завершення цього процесу.

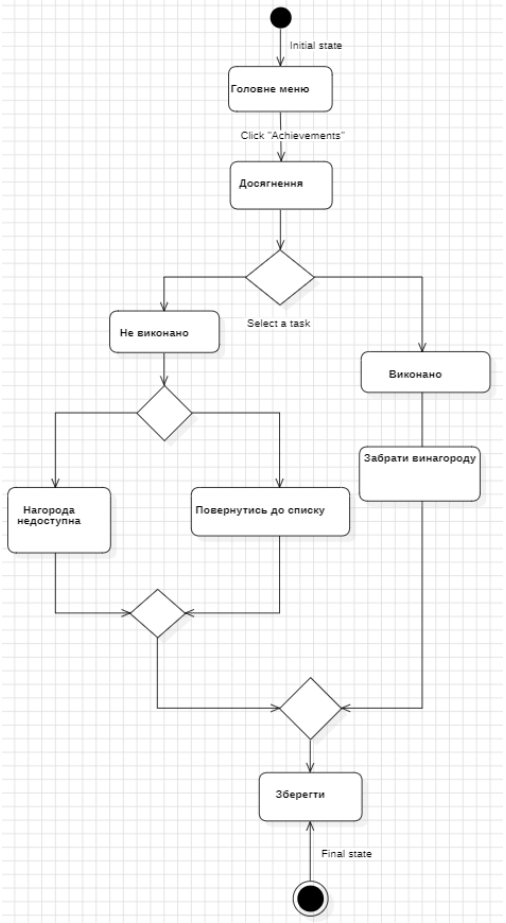

Рисунок 3.4 – Діаграма станів для досягнення

У цій діаграмі показано стани та переходи, пов’язані з роботою системи досягнень у грі. Вона демонструє, які дії виконує гравець для перегляду досягнень і отримання винагороди за їх виконання. Спочатку користувач переходить із головного меню до розділу досягнень, після чого система перевіряє стан обраного завдання.

Якщо досягнення ще не виконано, гравець бачить, що нагорода недоступна, або може повернутися до списку досягнень. Якщо ж умови досягнення виконані, користувач отримує можливість забрати винагороду. Після цього система переходить до збереження змін, щоб зафіксувати факт отримання нагороди.

Така діаграма є важливою, оскільки вона допомагає зрозуміти логіку роботи досягнень – коли гравець може отримати нагороду, за яких умов вона недоступна, і як система зберігає результат після виконання потрібних дій.

3.5 Діаграма діяльності

Діаграма діяльності (Activity Diagram) – це один із видів UML-діаграм, який використовується для опису послідовності дій у системі. Вона допомагає показати, які кроки виконуються, у якому порядку вони відбуваються та як пов’язані між собою.

За своїм виглядом діаграма діяльності нагадує звичайну блок-схему, але має ширші можливості. Вона може показувати не лише просту послідовність дій, а й розгалуження, паралельне виконання процесів та їх узгодження між собою. Завдяки цьому така діаграма дозволяє краще зрозуміти, як система реагує на певні події та як змінюється хід виконання процесу [15].

Діаграми діяльності часто використовують для опису бізнес-процесів, роботи програмних модулів або складних дій у програмних системах. Вони допомагають детально показати алгоритм роботи – які дії виконуються, які рішення приймаються, як дані переходять від одного етапу до іншого та за яких умов відбуваються переходи.

Завдяки своїй зрозумілій структурі діаграми діяльності є зручним інструментом для проєктування різних процесів – від простих сценаріїв роботи користувача до складної логіки програмного забезпечення.

Рисунок 3.5 – Діаграма діяльності покращення

Ця діаграма відображає процес покращення у грі та показує, які дії виконує гравець на кожному етапі. Спочатку користувач переходить до дії **«Покращення»**, після чого система перевіряє, чи має він достатню кількість ресурсів або ігрової валюти.

Якщо потрібні ресурси доступні, виконується покращення, після чого дія успішно завершується. Якщо ж ресурсів недостатньо, гравець не може виконати покращення та має можливість закрити відповідне вікно.

Діаграма наочно показує логіку прийняття рішень у процесі покращення результат залежить від наявності необхідних ресурсів та подальших дій гравця. Завдяки цьому можна краще зрозуміти, як працює механіка покращень і як вона впливає на ігровий процес.

Основні компоненти діаграми діяльності [15]:

1. **Дія (Action) –** основна операція, яку виконує система або її окремий компонент. Вона починається після певної події або отримання потрібних даних і завершується перед переходом до наступного етапу. На діаграмі дія зображується у вигляді прямокутника із заокругленими кутами.

2. **Перехід (Transition) –** зв’язок між діями або іншими елементами діаграми. Він показує, як процес переходить від одного кроку до іншого. На схемі перехід позначається стрілкою.

3. **Початковий і кінцевий стани (Initial and Final States) –** початковий стан показує, звідки починається процес, а кінцевий – де він завершується. Початковий стан зазвичай позначається зафарбованим колом, а кінцевий – колом із додатковим кільцем.

4. **Розгалуження (Decision) –** елемент, який використовується тоді, коли подальший хід процесу залежить від певної умови. Він зображується у вигляді ромба, від якого можуть виходити кілька стрілок із різними варіантами подальших дій.

5. **Паралельне виконання (Fork and Join) –** використовується для показу кількох дій, які можуть виконуватися одночасно. Також цей елемент допомагає показати момент, коли паралельні процеси знову об’єднуються в один потік. На діаграмі зазвичай позначається горизонтальною або вертикальною смугою.

6. **Доріжки (Swimlanes) –** застосовуються для поділу дій між різними учасниками процесу, наприклад користувачем, системою або окремими модулями. Вони допомагають зрозуміти, хто саме відповідає за виконання кожної дії.

Використання діаграми діяльності в бізнес-процесах

Діаграми діяльності часто використовують для опису бізнес-процесів, оскільки вони дозволяють наочно показати послідовність дій, які виконуються різними учасниками або підрозділами. За допомогою таких діаграм можна зрозуміти, як окремі завдання пов’язані між собою, які умови впливають на подальший хід процесу та які ресурси або ролі беруть участь у його виконанні.

Особливо корисними в бізнес-процесах є доріжки, оскільки вони допомагають чітко розподілити відповідальність між учасниками. Завдяки цьому діаграми діяльності зручно використовувати для аналізу процесів, пошуку слабких місць, покращення організації роботи та автоматизації окремих етапів.

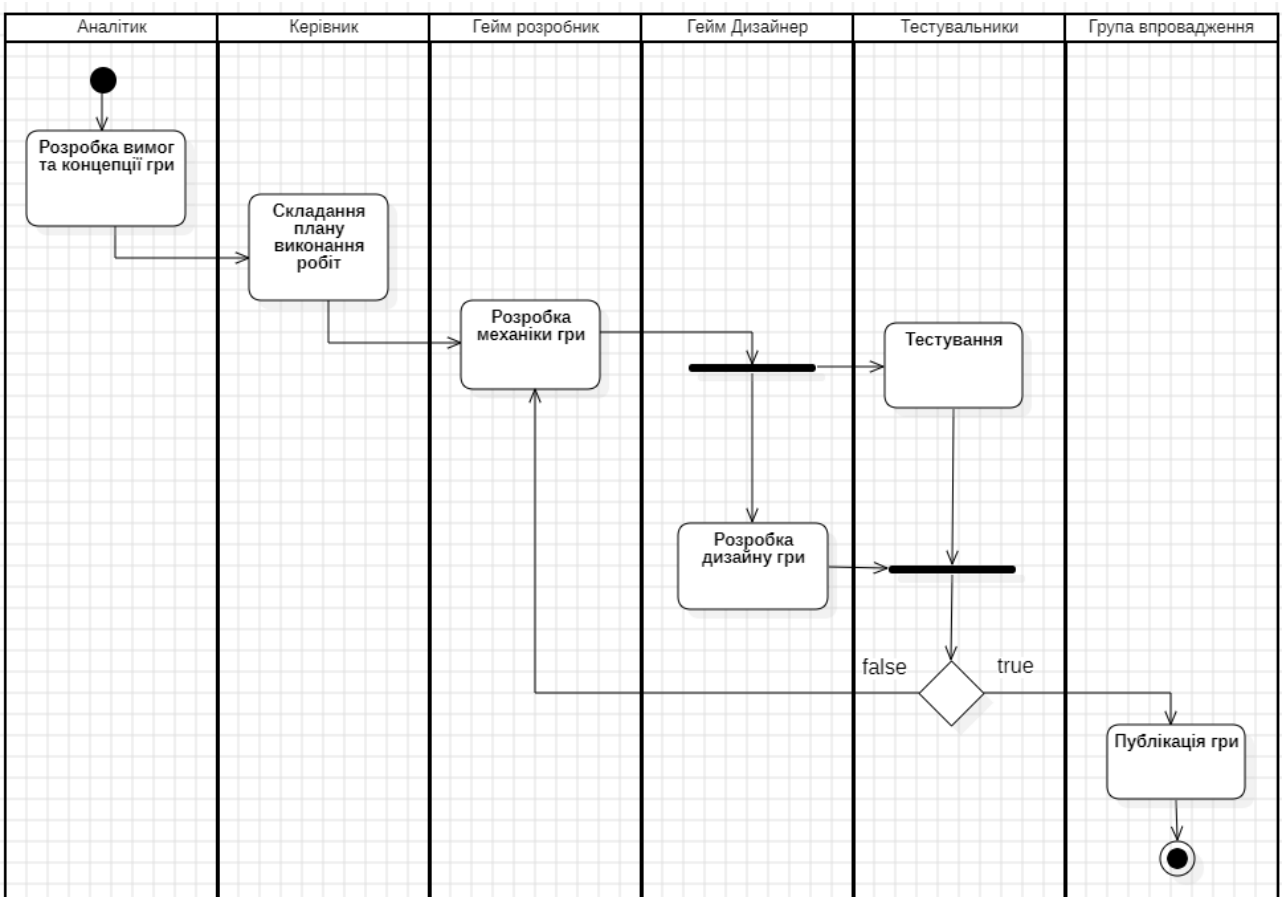

Рисунок 3.6 – Діаграма діяльності для відображення бізнес-процесів

Ця діаграма ілюструє **бізнес-процеси**, пов’язані з розробкою гри, та показує послідовність основних етапів роботи над проєктом. Вона допомагає зрозуміти, як поступово створюється гра, від аналізу вимог і формування ідеї до тестування та підготовки готового продукту.

Кожен етап на діаграмі подано у вигляді окремого блоку, а стрілки між ними показують порядок виконання дій. Наприклад, спочатку визначаються вимоги до майбутньої гри, потім формується її концепція, після цього розробляються основні механіки, створюється дизайн, виконується програмна реалізація, тестування та виправлення помилок.

Така діаграма є корисною, оскільки дозволяє наочно побачити бізнес-процеси розробки гри та краще зрозуміти зв’язок між окремими етапами. Вона допомагає організувати роботу над проєктом і визначити, які дії потрібно виконати для створення готового ігрового прототипу.

3.6 Розробка діаграм компонентів та розгортання

Діаграми компонентів (Component Diagram) та **діаграми розгортання (Deployment Diagram)** належать до важливих UML-інструментів, які використовуються для опису структури програмної системи. Вони допомагають показати, з яких частин складається система, як ці частини пов’язані між собою та в якому середовищі програма буде працювати. Завдяки таким діаграмам легше зрозуміти архітектуру програмного забезпечення, взаємодію його модулів та зв’язок із зовнішніми системами або середовищем виконання.

Діаграма компонентів

Діаграма компонентів використовується для відображення статичної структури програмного забезпечення. Вона показує основні модулі системи, їхні залежності, використані бібліотеки, інтерфейси та інші частини, які беруть участь у роботі програми.

Така діаграма є корисною під час проєктування модульної архітектури, оскільки дозволяє краще зрозуміти, як організований код, які компоненти можна повторно використовувати та які зв’язки існують між окремими частинами системи. Також вона допомагає виявити зайві залежності між модулями й зробити структуру програми більш зрозумілою та зручною для подальшої розробки.

Основні елементи діаграми компонентів:

1. Компонент (Component) – Окрема логічна частина системи, наприклад модуль, бібліотека або функціональний блок. На діаграмі зазвичай зображується прямокутником із назвою.

2. Інтерфейс (Interface) – Набір можливостей або послуг, які компонент надає іншим частинам системи або, навпаки, отримує від них. На діаграмі може позначатися у вигляді кола, з’єднаного з компонентом.

3. Зв’язок (Dependency) – залежність між компонентами або між компонентом та інтерфейсом. Він показує, які частини системи використовують функції інших модулів.

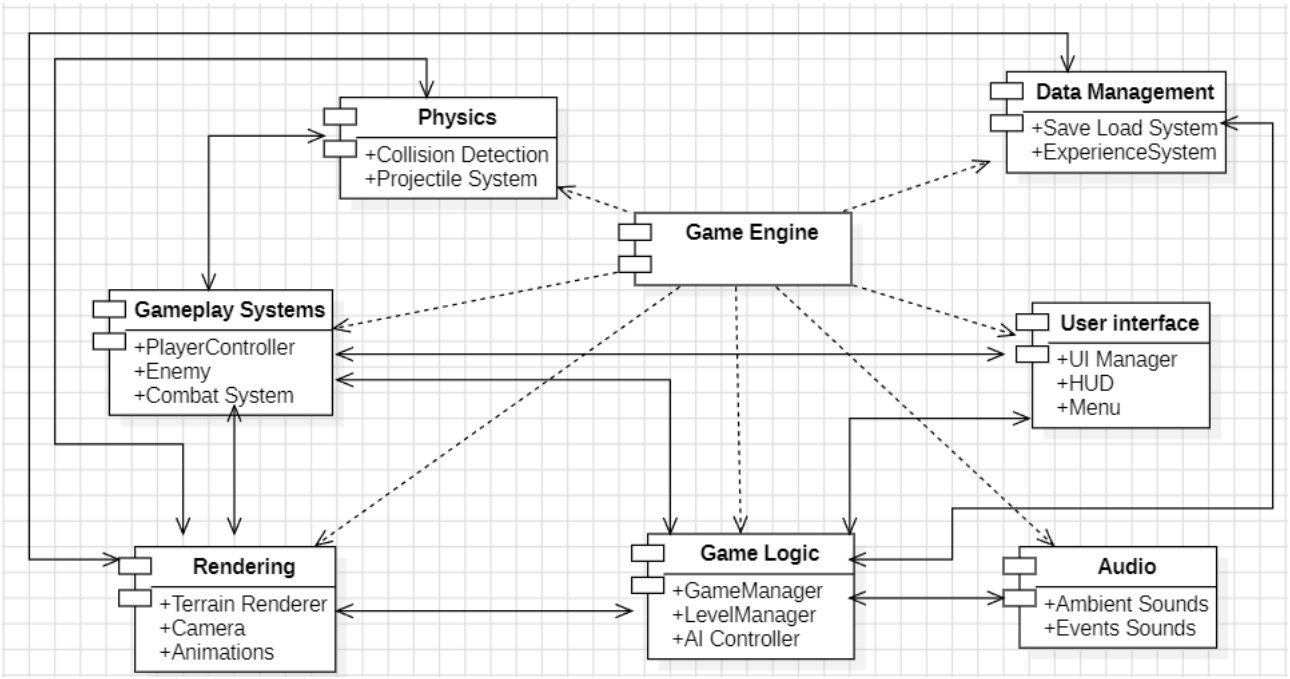

Рисунок 3.7 – Діаграма компонентів

У цій діаграмі показано основні компоненти гри та зв’язки між ними. Вона допомагає краще зрозуміти загальну архітектуру програми, зокрема роботу таких частин, як ігрова логіка, графіка, керування даними, інтерфейс користувача та звукова система.

Кожен блок на діаграмі відповідає за окрему частину гри. Наприклад, компонент рендерингу забезпечує відображення графіки, а аудіосистема відповідає за звукові ефекти та фонові звуки.

Стрілки між компонентами показують напрям взаємодії та передачі даних між окремими частинами системи. Завдяки цьому можна побачити, які модулі залежать один від одного і як вони разом забезпечують роботу гри.

Така діаграма є важливою для проєктування архітектури, оскільки дозволяє наочно показати структуру програми, спростити розуміння її роботи та полегшити подальшу розробку або вдосконалення окремих компонентів.

Діаграма розгортання

Діаграма розгортання використовується для того, щоб показати, як програмні частини системи розміщуються на фізичних пристроях або в певному середовищі виконання. Вона допомагає зрозуміти, на яких вузлах працюють окремі модулі програми та як між ними відбувається обмін даними.

Основні елементи діаграми розгортання [15]:

1. Вузол (Node) – фізичний пристрій або середовище, на якому виконується частина системи. Наприклад, це може бути комп’ютер користувача, сервер, мобільний пристрій або інше обладнання. На діаграмі вузол зазвичай зображується у вигляді об’ємного прямокутника.

2. Артефакт (Artifact) – файл або ресурс, який розміщується на певному вузлі. До артефактів можуть належати виконувані файли, бібліотеки, конфігураційні файли або інші частини програми. На діаграмі артефакт позначається прямокутником із відповідною назвою.

3. Зв’язок (Association) – лінія між вузлами, яка показує взаємодію або передачу даних між ними. За допомогою таких зв’язків можна побачити, як окремі частини системи обмінюються інформацією та працюють разом.

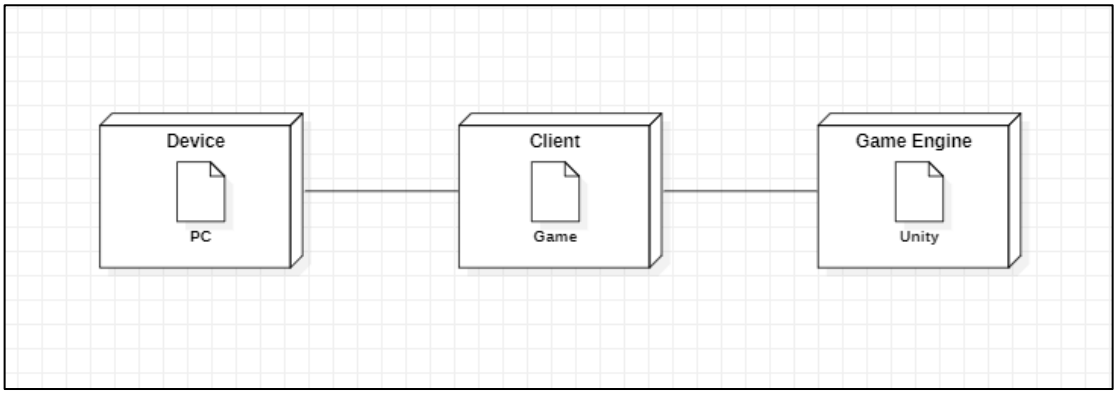

Рисунок 3.8 – Діаграма розгортання

Ця діаграма показує архітектуру гри на рівні розгортання. Вона містить три основні елементи – Device (пристрій), Client (клієнтська частина) та Game Engine

(ігровий рушій). Пристрій користувача запускає клієнтську частину гри, яка взаємодіє з рушієм Unity для обробки графіки, фізики, звуку та основної ігрової логіки. Діаграма допомагає зрозуміти, як компоненти системи пов’язані між собою під час роботи гри..

Висновки до розділу 3

У третьому розділі було розглянуто процес моделювання та проєктування ігрового застосунку. Основну увагу приділено опису сценаріїв використання, створенню UML-діаграм та визначенню основних зв’язків між елементами системи. Це дозволило краще зрозуміти, як гравець взаємодіє з грою та які функції мають бути реалізовані в майбутньому прототипі.

У межах розділу було сформовано use case, у якому описано основні та альтернативні сценарії роботи гри. Завдяки цьому вдалося визначити ключові дії користувача, зокрема запуск гри, проходження рівнів, взаємодію з ворогами, отримання нагород, покращення персонажа та роботу з меню.

Також було створено кілька UML-діаграм. Діаграма варіантів використання дала змогу показати основні можливості гри з точки зору користувача. Діаграми послідовності та кооперації допомогли описати порядок взаємодії між елементами системи під час виконання певних дій. Діаграми станів і переходів показали можливі стани гри та умови переходу між ними. Діаграма діяльності відобразила логіку виконання окремих процесів, а діаграми компонентів і розгортання дозволили показати загальну структуру програмної системи та її розміщення в середовищі виконання.

Таким чином, у третьому розділі було сформовано основу для подальшої програмної реалізації гри. Виконане моделювання допомогло визначити основні функції, структуру взаємодії між компонентами та логіку роботи ігрового процесу. Отримані результати можуть бути використані як база для розробки прототипу гри у жанрі PVE Shooter на рушії Unity.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ

4.1 Створення дизайну ігрового застосунку

UI-дизайн (дизайн користувацького інтерфейсу) в ігровому застосунку відіграє надзвичайно важливу роль, оскільки він визначає спосіб взаємодії користувачів з грою. Грамотно розроблений дизайн може значно покращити враження гравців від гри та забезпечити їм комфортну та просту взаємодію з геймплеєм. Весь інтерфейс виконано у єдиному художньому стилі із використанням текстурованих рам під ковану бронзу, камінь та напівпрозорих підкладок, що гармонує з пустельним сеттингом гри.

В ігровому застосунку присутні такі важливі елементи інтерфейсу, як головне меню, меню налаштувань, меню паузи, меню магазину характеристик та здібностей, а також вікна вибору випадкових покращень. Для створення цього інтерфейсу були використані стандартні інструменти Unity, зокрема UI Canvas-компоненти, що включають різноманітні елементи, такі як кнопки, текстові поля TextMeshPro, панелі, повзунки Slider, прапорці Toggle та випадаючі списки TMP_Dropdown.

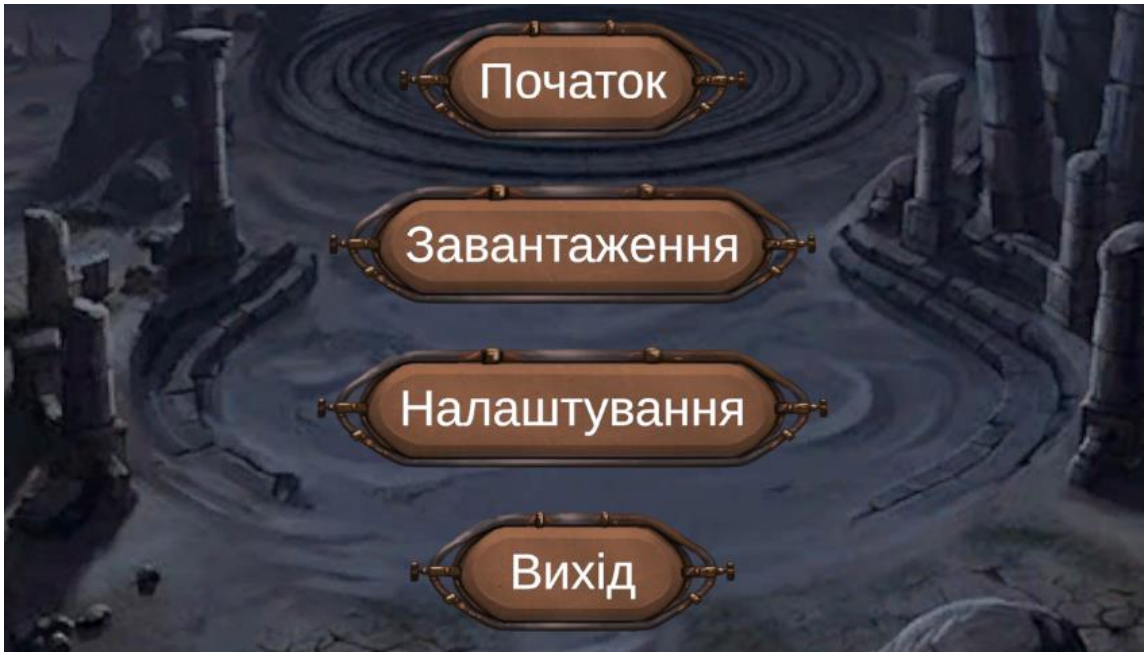

Рисунок 4.1 – Головне меню

Головне меню, включає в собі чотири знаходяться основні кнопки – Початок, Завантаження, Налаштування та Вихід. При натисненні на кнопку (Початок) гравець переходить безпосередньо у геймплей активного рівня. А також на фоні знаходиться текстурована панель із зображенням стародавніх руїн на фоні нічного неба якє забезпечує занурення в атмосферу гри з перших секунд запуску ігрового застосунку.

Рисунок 4.2 – Меню налаштувань

При натисненні на кнопку «Налаштування» користувачу відкривається панель із технічними конфігураціями застосунку (рис. 4.2). Тут вже реалізовано випадаючий список «Роздільна здатність» для адаптації екрана під монітор, повзунки «Гучність» та «Чутливість миші» для лінійного налаштування аудіосистеми й огляду камери персонажа, а також прапорці «Повний екран» та «Звук» для швидкого перемикання активних станів.

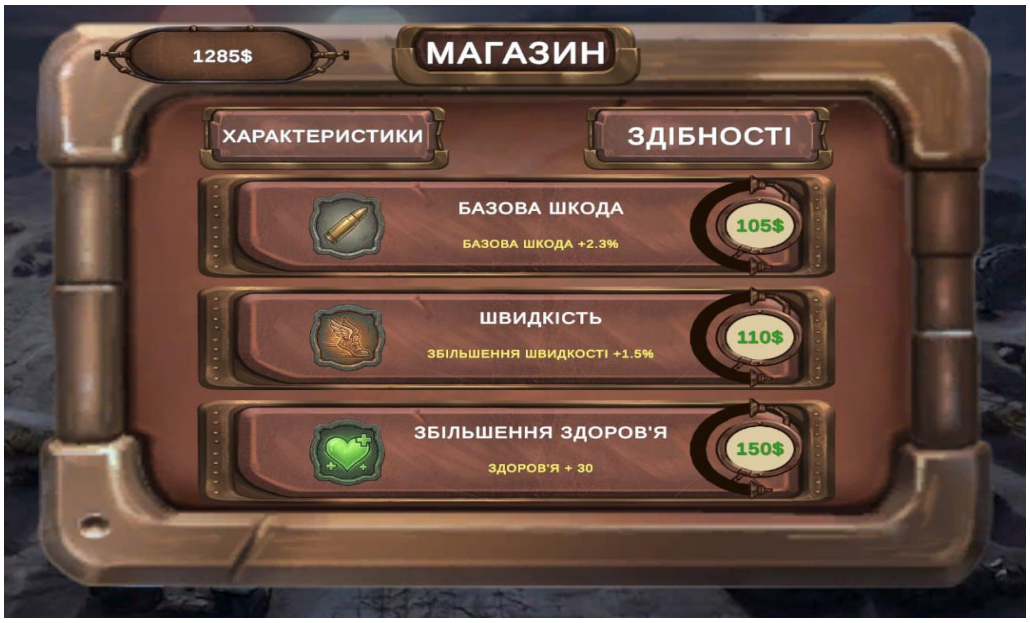

Рисунок 4.3 – Меню магазину характеристики

Коли гравець натискає у грі на кнопку торгового інтерфейсу відкривається меню «Магазин» (рис. 4.3), розділене на дві основні сутності. Перша сутність «Характеристики» відображає поточний баланс грошей гравця у верхньому кутку (наприклад, 1285$) та три модульні блоки для здійсненя прокачки ігрового персонажа – «Базова шкода» (збільшення атаки на 2.3%), «Швидкість» (модифікатор бігу на 1.5%) та «Збільшення здоров'я» (додавання +30 одиниць до максимального здоров’я). Ціна кожного покращення окремо відображується та виводиться на кнопках з ціною та зеленого кольору.

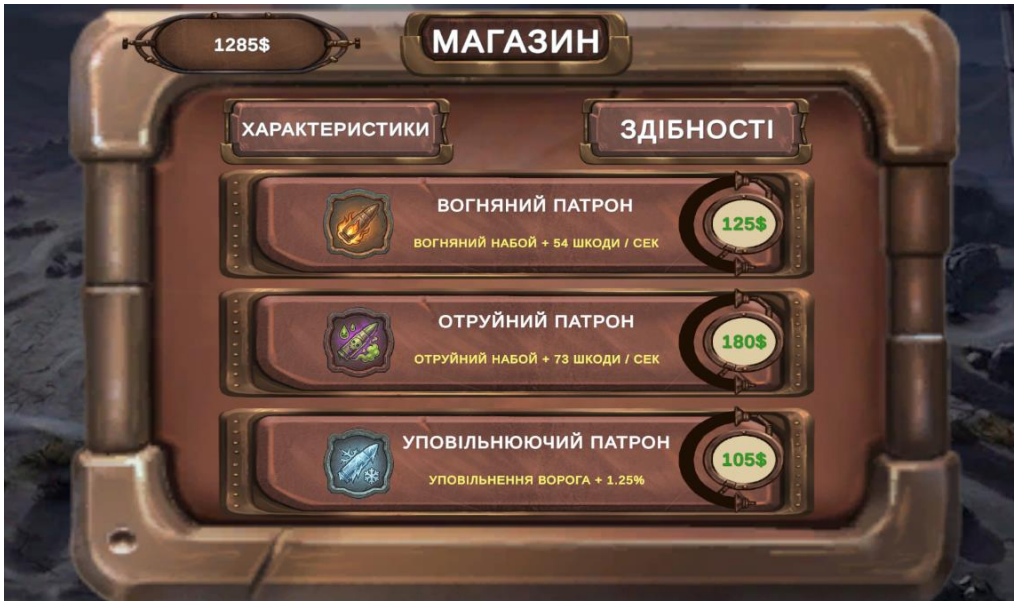

Рисунок 4.4 – Меню магазину здібностей

На другій вкладці магазину під назвою «Здібності» (рис. 4.4) реалізовано механіку придбання та еволюції спеціальних типів боєприпасів для зброї. Гравець за накопичену валюту може розблокувати «Вогняний патрон» (+54 шкоди/сек ефектом горіння), «Отруйний патрон» (+73 шкоди/сек ефектом отрути) та «Уповільнюючий патрон», який накладає на NavMesh-агентів ворогів дебаф швидкості пересування для зменщеня їх швидкості та маневреності.

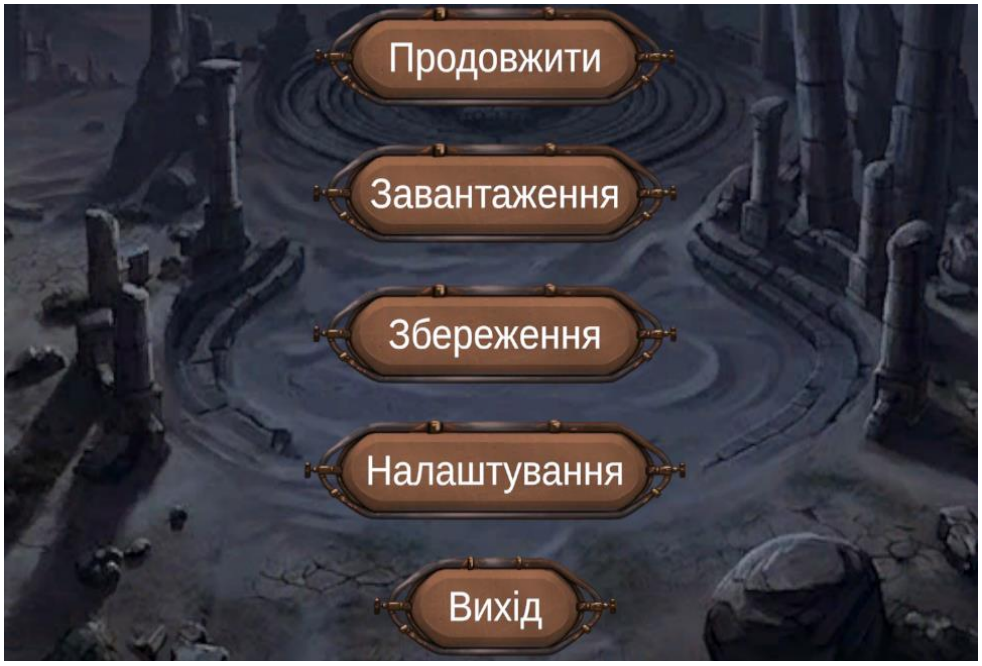

Рисунок 4.5 – Меню паузи

Під час активного бою при натискання клавіші «Меню» викликає меню паузи (рис. 4.5). Воно містить вертикальний набір кнопок – «Продовжити», «Завантаження», «Збереження», «Налаштування» та «Вихід». Цей інтерфейс повністю зупиняє ігровий таймер, звільняє курсор миші та дозволяє гравцеві зберегти поточний прогрес або скоригувати параметри чутливості безпосередньо під час проходження рівня.

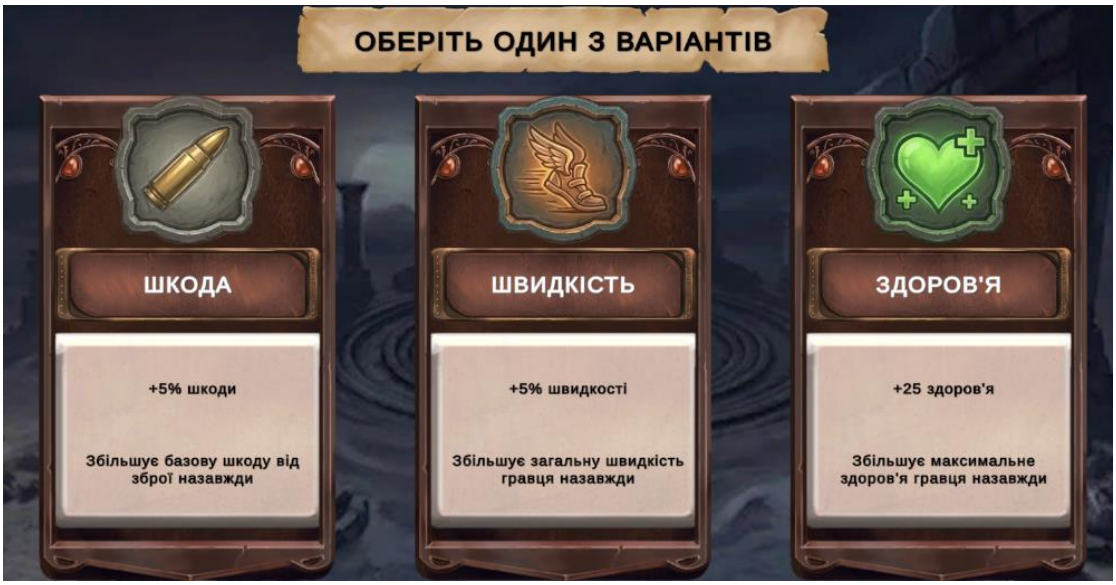

Рисунок 4.6 – Випадкові покращення та головна винагорода

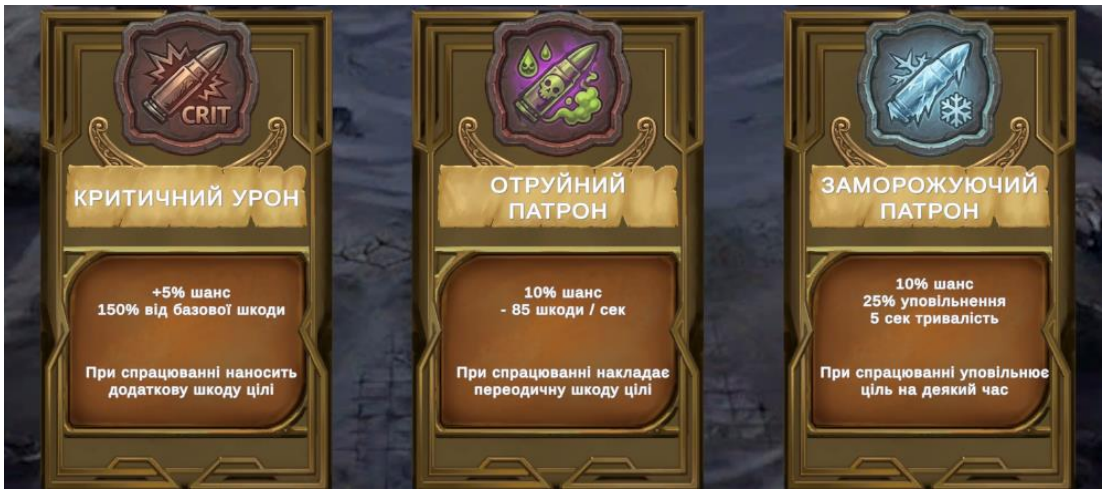

Рисунок 4.7 – Випадкові покращення та головна винагорода

Окремим елементом ігрового дизайну виступають системи випадкових нагород, що реалізовані на рис. 4.6 та рис. 4.7. Екран «Оберіть один з варіантів» генерує перед користувачем три випадкові інтерактивні картки. Це можуть бути як модифікатори («Шкода +5%», «Швидкість +5%», «Здоров'я +25»), так і складні ймовірнісні перки, що випадають як нагорода після вбивства головного боса рівня (рис. 4.7) – «Критичний урон» (+5% шанс криту, 150% урон), «Отруйний патрон» (10% шанс на урон 85/сек) або «Заморожуючий патрон» (10% шанс уповільнити ціль на 25% на 5 секунд).

Як вже було описано вище, створення зручного та цікавого UI-інтерфейсу потрібне для дотримання принципів ергономічності й доступності для

користувача. В результаті маємо такий сценарій гри – гравець знищує ворогів на рівні, заробляє початковий капітал, відкриває меню магазину або оверлей випадкових карт при підвищенні рівня, та покращує базові параметри активних навичок та зброї або купує статусний тип набоїв. Це створює варіативність бойової системи та мотивує до подальшого проходження гри.

4.2 Програмна реалізація UI-елементів до застосунку

Програмна реалізація UI-елементів до застосунку полягає в створенні логіки, що керує взаємодією між користувачем та грою через інтерфейс. Це охоплює такі аспекти, як обробка введення, відображення графічних компонентів і реакція на події.

Головним ядром збереження даних та економіки став менеджер UpgradeManager.cs, побудований за патерном Синглтон (Singleton). Він не знищується при перезавантаженні сцен (DontDestroyOnLoad) і зберігає глобальні змінні грошей (currentMoney), очок прокачки (upgradePoints), рівнів досвіду, а також відсоткові коефіцієнти покращень характеристик зброї та персонажа [Додаток А].

Рисунок 4.8 – Структура інтерфейсу HUD

На рис. 4.8 показано структуру інтерфейсу HUD під час активного геймплею. Гравець бачить поточний стан боєкомплекту та динамічну смугу здоров’я, яка автоматично оновлюється при отриманні пошкоджень або активації регенерації.

Для виведення числових значень модифікаторів на екран меню Tab було розроблено скрипт TabMenuController.cs. На відміну від стандартних систем, де опитування клавіш відбувається всередині самого оверлею, тут логіка перехоплення клавіші Tab винесена у скрипт CanvasSpawner.cs на об'єкті гравця. Якщо меню вимкнене, воно взагалі не витрачає ресурси процесора в Update-циклі. При виклику методу ToggleMenu() контролер зчитує змінні з UpgradeManager.Instance, переводить float-коефіцієнти (наприклад, 0.15f) у відсотки (+15%) та виводить їх у текстові поля TextMeshPro. Також метод повністю керує станом миші, звільняючи її курсор для кліків по кнопках магазину та блокуючи обертання камери гравця [Додаток А].

Рисунок 4.9 – Меню характеристик гравця у грі

Для графічного відображення життєвих показників гравця та тривимірних шкал над головами противників було написано універсальний скрипт HealthBar.cs.

Програмна логіка відображення життєвих сил гравця побудована на основі універсального скрипта HealthBar.cs [Додаток Б]. При ініціалізації у методі Start() скрипт виконує перевірку приналежності до типу об'єкта за допомогою методу GetComponentInParent<EnemyHealth>(). Якщо цей компонент відсутній (повертає null), скрипт ідентифікує об'єкт як самого гравця і через пряме посилання FindFirstObjectByType<FPSCharacter>().GetComponent<PlayerHealth>() підключається до його пулу здоров'я.

Рисунок 4.10 – Шкала здоров’я

За виведення прогресу розвитку персонажа та індикацію його поточного рівня відповідає автономний скрипт ExpBar.cs [Додаток Б]. Він працює у прямій зв'язці з глобальним синглтоном UpgradeManager.Instance, що усуває необхідність проміжних обчислень.

Оскільки змінні накопиченого досвіду (currentExperience) та ліміту для переходу на наступний рівень (experienceToLevelUp) збережені у менеджері у форматі цілих чисел (int), для уникнення логічної помилки цілочисельного ділення в Unity (яка завжди повертає нуль при меншому діленому) у коді застосовано явне приведення типів до float. Скрипт розраховує коефіцієнт та оновлює графічний повзунок Slider (або Image.fillAmount), а також динамічно виводить текстове значення поточного рівня у поле levelText та числове відношення у полі expValuesText.

Рисунок 4.11 – Шкала досвіду

Реалізація інтерфейсу відображення набоїв зброї інтегрована безпосередньо у базову архітектуру системи стрільби. Лічильник боєкомплекту розділений на два автономні класи, які керують різними текстовими полями TextMeshPro – TextAmmunitionCurrent.cs (патрони в поточному магазині) [Додаток Б] та TextAmmunitionTotal.cs (загальний запас патронів у резерві). Обидва класи наслідуються від базового скрипта ElementText та оновлюють дані через перевизначений метод Tick(), який викликається ядром інтерфейсу.

Рисунок 4.12 – Лічильник набоїв

Синхронізація текстових полів із фактичним станом боєкомплекту активної зброї відбувається в реальному часі безпосередньо під час виконання бойових дій персонажа . Зокрема, під час кожного пострілу метод Fire() у класі Weapon зменшує внутрішнє значення поточної кількості набоїв у магазині, що миттєво відображається на HUD-панелі завдяки динамічному оновленню через TextAmmunitionCurrent. У момент завершення анімації перезарядки, яка контролюється відповідними подіями аніматора у FPSCharacter, викликається метод FillAmmunition(), що відновлює максимальну місткість магазину на основі конфігураційного префаба Magazine . Таке розділення лічильників на автономні класи дозволяє повністю відокремити логіку обробки подій від безпосереднього рендерингу графічного інтерфейсу, що суттєво знижує навантаження на центральний процесор і запобігає зайвим перевикликам методів у кожному кадрі.

4.3 Програмна реалізація основних механік

Було реалізовано логіку ворогів за допомогою скрипту EnemyAI. Для переміщення противника використовується компонент NavMeshAgent, який

дозволяє ворогу рухатися по навігаційній сітці рівня після запечення карти. У скрипті створено декілька станів поведінки ворога – патрулювання, переслідування гравця та втрата цілі. Такий підхід дозволяє зробити поведінку ворога більш природнюю, оскільки він не просто стоїть на місці, а переміщується по зоні, реагує на появу гравця та повертається до патрулювання після втрати цілі [Додаток В].

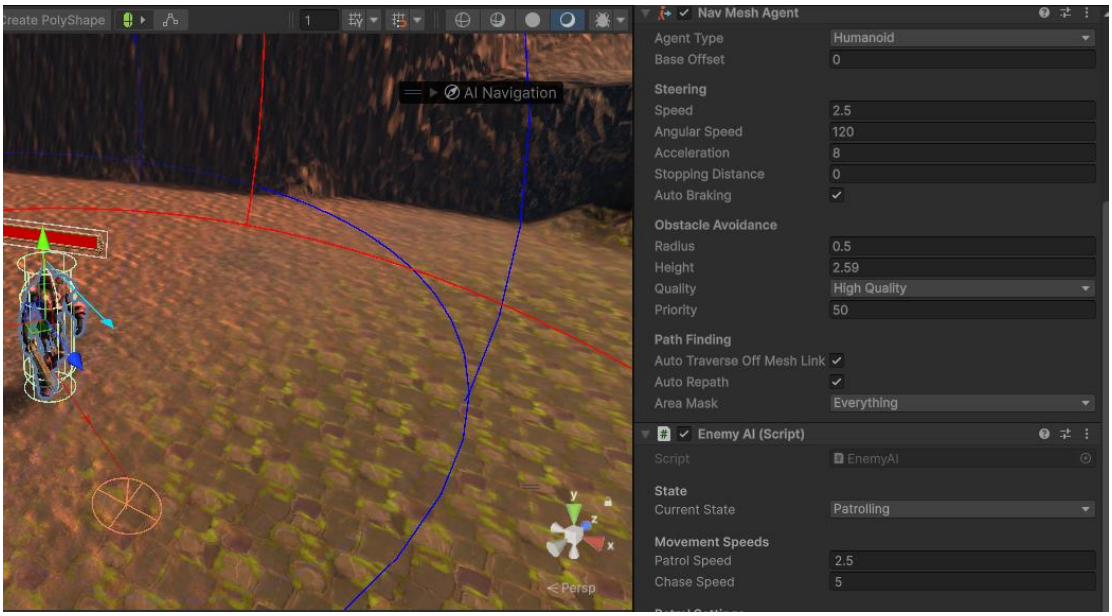

Рисунок 4.13 – Компонент NavMeshAgent

Під час патрулювання ворог обирає випадкову точку в межах заданого радіуса за допомогою NavMesh.SamplePosition, рухається до неї, очікує певний час, після чого отримує нову точку призначення. Для пошуку гравця використовується Physics.OverlapSphere, який перевіряє простір навколо ворога в межах радіуса агресії. Якщо гравець потрапляє в цей радіус, ворог переходить у стан переслідування, збільшує швидкість руху та встановлює позицію гравця як ціль для NavMeshAgent.

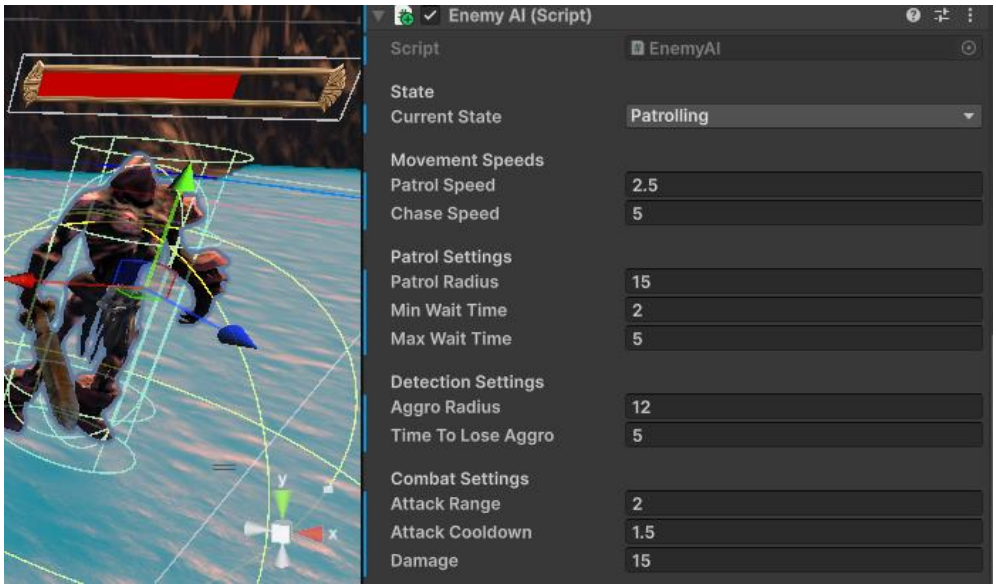

Рисунок 4.14 – Компонент скрипту ворога

Коли ворог наближається до гравця на достатню відстань, він переходить до атаки. Для цього у скрипті задано параметри attackRange, attackCooldown та damage. Якщо гравець знаходиться в радіусі атаки, ворог зупиняється, повертається у напрямку гравця та запускає анімацію атаки через параметр Attack в Animator. Після цього у гравця викликається метод TakeDamage, який зменшує його здоров’я. Якщо гравець виходить за межі радіуса агресії, ворог переходить у стан втрати цілі, де ще деякий час буде намагатись знайти гравця та починає рухаєтися до останньої відомої позиції гравця. Якщо за цей час гравець знову не потрапляє в радіус виявлення, ворог повертається до свого звичного патрулювання.

Реалізація переміщення гравця та його управління

Управління персонажем повністю переведено на **пряме опитування пристроїв введення** (Keyboard.current та Mouse.current) через нову систему *Unity Input System*. Ми позбулися важкої подійної моделі (Callbacks) та збираємо дані введення лінійно в методі Update.

– **зчитування осей –** рух (клавіші W, A, S, D) трансформується у двовимірний вектор axisMovement. Рух миші записується безпосередньо через дельту зміщення (Mouse.current.delta.ReadValue()) у вектор axisLook;

– **блокування миші – у**правління працює лише тоді, когда курсор заблоковано в ігровій сцені (cursorLocked = true). Натискання на клавішу Escape звільняє мишу (наприклад, для кліків по вкладках меню).

FPSCharacter.cs (Мозок гравця)

Цей скрипт є центральним хабом («центром керування») для всього, що стосується логіки взаємодії персонажа [Додаток Г].

За що відповідає:

– постійне опитування клавіатури та миші для отримання чистих векторів введення;

– зберігання та перемикання станів гравця – прицілювання (isAiming), біг (isRunning), стрибок;

– управління зброєю (стрільба з урахуванням затримки, ініціалізація перезарядки, огляд зброї);

– передача параметрів швидкості руху та прицілювання в Animator для плавного відтворення анімацій;

– реініціалізація скриптів нової зброї через механізм рефлексії C#.

FPSMovement.cs (Фізичне тіло гравця)

Цей скрипт відповідає суто за фізичну взаємодію з ігровим світом за допомогою компонентів Rigidbody та CapsuleCollider [Додаток Г].

За що відповідає:

– отримання напрямку руху від FPSCharacter та розрахунок фінальної швидкості;

– розрахунок швидкостей – скрипт перевіряє, чи біжить гравець , чи цілиться він (швидкість ділиться навпіл за допомогою aimMovementMultiplier), а також множить результат на глобальний бонус швидкості з UpgradeManager.Instance.movementSpeedBonusPercentage;

– трансформація напрямку з локальних координат гравця у світові (transform.TransformDirection) та пряме керування швидкістю Rigidbody (rigidBody.linearVelocity);

– кастомний розрахунок гравітації (прискорення падіння через gravityMultiplier) та обробка імпульсу стрибка;

– **перевірка землі (Grounded Check) –** через метод OnCollisionStay та невиділяючий пам'ять Physics.SphereCastNonAlloc скрипт перевіряє, чи стоїть гравець на підлозі. Метод IsGrounded() передає цю інформацію назад у FPSCharacter , щоб повністю заблокувати баг фантомного стрибка в повітрі.

FPSCameraLook.cs

Цей скрипт повністю керує обертанням камери гравця та синхронізацією повороту його фізичного тіла [Додаток Г].

– **по вертикалі (Y / Pitch) –** скрипт зчитує вертикальне зміщення миші, накопичує його в змінній xRotation та жорстко обмежує кути огляду (вгору/вниз) параметрами minPitch та maxPitch (за замовчуванням від -80 до 80 градусів). Це обертання застосовується лише локально до самої камери;

– **по горизонталі (X / Yaw) – г**оризонтальний рух миші повертає не камеру, а безпосередньо весь Rigidbody гравця навколо осі Vector3.up. Завдяки цьому персонаж завжди дивиться туди, куди спрямована камера;

– **режими обертання – с**крипт підтримує прапорець smooth. Якщо він увімкнений, поворот камери та тіла згладжується за допомогою Quaternion.Slerp з урахуванням швидкості інтерполяції. Якщо вимкнений – поворот відбувається миттєво і реактивно, слідуючи за рукою гравця.

Система стрільби, попадання та нанесення урону

Процес від натискання кнопки до вилітання цифр урону у нас розділений на 4 етапи [Додаток Д].

Ініціація вистрілу

Коли утримується або натискається ліва кнопка миші, скрипт розраховує темп вогню. Базова скорострельність зброї (equippedWeapon.GetRateOfFire()) динамічно розганяється за рахунок процентного модифікатора прокачки UpgradeManager.Instance.fireRateBonusPercentage. Якщо затримка пройшла – викликається метод Fire().

Спавн снаряду (Weapon)

Всередині Weapon.cs метод Fire запускає фізичний промінь Physics.Raycast через центр екрана камери гравця на максимальну дистанцію зброї.

– якщо промінь зустрічає перешкоду, пуля розвертається точно в точку влучання (hit.point). Якщо шлях чистий – пуля летить прямо вперед по камері;

– скрипт створює (Instantiate) фізичний об'єкт пулі з префабу prefabProjectile і задає йому високу початкову швидкість через Rigidbody – transform.forward * projectileImpulse.

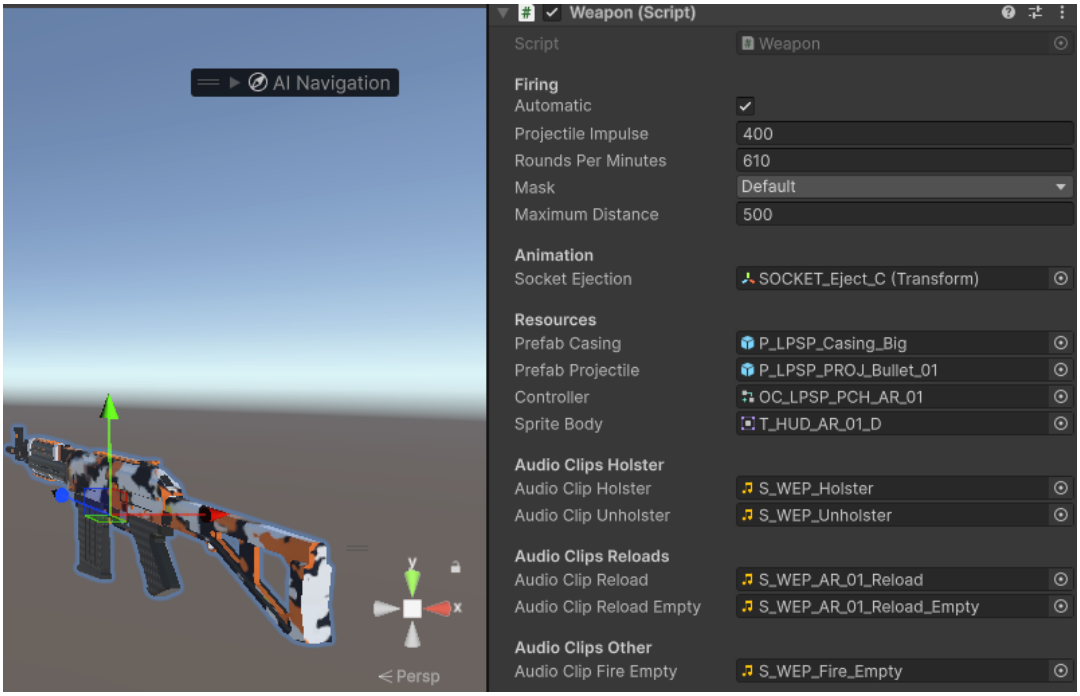

Рисунок 4.15 – Компонент зброї

Також було реалізовано необдхідних звуків для зброї: стрільба, перезарядка, та різні зміни станів.

Реєстрація влучання та розрахунок криту (Projectile)

Коли пуля фізично стикається з об'єктом у методі OnCollisionEnter, вона ігнорує інші кулі та перевіряє наявність компонента EnemyHealth на об'єкті влучання.

– **напрямок удару –** записується вектор transform.forward пулі в момент зіткнення;

– **розрахунок урону –** базовий урон пулі (25.0f) збільшується на відсоток прокачки bonusDamagePercentage з менеджера покращень;

– **перевірка на крит –** за допомогою функції Random.value прораховується ймовірність критичного удару. Якщо випадкове число менше або дорівнює UpgradeManager.Instance.critChance, фінальний урон множиться на показник криту critDamageMultiplier, а прапорець isCrit стає рівним true;

– Урон передається у ворога, а пуля знищується.

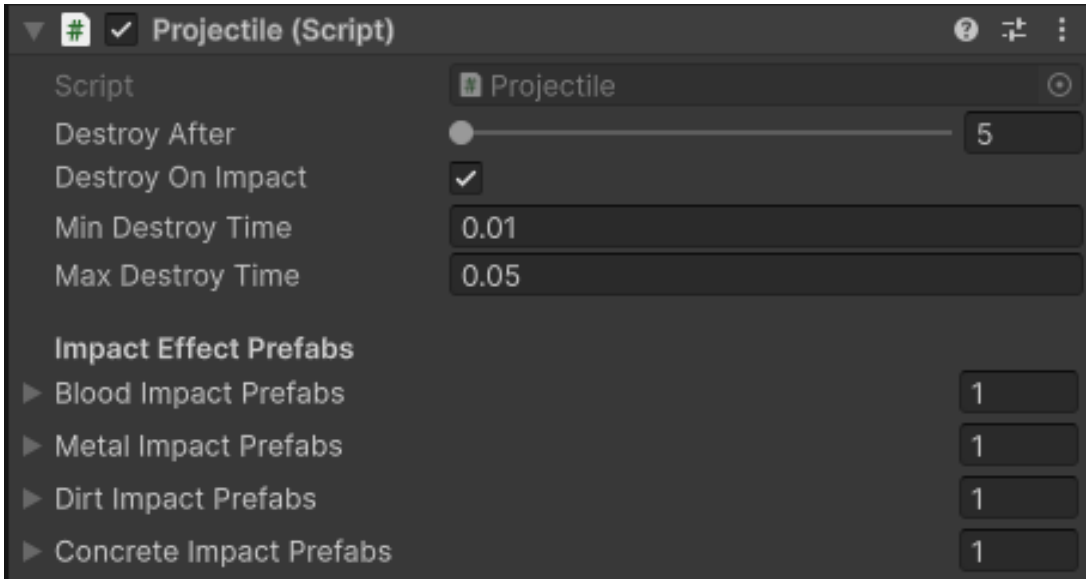

Рисунок 4.16 – Компонент пулі на інпекторі

Кількість патронів у магазині описується у скрипті Magazine. Він зберігає максимальну кількість боєприпасів та спрайт для інтерфейсу. Дульні ефекти реалізовані у скрипті Muzzle, який відповідає за точку пострілу, звук пострілу, частинки спалаху та короткочасне світло від пострілу. Завдяки цьому постріл має не тільки механічний ефект, але й візуальне та звукове оформлення. Для зміни зброї використовується скрипт Inventory. У ньому всі об’єкти зброї отримуються з дочірніх об’єктів персонажа. На початку гри вся зброя вимикається, після чого активується тільки вибрана. При зміні зброї поточна зброя вимикається, а нова активується. Це дозволяє легко додавати нові види зброї без значної зміни коду.

Застосування урону та реакція (EnemyHealth)

Отримавши дані, скрипт EnemyHealth.cs віднімає урон від поточного здоров'я ворога (currentHealth).

– **ефект урону –** викликається метод ShowDamagePopup, який створює об'єкт спливаючих цифр урона. Якщо у виклику передано прапорець isCrit = true,

скрипт автоматично перефарбовує цифри у червоний/золотий колір і збільшує розмір шрифту (transform.localScale *= 1.6f);

- **агр штучного інтелекту –** якщо ворог вижив, скрипт через команду enemyAI.OnDamagedByPlayer() миттєво переводить ІІ ворога в стан погоні за гравцем, навіть якщо він стояв спиною;
- **смерть та Лут –** якщо здоров'я падає до 0, викликається метод Die(). Ворог випадковим чином розраховує нагороду досвіду та грошей відповідно до своєї складності (Normal, Elite, Miniboss, Boss). Опыт нараховується в UpgradeManager , а на підлогу з ефектом фізичного фонтану викидається префаб грошей з записаною сумою всередині.

Система здоров’я гравця реалізована у скрипті PlayerHealth. У ньому зберігається базове максимальне здоров’я, поточне здоров’я та динамічний максимум, який може змінюватися залежно від покращень. Метод TakeDamage зменшує здоров’я гравця, а якщо воно дорівнює нулю, викликається метод смерті гравця. Також реалізовано автоматичну регенерацію, яка працює за умови, що у UpgradeManager задано значення відновлення здоров’я за секунду. При купівлі покращення максимального здоров’я викликається метод UpdateMaxHealthFields, який перераховує новий максимум здоров’я та за потреби додає гравцю різницю між старим і новим значенням.

Також було розроблено звуки у грі за допомогою компоненту Audio Source у інспекторі та підготовлені звукові дорожки для різних взаємодій. Таким чином, за допомогою коду: «audioSource.Play();», де audioSource прив’язка до компоненту, відбувається програвання звуку.

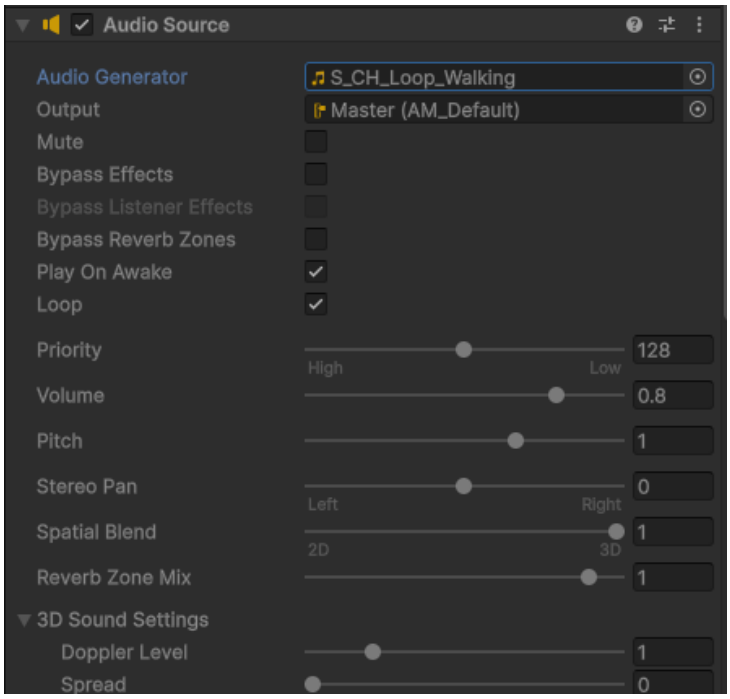

Рисунок 4.17 – Компонент Audio Source

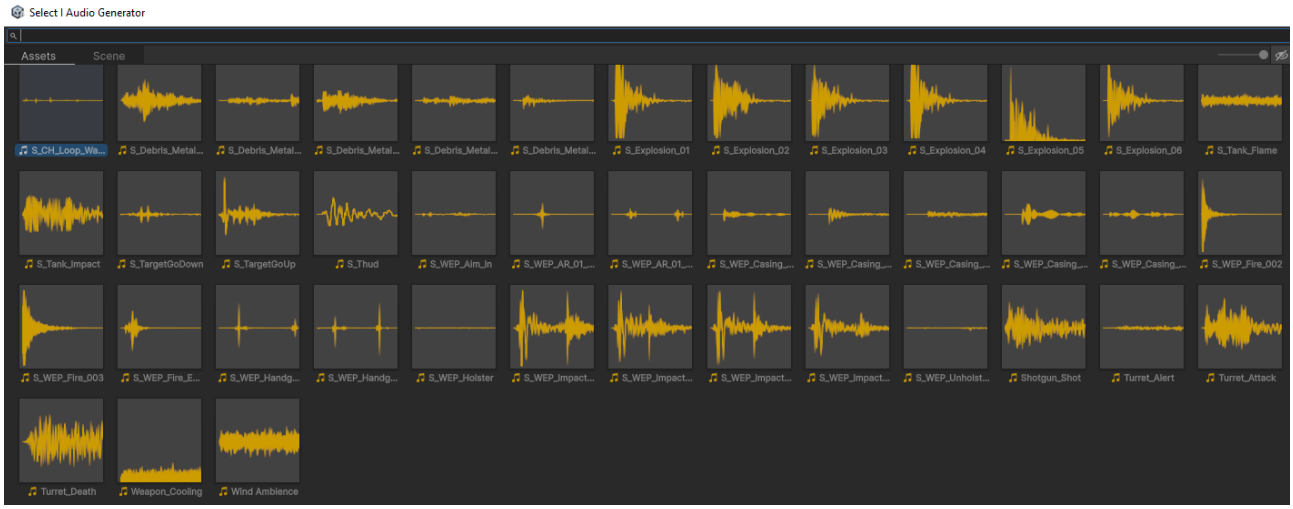

Рисунок 4.18 – Всі звуки

Динамічна зміна Animator Controller

Замість створення одного гігантського аніматора для всіх видів зброї, було застосовано більш чистий та продуктивний підхід – перезапис контролерів на льоту:

– **налаштування зброї** – кожна зброя у своєму скрипті Weapon зберігає персональне посилання на RuntimeAnimatorController. У цьому

контролері запечені унікальні анімації перезарядки, стрільби та утримання саме для цієї моделі;

– **зміна в коді** – у скрипті FPSCharacter під час екіпірування зброї спрацьовує метод RefreshWeaponSetup(). Рядок characterAnimator.runtime AnimatorController = equippedWeapon.GetAnimatorController(); миттєво підміняє логіку аніматора гравця під поточну зброю.

Налаштування шарів (Layers) в Інспекторі

Animator Controller використовує багатошарову структуру. Кожен шар відповідає за свою ізольовану частину тіла або тип дій:

– **Base Layer (Базовый шар)** – відповідає за рух ніг та нижню частину тіла персонажа (Idle, Walk, Run).

– **Layer Holster (Слой кобури)** – керує анімаціями хования (Holster) та діставання (Unholster) зброї.

– **Layer Actions (Слой дій)** – перекриває верхню часть тіла для відтворення важких ізольованих процесів, таких як перезарядка (Reload) або огляд зброї (Inspect).

– **Layer Overlay (Слой перекриття)** – найвищий шар із мінімальною вагою або аддитивним змішуванням. Використовується для миттєвих ефектів, таких як віддача при стрільбі (Fire).

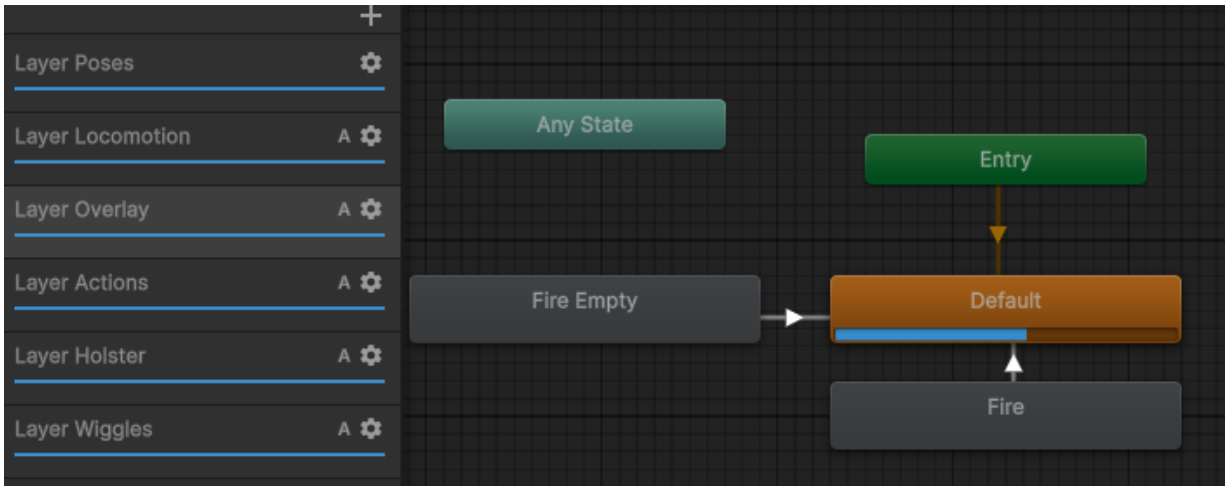

Рисунок 4.19 – Слої аніматора

Всі додаткові шари налаштовані в режимі Override (Заміщення) или Additive (Додавання) та використовують Avatar Mask (Маски Аватара). Маска відсікає ноги, дозволяючи персонажу повноцінно бігти на базовому шарі, поки руки на шарі Actions перезаряджають автомат.

Параметри та логіка переходів (StateMachine)

Усередині Animator Controller налаштовано чотири основні параметри, якими FPSCharacter керує в методі UpdateAnimator():

- **Movement (Float)** – розраховується як сума модулів осей переміщення x+y. Усередині контролера цей параметр керує деревом змішування (Blend Tree), плавно переводячи анімацію рук із Idle в Walk.
- **Aiming (Float)** – числовий коефіцієнт від 0.0f до 1.0f. Керує змішуванням анімації утримання зброї від стегна до утримання через мушку прицела.
- **Aim та Running (Bool)** – логічні перемикачі, які використовуються як умови (Conditions) для жорсткого переходу в спеціалізовані стани бігу або прицілу.

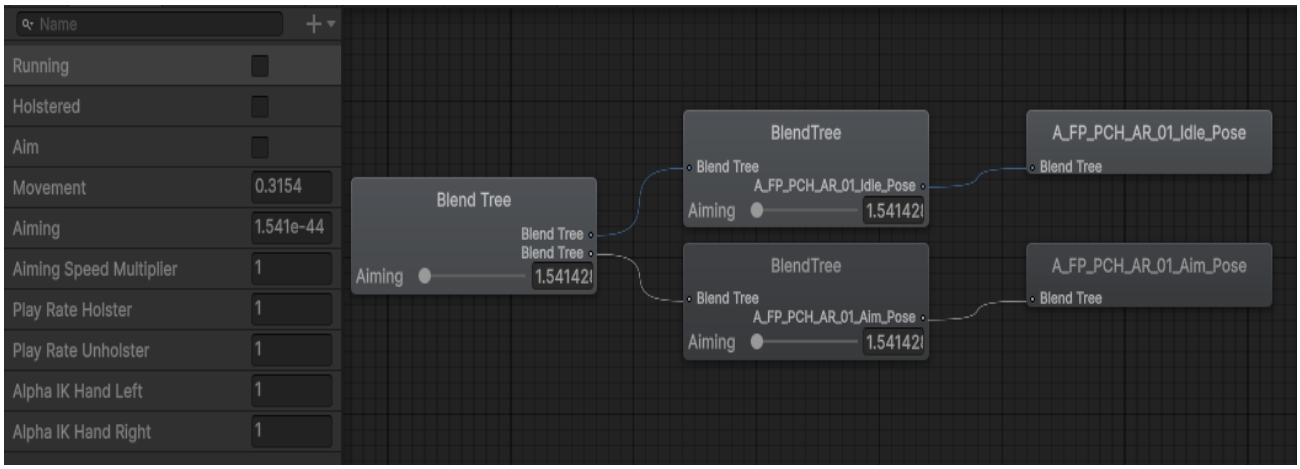

Рисунок 4.20 – Параметри аніматора та дерево переходів станів

Прямий виклик анімацій із C#

Для унікальних тригерних дій (стрільба, перезарядка) State Machine не використовується. Код викликає їх напряму, минаючи стрілочки переходів:

- characterAnimator.Play(stateName, layerActions, 0.0f); – миттєво вмикає анімацію перезарядки на потрібному шарі;

– characterAnimator.CrossFade("Fire", 0.05f, layerOverlay, 0); – плавно за 0.05 секунди підмішує анімацію віддачі.

Інверсна кінематика рук (CharacterKinematics)

Скрипт CharacterKinematics.cs – це просунута математична система, яка змушує руки персонажа ідеально слідувати за цівкою та руків'ям зброї, хоч би як зброя тряслася при анімаціях.

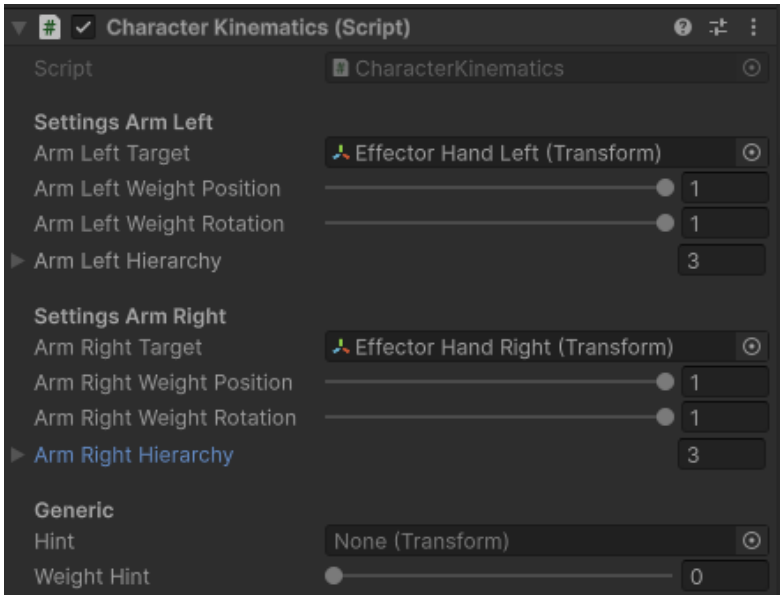

Рисунок 4.21 – Компонент скрипту CharacterKinematics

[Гравець: Animator] ──> Відтворює анімацію зброї (трясіння, віддача)
│
[CharacterKinematics] <──┘ Зчитує положення Sockets на зброї
├──> Обчислює кути ліктів і плечей за теоремою косинусів
└──> Намертво притягує кістки рук до точок утримання ствола

Принцип роботи коду:

– **прив'язка кісток** – в інспекторі в масиви armLeftHierarchy та armRightHierarchy передаються три кістки руки: Плечо (Root), Передпліччя (Mid) та Кисть (Tip). У поля Target передаються пустушки-маркери (Sockets) з самої моделі зброї;

– **точка сходження (Target)** – метод Compute() викликається в LateUpdate гравця, коли звичайна анімація вже відпрацювала. Він бере позицію сокета зброї target.position та поточну позицію кисті;

– **метод TriangleAngle –** щоб рука не розтягувалася і згиналася природно, скрипт розраховує геометрію трикутника, утвореного плечем, ліктем і кистю. Кут ліктя обчислюється математично через арккосинус;

– **обертання ліктьового суглоба (deltaR)** – код розраховує вісь вигину Vector3.Cross(ab, bc). Отриманий кватернион deltaR примусово розгортає лікоть на потрібний кут.

Запобігання вивертанню рук – поле hint (зазвичай це невидима точка, винесена вбік і назад від персонажа) служить орієнтиром для ліктів. Якщо рука випрямляється в рівну лінію, математичний вектор вигину ліктя втрачається. У цей момент код перемикається на hint.position, примусово розгортаючи суглоб ліктя в бік хінта, що захищає графіку від жахливих заломів мешу.

Висновки до розділу 4

У четвертому розділі було детально описано процес програмної реалізації, розробки ігрового дизайну в єдиному художньому стилі та інтеграції ключових елементів користувацького інтерфейсу (UI) для 3D шутера від першої особи (FPS). Основну увагу приділено побудові головного меню, меню налаштувань, паузи, магазину характеристик і здібностей, оверлею вибору випадкових покращень, а також механікам переміщення персонажа, стрільби та штучного інтелекту ворогів EnemyAI на базі компонента NavMeshAgent.

Також велику увагу приділено деталям візуалізації, таким як шкали здоров’я HealthBar та досвіду ExpBar, лічильники набоїв, спливаючі цифри пошкоджень DamagePopup, фізичний розліт валюти та система інверсної кінематики рук CharacterKinematics. Для створення приємного ігрового досвіду реалізовано оптимізоване керування меню характеристик TabMenuController у зв'язці зі спавнером канвасу, впроваджено систему керування персонажем за допомогою Unity Input System із чітким розподілом обов'язків між логічним хабом FPSCharacter та фізичним тілом FPSMovement, а також інтегровано комплексне звукове оформлення через компоненти Audio Source.

Інтерфейс та основні геймплейні системи було створено за допомогою інструментів Unity UI, TextMeshPro та глобального менеджера UpgradeManager (Покращення всіх характеристик персонажа), побудованого за патерном Singleton, які забезпечують високу оптимізацію застосунку, інтуїтивно зрозумілу взаємодію гравця з грою, гнучкість конфігурацій усіх характеристик та комфортний ігровий процес.

ВИСНОВКИ

Під час виконання кваліфікаційної бакалаврської роботи було проведено аналіз актуальності сфери 3D-шутерів від першої особи та розглянуто декілька існуючих аналогів за обраним жанром. У ході роботи були детально проаналізовані їхні функціональні можливості, переваги та недоліки. Також був проведений системний аналіз для визначення цілей гри, потенційних користувачів та основних сценаріїв використання ігрового застосунку.

Крім того, було проаналізовано види засобів розробки для комп'ютерних ігор, зокрема Unity, Unreal Engine та CryEngine, зосередившись саме на створенні 3D-ігор. Був проведений огляд спеціальних програм для моделювання 3D-моделей, які необхідні для розробки гри, виконано порівняння доступних варіантів та здійснено вибір найкращого інструменту.

У рамках роботи було повністю реалізовано поставлене технічне завдання щодо створення ігрового застосунку. Правильно розроблений інтерфейс та логіка дозволили забезпечити гармонійне середовище, в якому гравці зможуть комфортно орієнтуватись та насолоджуватись грою. Для досягнення цієї мети були виконані такі важливі завдання:

- провести аналіз сучасних технологій у сфері розробки ігор;
- огляд аналогів жанру PVE Shooter;
- визначити основні функції майбутньої гри;
- оформлення специфікацій вимог до ПЗ;
- створення загального алгоритму розробки гри.

Результатом кваліфікаційної роботи є готовий ігровий застосунок, який відповідає поставленим вимогам та відповідає всім поставленим цілям.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Teaching computer game development with Unity engine: a case study / Н. В. Моісеєнко та ін. CEUR Workshop Proceedings, 2023. URL: https://elibrary.kdpu.edu.ua/xmlui/handle/123456789/8486 (дата звернення: 20.04.2026).

2. Valve. Left 4 Dead 2 on Steam. *Steam*. URL: https://store.steampowered.com/app/550/Left_4_Dead_2/ (дата звернення: 22.04.2026).

3. Tripwire Interactive. Killing Floor 2. *Steam*. URL: https://store.steampowered.com/app/232090/Killing_Floor_2/ (дата звернення: 22.04.2026).

4. Fatshark. Warhammer: Vermintide 2. *Steam*. URL: https://store.steampowered.com/app/552500/Warhammer_Vermintide_2/ (дата звернення: 22.04.2026).

5. ТЗ для гри, UX-проєктування та UX-прототипування ігор. *ArionisGames*. URL: https://arionisgames.com/uk/services/full-cycle-gamedev/technical-specifications/ (дата звернення: 22.04.2026).

6. Unity. *Unity Learn*. URL: https://learn.unity.com/pathway/unity-essentials (Accessed: 24.04.2026).

7. Unreal Engine. VOKI Games. URL: https://vokigames.com/ua/unreal-engine (дата звернення: 24.04.2026).

8. Ігровий движок вибрати: Unity, UDK або CryENGINE?. *3DAS*. URL: https://3das.com.ua/igrovij-dvizhok-vibrati-unity-udk-abo-cryengine/ (дата звернення: 25.04.2026).

9. C# Guide - .NET managed language - C#. *Microsoft Learn: Build with answers in reach*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (Accessed: 25.04.2026).).

10. What Is Visual Studio?. *Microsoft Learn: Build with answers in reach*. URL: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022 (Accessed: 25.04.2026).

11. Cinema 4D | 3D Animation & Modeling Software. *Maxon*. URL: https://www.maxon.net/en/cinema-4d?srsltid=AfmBOop9lIu1tsQSlxvIP_HqSwbmb08a8s8ISBhlDK1vOwOQ4iiDlldP (Accessed: 26.04.2026).

12. Blender. The Free and Open Source 3D Creation Software. blender.org. *Blender*. URL: https://www.blender.org/ (Accessed: 26.04.2026).

13. Кравченко С. М., Сугоняк І. І., Марчук Г. В., Гришкун Є. О., Венгловська Ю. М. UML-моделювання процесу проектування гри в жанрі головоломки. *Вчені записки ТНУ імені В. І. Вернадського. Серія: Технічні науки*. Т. 34, № 3. С. 157–162. DOI: https://doi.org/10.32782/2663-5941/2023.3.1/25.

14. UML-Based Software Product Line Engineering with SMarty / ed. by E. OliveiraJr. Cham : Springer, 2023. DOI: https://doi.org/10.1007/978-3-031-18556-4.

15. Кирийчук Д. Л. Інформатика, обчислювальна техніка та автоматизація. Вчені записки ТНУ імені В.І. Вернадського. Серія: технічні науки. Т. 30, № 6. С. 57-64 DOI: https://doi.org/10.32838/2663-5941/2019.6-1/11.

ДОДАТОК А

Скрипти управління характеристиками та їх покращеннями

UpgradeManager

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

    public int currentMoney = 1000; upgradePoints = 1; currentLevel = 1;
currentExperience = 0;experienceToLevelUp = 1000;
    public float bonusDamagePercentage = 0.0f;critChance = 0.10f;critDamageMultiplier
= 1.50f;fireRateBonusPercentage = 0.0f;reloadSpeedMultiplier = 1.0f;maxHealthBonus =
0.0f;healthRegenPerSecond = 0.0f;movementSpeedBonusPercentage = 0.0f;
    private void Awake()
    {
        if (Instance == null) {Instance = this; DontDestroyOnLoad(gameObject); }
        else{ Destroy(gameObject); }
    }
    public void UpgradeDamage(float amount) => bonusDamagePercentage += amount;
    public void UpgradeCritChance(float amount) => critChance += amount;
    public void UpgradeCritDamage(float amount) => critDamageMultiplier += amount;
    public void UpgradeFireRate(float amount) => fireRateBonusPercentage += amount;
    public void UpgradeReloadSpeed(float amount) => reloadSpeedMultiplier += amount;
    public void UpgradeRegen(float amount) => healthRegenPerSecond += amount;
    public void UpgradeMaxHealth(float amount)
    {
        maxHealthBonus += amount;
        PlayerHealth playerHP = FindFirstObjectByType<PlayerHealth>();
        if (playerHP != null) playerHP.UpdateMaxHealthFields();
    }
public void UpgradeMovementSpeed(float amount)=>movementSpeedBonusPercentage+=amount;
    public void AddExperience(int amount)
    {
        currentExperience += amount;
        while (currentExperience >= experienceToLevelUp)
        {
            currentExperience -= experienceToLevelUp;
            currentLevel++;
            upgradePoints++;
        }
    }
}
```

TabMenuController

```
using TMPro;
public class TabMenuController : MonoBehaviour
      {
        [SerializeField] private GameObject menuPanel;
        [SerializeField] private TMP_Text arBonusText;pistolBonusText;rpmBonusText;
critChanceStatText;critDamageStatText;reloadTimeBonusText;maxHPStatText;regenStatText;
moneyText;pointsText;
        private bool isMenuOpen = false;
        private FPSCharacter playerCharacter;

        void Start()
        {
            playerCharacter = FindFirstObjectByType<FPSCharacter>();
            if (menuPanel != null) menuPanel.SetActive(false);
            isMenuOpen = false;
        }
```

```
        public void ToggleMenu()
        {
            isMenuOpen = !isMenuOpen;
            if (menuPanel != null) menuPanel.SetActive(isMenuOpen);
            if (isMenuOpen) UpdateStatsUI();
            UpdateCursorAndPlayerState();
        }

        public void UpdateStatsUI()
        {
            if (UpgradeManager.Instance == null) return;
            UpgradeManager manager = UpgradeManager.Instance;
            arBonusText.text = $"+{manager.bonusDamagePercentage * 100:0}%";
            pistolBonusText.text = $"+{manager.bonusDamagePercentage * 100:0}%";
            rpmBonusText.text = $"+{manager.fireRateBonusPercentage * 100:0}%";
            critChanceStatText.text = $"{manager.critChance * 100:0}%";
            critDamageStatText.text = $"{manager.critDamageMultiplier * 100:0}%";
      reloadTimeBonusText.text = $"+{(manager.reloadSpeedMultiplier - 1f) * 100:0}%";
            PlayerHealth playerHealth = FindFirstObjectByType<PlayerHealth>();
     if (playerHealth != null) { maxHPStatText.text = $"{playerHealth.MaxHealth:0}"; }
            else { maxHPStatText.text = "100"; }
            regenStatText.text = $"{manager.healthRegenPerSecond:0} хп/с";
            moneyText.text = $"{manager.currentMoney}$";
        pointsText.text = $"У вас доступно {manager.upgradePoints} очок покращення";
        }

        private void UpdateCursorAndPlayerState()
        {
            if (isMenuOpen)
            {
                Cursor.visible = true;
                Cursor.lockState = CursorLockMode.None;

                if (playerCharacter != null)
                {
                    typeof(FPSCharacter).GetField("cursorLocked",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance)
                        ?.SetValue(playerCharacter, false);
                }
            }
            else { Cursor.visible = false;Cursor.lockState = CursorLockMode.Locked;
                if (playerCharacter != null)
                {
                    typeof(FPSCharacter).GetField("cursorLocked",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance)
                        ?.SetValue(playerCharacter, true);
                }
            }
        }
        public void OnUpgradeButtonClicked()
        {
            if (UpgradeManager.Instance == null) return;
            if (UpgradeManager.Instance.upgradePoints > 0)
            {
                UpgradeManager.Instance.upgradePoints--;
                UpgradeManager.Instance.UpgradeDamage(0.10f);
                UpdateStatsUI();
            }
        }
    }
```

ДОДАТОК Б

Скрипти відображення HUD елементів

HealthBar

```
using UnityEngine;
using UnityEngine.UI;

public class HealthBar : MonoBehaviour
{
    [SerializeField] private Image healthFillImage;
    [SerializeField] private Transform canvasTransform;

    private PlayerHealth playerHealth;
    private EnemyHealth enemyHealth;
    private Transform playerCameraTransform;

    void Start()
    {
        enemyHealth = GetComponentInParent<EnemyHealth>();
        if (enemyHealth != null)
        {
            FPSCharacter player = FindFirstObjectByType<FPSCharacter>();
            if (player != null && player.GetCameraWorld() != null)
            { playerCameraTransform = player.GetCameraWorld().transform;  }
            else if (Camera.main != null)
            { playerCameraTransform = Camera.main.transform; }
        }else playerHealth =
FindFirstObjectByType<FPSCharacter>().GetComponent<PlayerHealth>();
    }
    void Update()
    {
        UpdateBar();
        BillboardToPlayer();
    }
    private void UpdateBar()
    {
        float fillPercentage = 1f;

        if (playerHealth != null)
        {fillPercentage = playerHealth.CurrentHealth / playerHealth.MaxHealth; }
        else if (enemyHealth != null)
        { fillPercentage = enemyHealth.CurrentHealth / enemyHealth.MaxHealth;  }

        if (healthFillImage != null)
        {  healthFillImage.fillAmount = fillPercentage; }
    }
    private void BillboardToPlayer()
    {
 if (enemyHealth != null && canvasTransform != null && playerCameraTransform != null)
        {  canvasTransform.rotation = playerCameraTransform.rotation;  }    }
}
```

ExpBar

```
using UnityEngine.UI;
using TMPro;

    public class ExpBar : MonoBehaviour
    {
        [SerializeField] private Slider expSlider;
        [SerializeField] private Image expFillImage;
        [SerializeField] private TMP_Text levelText;
        [SerializeField] private TMP_Text expValuesText;
```

```
        void Update()
        {
            if (UpgradeManager.Instance == null) return;
            UpgradeManager manager = UpgradeManager.Instance;
float fillPercentage = (float)manager.currentExperience / manager.experienceToLevelUp;
            if (expSlider != null) { expSlider.value = fillPercentage;  }
            if (expFillImage != null) { expFillImage.fillAmount = fillPercentage; }
            if (levelText != null){levelText.text = $"Ур. {manager.currentLevel}";}
            if (expValuesText != null)
            {  expValuesText.text = $"{manager.currentExperience} /
{manager.experienceToLevelUp}"; }
        }    }     }
```

TextAmmunitionCurrent

```
using UnityEngine;
using System.Globalization;

    public class TextAmmunitionCurrent : ElementText
    {
        [SerializeField] private bool updateColor = true;
        [SerializeField] private float emptySpeed = 1.5f;
        [SerializeField] private Color emptyColor = Color.red;

        protected override void Tick()
        {
            float current = equippedWeapon.GetAmmunitionCurrent();
            float total = equippedWeapon.GetAmmunitionTotal();
            textMesh.text = current.ToString(CultureInfo.InvariantCulture);

            if (updateColor)
            {
                float colorAlpha = (current / total) * emptySpeed;
                textMesh.color = Color.Lerp(emptyColor, Color.white, colorAlpha);
            }
        }
    }
```

TextAmmunitionTotal

```
using System.Globalization;
    public class TextAmmunitionTotal : ElementText
    {
        protected override void Tick()
        {
            float ammunitionTotal = equippedWeapon.GetAmmunitionTotal();
            textMesh.text = ammunitionTotal.ToString(CultureInfo.InvariantCulture);
        }
    }
```

ДОДАТОК В

Скрипти системи ворогів

EnemyAI

```
using UnityEngine;
using UnityEngine.AI;
using System.Collections;

[RequireComponent(typeof(NavMeshAgent))]
public class EnemyAI : MonoBehaviour
{
    public enum AIState { Patrolling, Chasing, TargetLost }
    [SerializeField] private AIState currentState = AIState.Patrolling;

    [SerializeField] private float patrolSpeed = 2.5f;chaseSpeed = 5.0f;patrolRadius =
15.0f;minWaitTime = 2f;maxWaitTime = 5f;aggroRadius = 12.0f;timeToLoseAggro = 5.0f;
attackRange = 2.0f;attackCooldown = 1.5f;damage = 15.0f;
    [SerializeField] private Animator animator;
    private NavMeshAgent agent;
    private Transform playerTransform;
    private Vector3 startPosition;

    private float patrolWaitTimer;loseAggroTimer;nextAttackTime;

    void Awake()    {
        agent = GetComponent<NavMeshAgent>();
        if (animator == null) animator = GetComponentInChildren<Animator>();   }

    void Start()
    {
        startPosition = transform.position;
        agent.speed = patrolSpeed;
        StartCoroutine(ScanForPlayerRoutine());
        SetRandomPatrolDestination();
    }
    void Update()
    {
        switch (currentState)        {
            case AIState.Patrolling: PerformPatrolLogic(); break;
            case AIState.Chasing: PerformChaseLogic(); break;
            case AIState.TargetLost: PerformTargetLostLogic(); break;        }
        UpdateAnimationState();
    }
    private void PerformPatrolLogic()
    {
        if (!agent.pathPending && agent.remainingDistance <= agent.stoppingDistance)
        {
            patrolWaitTimer += Time.deltaTime;
            if (patrolWaitTimer >= Random.Range(minWaitTime, maxWaitTime))
            {
                SetRandomPatrolDestination();
                patrolWaitTimer = 0f;
            }
        }
    }
    private void PerformChaseLogic()
    {
        if (playerTransform == null) return;
        float distanceToPlayer = Vector3.Distance(transform.position,
playerTransform.position);

        if (distanceToPlayer <= attackRange)        {
```

```
            agent.isStopped = true;
            LookAtTarget(playerTransform.position);
            if (Time.time >= nextAttackTime){  ExecuteAttack(); }
        } else        {
            agent.isStopped = false;
            agent.destination = playerTransform.position;        }

        if (distanceToPlayer > aggroRadius)
        {
            agent.isStopped = false;
            currentState = AIState.TargetLost;
            loseAggroTimer = timeToLoseAggro;
        }
    }
    private void PerformTargetLostLogic()
    {
        if (playerTransform != null) {agent.destination = playerTransform.position;}
        loseAggroTimer -= Time.deltaTime;

        float distanceToPlayer = playerTransform != null ?
Vector3.Distance(transform.position, playerTransform.position) : float.MaxValue;
        if (distanceToPlayer <= aggroRadius)
        {
            currentState = AIState.Chasing;
            return;
        }
        if (loseAggroTimer <= 0f) { ForceReturnToPatrol();  }
    }
    private void UpdateAnimationState()
    {
        if (animator == null || agent == null) return;
        bool isMoving = agent.velocity.sqrMagnitude > 0.1f && !agent.isStopped;
        animator.SetBool("isMoving", isMoving);
    }
    private void ExecuteAttack()
    {
        nextAttackTime = Time.time + attackCooldown;
        if (animator != null) { animator.SetTrigger("Attack"); }
        if (playerTransform != null)
        {
            PlayerHealth pHealth = playerTransform.GetComponent<PlayerHealth>();
            if (pHealth != null) { pHealth.TakeDamage(damage); }
        }
    }
    private void SetRandomPatrolDestination()
    {
        Vector3 randomDirection = Random.insideUnitSphere * patrolRadius;
        randomDirection += startPosition;
        if (NavMesh.SamplePosition(randomDirection, out NavMeshHit hit, patrolRadius,
NavMesh.AllAreas)) { agent.destination = hit.position; }
    }
    private void ForceReturnToPatrol()
    {
        playerTransform = null;
        currentState = AIState.Patrolling;
        agent.speed = patrolSpeed;
        agent.isStopped = false;
        SetRandomPatrolDestination();
    }
    public void OnDamagedByPlayer()
    {
        if (currentState == AIState.Chasing) return;
        PlayerHealth player = FindFirstObjectByType<PlayerHealth>();
        if (player != null)        {
            playerTransform = player.transform;
```

```
            agent.speed = chaseSpeed;
            currentState = AIState.Chasing;        }
    }

    private IEnumerator ScanForPlayerRoutine()
    {
        while (true)
        {
            yield return new WaitForSeconds(1.0f);
       Collider[] colliders = Physics.OverlapSphere(transform.position, aggroRadius);

            foreach (var col in colliders)
            {
                PlayerHealth pHealth = col.GetComponent<PlayerHealth>();
                if (pHealth != null)  {
                    playerTransform = col.transform;
                    if (currentState == AIState.Patrolling)   {
                        agent.speed = chaseSpeed;
                        currentState = AIState.Chasing;  }  break;  }   }
        }
    }

    private void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.blue;
        Gizmos.DrawWireSphere(Application.isPlaying ? startPosition :
transform.position, patrolRadius);
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, aggroRadius);
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(transform.position, attackRange);
    }
}
```

EnemyHealth

```
public class EnemyHealth : MonoBehaviour
{
    public enum EnemyType { Normal, Elite, Miniboss, Boss }

    [SerializeField] private float maxHealth = 100.0f;
    [SerializeField] private GameObject damagePopupPrefab;
    [SerializeField] private EnemyType enemyType = EnemyType.Normal;
    [SerializeField] private GameObject moneyPrefab;
    private float currentHealth;
    private EnemyAI enemyAI;
    public float CurrentHealth => currentHealth;
    public float MaxHealth => maxHealth;

    private void Awake()
    {
        currentHealth = maxHealth;
        enemyAI = GetComponent<EnemyAI>();
    }
    public void TakeDamage(float amount, Vector3 hitDirection, bool isCrit = false)
    {
        currentHealth -= amount;
        currentHealth = Mathf.Clamp(currentHealth, 0, maxHealth);
        int damageInt = Mathf.RoundToInt(amount);
        ShowDamagePopup(damageInt, hitDirection, isCrit);
          if (enemyAI != null && currentHealth > 0) {enemyAI.OnDamagedByPlayer();}
        if (currentHealth <= 0) { Die(); }
    }

    private void ShowDamagePopup(int damage, Vector3 hitDirection, bool isCrit)
    {
```

```
          if (damagePopupPrefab != null) {
              GameObject popup = Instantiate(damagePopupPrefab, transform.position +
Vector3.up * 1.2f, Quaternion.identity);

              Vector3 randomDirection = hitDirection == Vector3.zero
           ? new Vector3(Random.Range(-1f, 1f), 0, Random.Range(-1f, 1f)).normalized
                 : new Vector3(hitDirection.x, 0, hitDirection.z).normalized;

           Vector3 moveDirection = (randomDirection + Vector3.up * 0.5f).normalized;
              DamagePopup popupScript = popup.GetComponent<DamagePopup>();
              if (popupScript != null)
           {
                 popupScript.Setup(damage, moveDirection);
               if (isCrit)
               {
                   TextMesh textMesh = popup.GetComponent<TextMesh>();
                   if (textMesh != null)
                   {
                       textMesh.color = Color.red;
                       popup.transform.localScale *= 1.6f;
                   }
               }
           }
       }
   }

   private void Die()
   {
       int expGained = 0;
       switch (enemyType)
       {
           case EnemyType.Normal: expGained = Random.Range(80, 131); break;
           case EnemyType.Elite: expGained = Random.Range(150, 211); break;
           case EnemyType.Miniboss: expGained = Random.Range(300, 401); break;
           case EnemyType.Boss: expGained = Random.Range(2000, 3501); break;
       }
       if (UpgradeManager.Instance != null)
       {
           UpgradeManager.Instance.AddExperience(expGained);
       }
       int moneyGained = 0;
       switch (enemyType)
       {
           case EnemyType.Normal: moneyGained = Random.Range(50, 81); break;
           case EnemyType.Elite: moneyGained = Random.Range(100, 161); break;
           case EnemyType.Miniboss: moneyGained = Random.Range(250, 401); break;
           case EnemyType.Boss: moneyGained = Random.Range(1500, 2501); break;
       }
       if (moneyPrefab != null)
       {
           GameObject droppedMoney = Instantiate(moneyPrefab, transform.position +
Vector3.up * 0.3f, Quaternion.identity);

           MoneyPickup pickupScript = droppedMoney.GetComponent<MoneyPickup>();
           if (pickupScript != null) { pickupScript.Initialize(moneyGained); }
       }
       Destroy(gameObject);
   }
}
```

ДОДАТОК Г

Скрипти системи управління персонажем

FPSCharacter

```
using System.Collections;
using UnityEngine.InputSystem;

    [RequireComponent(typeof(CharacterKinematics))]
    public sealed class FPSCharacter : CharacterBehaviour
    {
        [SerializeField] private InventoryBehaviour inventory;
        [SerializeField] private Camera cameraWorld;
        [SerializeField] private float dampTimeLocomotion = 0.15f;
        [SerializeField] private float dampTimeAiming = 0.3f;
        [SerializeField] private Animator characterAnimator;

        private bool isAiming;isRunning;jumpPressed;holstered;
        private float lastShotTime;
        private int layerOverlay;layerHolster;layerActions;

        private CharacterKinematics characterKinematics;
        private WeaponBehaviour equippedWeapon;
        private WeaponAttachmentManagerBehaviour weaponAttachmentManager;
        private ScopeBehaviour equippedWeaponScope;
        private MagazineBehaviour equippedWeaponMagazine;
        private bool reloading;inspecting;holstering;

        private Vector2 axisMovement;
        private Vector2 axisLook;
        private bool cursorLocked = true;

        private static readonly int HashAimingAlpha = Animator.StringToHash("Aiming");
        private static readonly int HashMovement = Animator.StringToHash("Movement");
        protected override void Awake()
        {
            cursorLocked = true;
            UpdateCursorState();
            characterKinematics = GetComponent<CharacterKinematics>();
            inventory.Init();
        }
        protected override void Start()
        {
            layerHolster = characterAnimator.GetLayerIndex("Layer Holster");
            layerActions = characterAnimator.GetLayerIndex("Layer Actions");
            layerOverlay = characterAnimator.GetLayerIndex("Layer Overlay");
            RefreshWeaponSetup();
        }
        protected override void Update()
        {
      if (Keyboard.current != null && Keyboard.current.escapeKey.wasPressedThisFrame)
            {   cursorLocked = !cursorLocked; UpdateCursorState(); }
            if (!cursorLocked) return;
            HandleInput();
  isAiming = Mouse.current != null && Mouse.current.rightButton.isPressed && CanAim();
            isRunning = Keyboard.current != null &&
Keyboard.current.leftShiftKey.isPressed && CanRun();

       if (Keyboard.current != null && Keyboard.current.spaceKey.wasPressedThisFrame)
            {
                FPSMovement movementComponent = GetComponent<FPSMovement>();
                if (movementComponent != null && movementComponent.IsGrounded())
```

```
                { jumpPressed = true; }
            }
            HandleShooting(); UpdateAnimator();
        }
        protected override void LateUpdate()
        {
            if (equippedWeapon == null) return;
            if (characterKinematics != null) characterKinematics.Compute();
        }
        private void HandleInput()
        {
            if (Keyboard.current == null || Mouse.current == null) return;
            Vector2 moveInput = Vector2.zero;
            if (Keyboard.current.wKey.isPressed) moveInput.y += 1f;
            if (Keyboard.current.sKey.isPressed) moveInput.y -= 1f;
            if (Keyboard.current.aKey.isPressed) moveInput.x -= 1f;
            if (Keyboard.current.dKey.isPressed) moveInput.x += 1f;
            axisMovement = moveInput;
            axisLook = Mouse.current.delta.ReadValue();
        }
        private void HandleShooting()
        {
            if (Mouse.current == null || equippedWeapon == null) return;
            if (Mouse.current.leftButton.isPressed)
            {
                if (CanPlayAnimationFire() && equippedWeapon.HasAmmunition() &&
equippedWeapon.IsAutomatic())
                {
                    float currentFireRate = equippedWeapon.GetRateOfFire() * (1.0f +
UpgradeManager.Instance.fireRateBonusPercentage);
              if (Time.time - lastShotTime > 60.0f / currentFireRate) { Fire(); }
                }
            }
            if (Mouse.current.leftButton.wasPressedThisFrame)
            {
                if (CanPlayAnimationFire())
                {
                    if (equippedWeapon.HasAmmunition())
                    {
                        if (!equippedWeapon.IsAutomatic())
                        {
                            float currentFireRate = equippedWeapon.GetRateOfFire() *
(1.0f + UpgradeManager.Instance.fireRateBonusPercentage);
                if (Time.time - lastShotTime > 60.0f / currentFireRate) {Fire();}}}
                    else{FireEmpty();}}
            }
        if (Keyboard.current.tKey.wasPressedThisFrame && CanPlayAnimationInspect())
            { Inspect(); }

            Vector2 scrollDelta = Mouse.current.scroll.ReadValue();
            if (Mathf.Abs(scrollDelta.y) > 0.1f && CanChangeWeapon()){
int indexNext = scrollDelta.y > 0?inventory.GetNextIndex():inventory.GetLastIndex();
                int indexCurrent = inventory.GetEquippedIndex();
           if (indexCurrent != indexNext)StartCoroutine(nameof(Equip), indexNext);}
        }

        public override Camera GetCameraWorld() => cameraWorld;
        public override InventoryBehaviour GetInventory() => inventory;
        public override bool IsCrosshairVisible() => !isAiming && !holstered;
IsRunning() => isRunning;IsAiming() => isAiming;IsCursorLocked() => cursorLocked;
IsTutorialTextVisible() => false;
public override Vector2 GetInputMovement() => axisMovement;GetInputLook() => axisLook;
        public override bool WasJumpPressed()
        {
            bool wasPressed = jumpPressed;
```

```
            jumpPressed = false;
            return wasPressed;
        }

        private void UpdateAnimator()
        {
            characterAnimator.SetFloat(HashMovement,
Mathf.Clamp01(Mathf.Abs(axisMovement.x) + Mathf.Abs(axisMovement.y)),
dampTimeLocomotion, Time.deltaTime);
            characterAnimator.SetFloat(HashAimingAlpha, Convert.ToSingle(isAiming),
0.25f / 1.0f * dampTimeAiming, Time.deltaTime);
            characterAnimator.SetBool("Aim", isAiming);
            characterAnimator.SetBool("Running", isRunning);
        }
        private void Inspect()
        {
            inspecting = true;
            characterAnimator.CrossFade("Inspect", 0.0f, layerActions, 0);
        }
        private void Fire()
        {
            lastShotTime = Time.time;
            equippedWeapon.Fire();
            characterAnimator.CrossFade("Fire", 0.05f, layerOverlay, 0);
        }
        private void PlayReloadAnimation()
        {
       string stateName = equippedWeapon.HasAmmunition() ? "Reload" : "Reload Empty";
            characterAnimator.Play(stateName, layerActions, 0.0f);
            characterAnimator.speed = UpgradeManager.Instance.reloadSpeedMultiplier;
            reloading = true;
            equippedWeapon.Reload();
        }
        private IEnumerator Equip(int index = 0)
        {
            if (!holstered)
            {
                SetHolstered(holstering = true);
                yield return new WaitUntil(() => holstering == false);
            }
            SetHolstered(false);
            characterAnimator.Play("Unholster", layerHolster, 0);

            inventory.Equip(index);
            RefreshWeaponSetup();
        }

        private void RefreshWeaponSetup()
        {
            if (inventory == null) return;
            equippedWeapon = inventory.GetEquipped();
            if (equippedWeapon == null) return;

            var weaponScript = equippedWeapon as Weapon;
            if (weaponScript != null)
            {
                var cameraField = typeof(Weapon).GetField("playerCamera",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance);
                if (cameraField != null && cameraWorld != null)
                {cameraField.SetValue(weaponScript, cameraWorld.transform); }

                var characterField = typeof(Weapon).GetField("characterBehaviour",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance);
                if (characterField != null)
                {characterField.SetValue(weaponScript, this);
```

```
                var startMethod = typeof(Weapon).GetMethod("Start",
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance);
         if (startMethod != null) {startMethod.Invoke(weaponScript, null);
            }

            characterAnimator.runtimeAnimatorController =
equippedWeapon.GetAnimatorController();
            weaponAttachmentManager = equippedWeapon.GetAttachmentManager();

            if (weaponAttachmentManager == null) return;
            equippedWeaponScope = weaponAttachmentManager.GetEquippedScope();
            equippedWeaponMagazine = weaponAttachmentManager.GetEquippedMagazine();
        }

        private bool CanPlayAnimationFire() => !holstered && !holstering && !reloading
&& !inspecting;
        private bool CanPlayAnimationReload() => !reloading && !inspecting;
        private bool CanChangeWeapon() => !holstering && !reloading && !inspecting;
        private bool CanPlayAnimationInspect() => !holstered && !holstering &&
!reloading && !inspecting;
        private bool CanAim() => !holstered && !inspecting && !reloading &&
!holstering && !isRunning;
        private bool CanRun()
        {
            if (inspecting || reloading || isAiming) return false;
            if (Mouse.current != null && Mouse.current.leftButton.isPressed &&
equippedWeapon != null && equippedWeapon.HasAmmunition()) return false;
            if (axisMovement.y <= 0 || Math.Abs(Mathf.Abs(axisMovement.x) - 1) <
0.01f) return false;
            return true;
        }

        public override void EjectCasing() { if (equippedWeapon != null)
equippedWeapon.EjectCasing(); }

        public override void FillAmmunition(int amount)
        { if (equippedWeapon != null) {equippedWeapon.FillAmmunition(amount);} }

        public override void SetActiveMagazine(int active)
        {
            if (equippedWeaponMagazine != null)
                equippedWeaponMagazine.gameObject.SetActive(active != 0);
        }

        public override void AnimationEndedInspect() => inspecting = false;
        public override void AnimationEndedHolster() => holstering = false;
    }
}
```

FPSMovement

```
using System.Linq;
using UnityEngine;

    [RequireComponent(typeof(Rigidbody), typeof(CapsuleCollider))]
    public class FPSMovement : MovementBehaviour
    {
        [SerializeField] private AudioClip audioClipWalking;audioClipRunning;
        [SerializeField] private float speedWalking = 5.0f;speedRunning = 8.0f;
aimMovementMultiplier = 0.5f;jumpForce = 5.0f;gravityMultiplier = 2.0f;

        [SerializeField] private FPSCharacter customPlayerCharacter;
        private Rigidbody rigidBody;
```

```
        private CapsuleCollider capsule;
        private AudioSource audioSource;
        private bool grounded;
        private readonly RaycastHit[] groundHits = new RaycastHit[8];

        protected override void Awake()
        {
if (customPlayerCharacter == null)customPlayerCharacter=GetComponent<FPSCharacter>();
            rigidBody = GetComponent<Rigidbody>();
            capsule = GetComponent<CapsuleCollider>();
            audioSource = GetComponent<AudioSource>();
        }
        protected override void Start()
        {
            if (rigidBody != null)
            {
                rigidBody.constraints = RigidbodyConstraints.FreezeRotation;
                rigidBody.useGravity = false;
            }
        }
        private void OnCollisionStay()
        {
            if (capsule == null) capsule = GetComponent<CapsuleCollider>();
            if (capsule == null) return;
            Bounds bounds = capsule.bounds;
            float radius = bounds.extents.x - 0.01f;
            Physics.SphereCastNonAlloc(bounds.center, radius, Vector3.down,
                groundHits, bounds.extents.y - radius * 0.5f, ~0,
QueryTriggerInteraction.Ignore);

            if (!groundHits.Any(hit => hit.collider != null && hit.collider !=
capsule))   return;
            for (var i = 0; i < groundHits.Length; i++)
                groundHits[i] = new RaycastHit();
            grounded = true;
        }
        private void MoveCharacter()
        {
            if (customPlayerCharacter == null || rigidBody == null) return;
            Vector2 frameInput = customPlayerCharacter.GetInputMovement();
            Vector3 movement = new Vector3(frameInput.x, 0.0f, frameInput.y);

            float currentSpeed = speedWalking;
            if (customPlayerCharacter.IsRunning())currentSpeed = speedRunning; }
      currentSpeed *= (1.0f + UpgradeManager.Instance.movementSpeedBonusPercentage);

            if (customPlayerCharacter.IsAiming())
            {currentSpeed *= aimMovementMultiplier; }

            movement *= currentSpeed;
            movement = transform.TransformDirection(movement);
            rigidBody.linearVelocity = new Vector3(movement.x,
rigidBody.linearVelocity.y, movement.z);

            if (grounded && customPlayerCharacter.WasJumpPressed())
            {
                rigidBody.AddForce(Vector3.up * jumpForce, ForceMode.VelocityChange);
                grounded = false;
            }
        }
        public bool IsGrounded() => grounded;
    }
}
```

FPSCameraLook

```
using UnityEngine;
    public class FPSCameraLook : MonoBehaviour
    {
        [SerializeField] private float mouseSensitivity = 0.1f; minPitch = -80f;
        [SerializeField] private float maxPitch = 80f;
        [SerializeField] private bool smooth;
        [SerializeField] private float interpolationSpeed = 25.0f;
        [SerializeField] private FPSCharacter customPlayerCharacter;

        private Rigidbody playerCharacterRigidbody;
        private float xRotation = 0f;
        private Quaternion rotationCharacter;
        private Quaternion rotationCamera;

        private void Awake()
        {
            if (customPlayerCharacter == null)
                customPlayerCharacter = FindObjectOfType<FPSCharacter>();

            if (customPlayerCharacter != null)
          playerCharacterRigidbody = customPlayerCharacter.GetComponent<Rigidbody>();
        }
        private void Start()
        {
            if (customPlayerCharacter == null) return;
            rotationCharacter = customPlayerCharacter.transform.localRotation;
            rotationCamera = transform.localRotation;
            xRotation = transform.localEulerAngles.x;
            if (xRotation > 180f) xRotation -= 360f;
        }
        private void LateUpdate()
        {
       if (customPlayerCharacter == null || playerCharacterRigidbody == null) return;

            Vector2 mouseDelta = customPlayerCharacter.IsCursorLocked() ?
customPlayerCharacter.GetInputLook() : Vector2.zero;
            float mouseX = mouseDelta.x * mouseSensitivity;
            float mouseY = mouseDelta.y * mouseSensitivity;

            xRotation -= mouseY;
            xRotation = Mathf.Clamp(xRotation, minPitch, maxPitch);

            Quaternion targetCameraRotation = Quaternion.Euler(xRotation, 0f, 0f);
            rotationCharacter *= Quaternion.Euler(0.0f, mouseX, 0.0f);
            if (smooth)
            {
                transform.localRotation = Quaternion.Slerp(transform.localRotation,
targetCameraRotation, Time.deltaTime * interpolationSpeed);

playerCharacterRigidbody.MoveRotation(Quaternion.Slerp(playerCharacterRigidbody.rotati
on, rotationCharacter, Time.deltaTime * interpolationSpeed));
            }
            else
            {
                transform.localRotation = targetCameraRotation;
               playerCharacterRigidbody.MoveRotation(playerCharacterRigidbody.rotation
* Quaternion.Euler(0.0f, mouseX, 0.0f));
            }        }    }
}
```

ДОДАТОК Д

Скрипти систем зброї та стрільби

Weapon

```
using UnityEngine;

    public class Weapon : WeaponBehaviour
    {
        [SerializeField] private bool automatic;
        [SerializeField] private float projectileImpulse = 400.0f;
        [SerializeField] private int roundsPerMinutes = 200;
        [SerializeField]private LayerMask mask;
        [SerializeField] private float maximumDistance = 500.0f;
        [SerializeField] private Transform socketEjection;
        [SerializeField] private GameObject prefabCasing;
        [SerializeField] GameObject prefabProjectile;
        [SerializeField] public RuntimeAnimatorController controller;
        [SerializeField]private Sprite spriteBody;
        [SerializeField]private AudioClip audioClipHolster;
        [SerializeField]private AudioClip audioClipUnholster;
        [SerializeField]private AudioClip audioClipReload;
        [SerializeField]private AudioClip audioClipReloadEmpty;
        [SerializeField]private AudioClip audioClipFireEmpty;

        private Animator animator;
        private WeaponAttachmentManagerBehaviour attachmentManager;
        private int ammunitionCurrent;
        private MagazineBehaviour magazineBehaviour;
        private MuzzleBehaviour muzzleBehaviour;
        private IGameModeService gameModeService;
        private CharacterBehaviour characterBehaviour;
        private Transform playerCamera;

        protected override void Awake()
        {
            animator = GetComponent<Animator>();
            attachmentManager = GetComponent<WeaponAttachmentManagerBehaviour>();
            gameModeService = ServiceLocator.Current.Get<IGameModeService>();
            characterBehaviour = gameModeService.GetPlayerCharacter();
            playerCamera = characterBehaviour.GetCameraWorld().transform;
        }
        protected override void Start()
        {
            magazineBehaviour = attachmentManager.GetEquippedMagazine();
            muzzleBehaviour = attachmentManager.GetEquippedMuzzle();
            ammunitionCurrent = magazineBehaviour.GetAmmunitionTotal();
        }

        public override Animator GetAnimator() => animator;

        public override Sprite GetSpriteBody() => spriteBody;

        public override AudioClip GetAudioClipHolster() => audioClipHolster;
        public override AudioClip GetAudioClipUnholster() => audioClipUnholster;

        public override AudioClip GetAudioClipReload() => audioClipReload;
        public override AudioClip GetAudioClipReloadEmpty() => audioClipReloadEmpty;

        public override AudioClip GetAudioClipFireEmpty() => audioClipFireEmpty;

        public override AudioClip GetAudioClipFire() =>
muzzleBehaviour.GetAudioClipFire();
```

```
        public override int GetAmmunitionCurrent() => ammunitionCurrent;

        public override int GetAmmunitionTotal() =>
magazineBehaviour.GetAmmunitionTotal();

        public override bool IsAutomatic() => automatic;
        public override float GetRateOfFire() => roundsPerMinutes;

        public override bool IsFull() => ammunitionCurrent ==
magazineBehaviour.GetAmmunitionTotal();
        public override bool HasAmmunition() => ammunitionCurrent > 0;

     public override RuntimeAnimatorController GetAnimatorController() => controller;
        public override WeaponAttachmentManagerBehaviour GetAttachmentManager() =>
attachmentManager;

        public override void Reload()
        {animator.Play(HasAmmunition() ? "Reload" : "Reload Empty", 0, 0.0f);}
        public override void Fire(float spreadMultiplier = 1.0f)
        {
            if (muzzleBehaviour == null)return;
            if (playerCamera == null)return;
            Transform muzzleSocket = muzzleBehaviour.GetSocket();
            const string stateName = "Fire";
            animator.Play(stateName, 0, 0.0f);
            ammunitionCurrent = Mathf.Clamp(ammunitionCurrent - 1, 0,
magazineBehaviour.GetAmmunitionTotal());
            muzzleBehaviour.Effect();
            Quaternion rotation = Quaternion.LookRotation(playerCamera.forward *
1000.0f - muzzleSocket.position);

            if (Physics.Raycast(new Ray(playerCamera.position, playerCamera.forward),
                out RaycastHit hit, maximumDistance, mask))
                rotation = Quaternion.LookRotation(hit.point - muzzleSocket.position);

            GameObject projectile = Instantiate(prefabProjectile,
muzzleSocket.position, rotation);
            projectile.GetComponent<Rigidbody>().linearVelocity =
projectile.transform.forward * projectileImpulse;
        }

        public override void FillAmmunition(int amount)
        {
            ammunitionCurrent = amount != 0 ? Mathf.Clamp(ammunitionCurrent + amount,
                0, GetAmmunitionTotal()) : magazineBehaviour.GetAmmunitionTotal();
        }

        public override void EjectCasing()
        {
            if(prefabCasing != null && socketEjection != null)
         Instantiate(prefabCasing, socketEjection.position, socketEjection.rotation);
        }
    }
}
```

Projectile

```
using System;
using UnityEngine;
using System.Collections;
using Random = UnityEngine.Random;

public class Projectile : MonoBehaviour {
```

```
	[Range(5, 100)]
	public float destroyAfter;
	public bool destroyOnImpact = false;
	public float minDestroyTime;
	public float maxDestroyTime;

	public Transform [] bloodImpactPrefabs;metalImpactPrefabs;dirtImpactPrefabs;
	public Transform []concreteImpactPrefabs;

	private void Start ()
	{
		var gameModeService = ServiceLocator.Current.Get<IGameModeService>();
	Physics.IgnoreCollision(gameModeService.GetPlayerCharacter().GetComponent<Collid
er>(), GetComponent<Collider>());
		StartCoroutine (DestroyAfter ());
	}

	private void OnCollisionEnter (Collision collision)
	{
		if (collision.gameObject.GetComponent<Projectile>() != null)return;
		if (!destroyOnImpact) {StartCoroutine (DestroyTimer ());}
		else {Destroy (gameObject);}

		if (collision.transform.tag == "Concrete")
		{
			Instantiate (concreteImpactPrefabs [Random.Range
				(0, bloodImpactPrefabs.Length)], transform.position,
				Quaternion.LookRotation (collision.contacts [0].normal));
			Destroy(gameObject);
		}
EnemyHealth enemyHealth = collision.gameObject.GetComponent<EnemyHealth>();
		if (enemyHealth != null)
		{
		Vector3 hitDirection = transform.forward;
		float baseDamage = 25.0f;
float finalDamage = baseDamage*(1.0f+UpgradeManager.Instance.bonusDamagePercentage);

		bool isCrit = false;
		if (Random.value <= UpgradeManager.Instance.critChance)
		{
		    finalDamage *= UpgradeManager.Instance.critDamageMultiplier;
		    isCrit = true;
		}
		enemyHealth.TakeDamage(finalDamage, hitDirection, isCrit);
		Destroy(gameObject);
		 }
	}

	private IEnumerator DestroyTimer ()
	{
		yield return new WaitForSeconds
			(Random.Range(minDestroyTime, maxDestroyTime));
		Destroy(gameObject);
	}

	private IEnumerator DestroyAfter ()
	{
		yield return new WaitForSeconds (destroyAfter);
		Destroy (gameObject);
	}
}
```