МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних інформаційних систем

_____Юрій КОНДРАТЕНКО

«____»_____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

РОЗРОБКА ARPG-ГРИ НА РУШІЇ UNITY

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

Здобувач _____ Сергій НАЗАРЧУК

«___»_____ 2025 р.

Керівник старший викладач кафедри ІПЗ _____ Марина ФАЛЕНКОВА

«___»_____ 2025 р

Миколаїв – 2025

Чорноморський національний університет імені Петра Могили
(повне найменування закладу вищої освіти)

Факультет	Комп’ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	122 Комп’ютерні науки
Освітня програма	Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Юрій КОНДРАТЕНКО

«____» _____ 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача

Назарчука Сергія Сергійовича
((прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Розробка ARPG-гри на рушії Unity».

Керівник роботи: Фаленкова Марина Володомирівна, старший викладач кафедри ІПЗ.

Затверджена наказом ЧНУ ім. Петра Могили від «18» листопада 2024 р. № 321.

2. Строк представлення кваліфікаційної роботи «___» _____ 2025 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні: Результатом роботи є повноцінний прототип ARPG-гри, реалізований на ігровому рушії Unity, який включає основні ігрові механіки: управління персонажем, бойову систему, систему взаємодії з ігровим середовищем, квестова система, а також базові елементи інтерфейсу користувача. Початковими даними для розробки слугуватимуть графічні

ресурси, технічне завдання з описом функціоналу, а також документація Unity і мови програмування C#.

4. Перелік питань, що підлягають розробці: Аналіз особливостей жанру ARPG та основних механік ігрового процесу; Вибір і налаштування ігрового рушія Unity для розробки 2D ARPG-гри; Розробка системи управління персонажем та бойової системи; Створення інтерактивного ігрового середовища та логіки взаємодії з об’єктами та NPC; Реалізація базового інтерфейсу користувача (UI) та візуальних ефектів.

5. Перелік графічних матеріалів: презентація.

Керівник роботи _____ Марина ФАЛЕНКОВА

(Особистий підпис) *(Власне ім’я ПРІЗВИЩЕ)*

Здобувач _____ Сергій НАЗАРЧУК

(Особистий підпис) *(Власне ім’я ПРІЗВИЩЕ)*

Дата видачі завдання «18» грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

кваліфікаційної роботи

Тема: Розробка ARPG-гри на рушії Unity

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	18.12.2024	24.12.2024	
2	Аналіз предметної області та постановка задачі	25.12.2024	27.01.2025	
3	Огляд джерел за темою кваліфікаційної роботи, зокрема огляд існуючих проєктів та релізних ігор в цьому жанрі та аналогічних проєків, схожих по жанру до моєї гри. Занотовування ідей	31.01.2025	01.03.2025	
4	Створення макету гри	02.03.2025	01.04.2025	
5	Робробка гри	02.04.2025	15.05.2025	
6	Перший попередній захист КР на засіданні комісії кафедри	16.05.2025	16.05.2025	
7	Корегування роботи за результатами попереднього захисту	29.05.2025	29.05.2025	
8	Другий попередній захист КР на засіданні комісії кафедри	30.05.2025	30.05.2025	
9	Доробка та остаточне оформлення КР	31.05.2025	10.05.2025	
10	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	11.05.2025	17.05.2025	

Керівник роботи _____ Марина ФАЛЕНКОВА

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Назарчук Сергій

(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану

«__» _____ 2025 р.

АНОТАЦІЯ

до кваліфікаційної роботи
здобувача групи 401 ЧНУ ім. Петра Могили

Назарчук Сергій Сергійович

на тему: **“ РОЗРОБКА ARPG-ГРИ НА РУШІЇ UNITY ”**

Актуальність даного дослідження полягає у зростаючому попиті на якісний інтерактивний ігровий контент, зокрема в жанрі action role-playing game (ARPG), а також у необхідності опанування сучасних засобів розробки цифрових ігор, таких як Unity та мова програмування C#. Створення подібних проєктів дозволяє не лише реалізувати креативні ідеї, а й набути практичних навичок, що є актуальними на ринку праці в галузі геймдеву.

Об’єктом процес розробки відеоігор, що охоплює створення ігрових механік, графіки, та інтерактивної логіки.

Предметом роботи є реалізація основних механік ARPG-гри за допомогою рушія Unity.

Метою створення повноцінного ігрового прототипу у жанрі ARPG, який включає бойову систему, управління персонажем, інтерфейс користувача та взаємодію з ігровим середовищем.

У результаті виконання роботи було реалізовано ігровий проєкт, що складається з декількох локацій, інтегровано базові механіки жанру, створено інтерфейс користувача (UI), а також розроблено візуальні ефекти й систему бою.

Робота складається з чотирьох розділів. У першому розділі проведено аналіз жанру ARPG. У другому – наведено інструменти розробки ігор, структуру ігрового проєкту. У третьому – розглянуто етапи реалізації ігрової логіки, опис основних систем гри, бойових механік і UI. У четвертому – проведено тестування, аналіз результатів і запропоновано напрями подальшого розвитку гри.

Загальний обсяг роботи – 94 сторінки. Кваліфікаційна робота містить 1 додаток, 30 рисунки, 2 таблиці і 30 джерел посилання.

Ключові слова: ARPG, Unity, розробка ігор, ігрові механіки, C#, бойова система, Tilemap, UI.

ABSTRACT

to the qualification work by the student of the group 401 of Petro Mohyla Black Sea National University

Nazarchuk Sergiy

“ARPG GAME DEVELOPMENT ON UNITY ENGINE”

A relevance of this study lies in the growing demand for high-quality interactive game content, particularly in the action role-playing game (ARPG) genre, as well as the need to master modern digital game development tools, such as Unity and the C# programming language. Creating such projects allows not only to implement creative ideas, but also to acquire practical skills that are relevant in the game development labor market.

An object of the work is the process of developing video games.

A subject of the work is the implementation of the main mechanics of the ARPG game using the Unity engine.

A purpose of the work is to create a full-fledged game prototype in the ARPG genre, which includes a combat system, character control, user interface and interaction with the game environment.

The work consists of four sections. The first section analyzes the ARPG genre. The second provides game development tools and the structure of the game project. The third considers the stages of implementing game logic, describes the main game systems, combat mechanics and UI. The fourth tests, analyzes the results and suggests directions for further development of the game.

The total volume of the work is 94 pages. The qualification work contains 1 appendice, 30 figures, 2 tables and 30 reference sources

Key words: ARPG, Unity, game development, game mechanics, C#, combat system, Tilemap, UI.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 3
ВСТУП 4
1 АНАЛІЗ СУЧАСНОГО СТАНУ РОЗРОБКИ КОМП'ЮТЕРНИХ ІГОР 5
1.1 Актуальність розробки гри 5
1.2 Огляд та аналіз ринку ігор в жанрі action-RPG 7
1.3 Постановка задачі 13
Висновки до першого розділу 15
2 АНАЛІЗ ІНСТРУМЕНТІВ РОЗРОБКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ 17
2.1 Технології та методи для вирішення поставленої задачі 17
2.2 Аналіз інструментів розробки 18
Висновки до другого розділу 27
3 РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ 29
3.1 Опис вхідних даних та структури системи 29
3.2 Проєктування/моделювання 32
3.3 Аналіз отриманих результатів розробки гри 37
Висновки до третього розділу 45
4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА СХЕМИ 48
4.1 Опис програмної реалізації, схеми та діаграми 48
4.2 Керівництво користувача 57
Висновки до четвертого розділу 59
ВИСНОВКИ 60
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 61
ДОДАТОК КОД ПРОГРАМНОЇ РЕАЛІЗАЦІЇ 64

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ЛКМ – Ліва Кнопка Миші

ШІ – Штучний Інтелект

RPG – Role Play Game

ARPG – Action Role Play Game

UI – User Interface

NPC – Non-Player Character

ОС – Операційна система

HUD – Heads-Up Display

AI – Artificial Intelligence

ВСТУП

Метою кваліфікаційної роботи є закріплення набутих під час навчання теоретичних знань, а також отримання професійних знань в процесі розробки гри в жанрі action-RPG на рушії Unity, для подальшого їх застосування у професійній діяльності.

Для досягнення визначеної мети необхідно вирішити такі завдання:

– ознайомлення з основними етапами роботи в розробці відео-ігор;

– оволодіння методиками збору та аналізу інформації;

– пошук і вивчення матеріалів за обраною темою;

– самостійне виконання та розробка фрагментів кваліфікаційної бакалаврської роботи;

– застосування вивчених методів під час розв’язання практичних задач.

В ході кваліфікаційної роботи було реалізовано ряд практичних завдань, що стосуються концепції та розробки гри в жанрі action-RPG. Головною метою було закріпити теоретичні знання, отримані під час навчання, та інтегрувати їх на практиці шляхом виконання власного проекту.

Кваліфікаційна робота надала можливість детально зануритися в усі аспекти розробки гри – від первинної ідеї до створення основних робочих механізмів. Значну увагу було приділено забезпеченню цілісності ігрового процесу та розробці інтуїтивно зрозумілого інтерфейсу.

Актуальність даної теми полягає в стрімкому розвитку індустрії відеоігор, що, відповідно, збільшує попит на високоякісний ігровий контент. Крім того, реалізація подібного проекту в освітніх умовах сприяє розвитку креативного мислення, аналітичних навичок та здатності працювати з передовими технологіями.

1 АНАЛІЗ СУЧАСНОГО СТАНУ РОЗРОБКИ КОМП'ЮТЕРНИХ ІГОР

1.1 Актуальність розробки гри

Ігрова індустрія на сьогодні є одним з найдинамічніших і найперспективніших секторів цифрової економіки, що демонструє стабільне зростання та постійне оновлення як у технологічному, так і в творчому аспектах. Щороку світовий ринок поповнюється сотнями, а то й тисячами нових ігрових проєктів, які охоплюють широкий спектр жанрів, платформ, стилів візуального оформлення, а також підходів до побудови геймплею. У цьому багатоманітті особливе місце посідає жанр action-RPG (екшн-рольова гра), який заслужено користується високою популярністю серед гравців різного віку та вподобань.

Жанр action-RPG вирізняється поєднанням двох ключових складових: динамічного бойового процесу в реальному часі (характерного для екшн-ігор) та системи розвитку персонажа, сюжетних розгалужень і кастомізації характеристик (що є основою RPG – рольових ігор). Така комбінація дозволяє гравцеві не лише брати активну участь у бойових сутичках, а й приймати рішення, що впливають на розвиток сюжету, створювати власну стратегію прокачування героя та занурюватися у світ гри значно глибше.

Однією з головних переваг action-RPG є саме інтерактивність та свобода дій, що відкриває широкі можливості для залучення користувача. Гравець керує персонажем у режимі реального часу, обирає стиль бою, поступово покращує здібності героя, здобуває нові навички, взаємодіє з іншими персонажами, досліджує навколишній світ та виконує квести. Саме завдяки цим елементам жанр зберігає свою актуальність і привабливість для мільйонів користувачів по всьому світу.

У сучасних умовах розвитку інформаційних технологій та програмного забезпечення створення ігор такого типу стало доступним не лише великим студіям,

а й індивідуальним розробникам. Особливу роль у цьому відіграє ігровий рушій Unity, який є одним з найпоширеніших і найпотужніших інструментів для кросплатформенної розробки ігор. Unity надає розробникам цілий спектр можливостей: підтримку тривимірної та двовимірної графіки, інтегровану систему фізики, анімацій, звукового оформлення, взаємодії з інтерфейсом користувача (UI), систему скриптів на мові C# та багато іншого. Завдяки гнучкості та відкритості Unity дозволяє створювати проєкти будь-якої складності — від простих прототипів до комерційних ігор.

Процес розробки гри в жанрі action-RPG є надзвичайно багатогранним. Він вимагає поєднання технічних знань, дизайнерського мислення, логічного проєктування і ретельного тестування. Це не лише програмування логіки гравця чи ворогів, а й продумана структура проєкту, баланс ігрових механік, зручна навігація інтерфейсу, оптимізація ресурсів та забезпечення стабільної роботи гри на різних пристроях. Важливими елементами також є побудова сюжетних ліній, реалізація системи завдань (квестів), інвентаризації та інтерактивної взаємодії між об’єктами гри.

У рамках цієї кваліфікаційної роботи було обрано саме жанр action-RPG як найбільш відповідний для демонстрації комплексу навичок, що охоплюють як технічні аспекти (програмування, робота з рушієм), так і творчі (дизайн, побудова ігрової логіки). Розробка гри дозволяє повноцінно продемонструвати вміння працювати з різними модулями Unity, створювати інтерактивні об’єкти, реалізовувати систему керування персонажем, взаємодію з навколишнім середовищем та інтерфейсом користувача.

Предметною областю дипломного проєкту є створення повноцінного ігрового прототипу в жанрі action-RPG, який включає в себе базові механіки пересування персонажа, бойову систему, інвентар, систему виконання завдань (квестів), елементи інтерфейсу (HUD, меню), а також стилізацію та анімацію. Такий підхід дозволяє на

практиці охопити повний цикл створення гри: від постановки ідеї — до реалізації функціонального прототипу, що працює у реальному середовищі розробки.

1.2 Огляд та аналіз ринку ігор в жанрі action-RPG

Перш ніж безпосередньо переходити до етапу розробки власної гри в жанрі action-RPG, доцільно було здійснити всебічний аналіз вже існуючих проєктів, які репрезентують цей напрямок у сучасній ігровій індустрії. Такий підхід є не лише корисним з точки зору вивчення технічних рішень, а й необхідним для формування глибшого розуміння загальних тенденцій ринку, очікувань цільової аудиторії та особливостей реалізації ключових ігрових механік.

Аналіз аналогів дозволяє виявити сильні сторони успішних проєктів, що здобули визнання серед гравців, а також вказати на слабкості або недоліки, які слід уникати у власній розробці. Крім того, він дає змогу оцінити ступінь реалізації бойової системи, системи прокачки, інтерфейсу користувача, візуального стилю, балансу складності тощо. Завдяки цьому формується більш цілісне бачення майбутньої гри, що поєднує як найкращі практики, запозичені з інших проєктів, так і власні інноваційні рішення.

Одним із цікавих прикладів є Cat Quest II — яскравий представник легких action-RPG із мультяшним візуальним стилем і доступним геймплеєм. Гра орієнтована на широку аудиторію і пропонує захопливий світ, наповнений гумором, квестами та динамічними боями в реальному часі. Гравець може перемикатися між двома персонажами (котом і псом), що відкриває можливості для кооперативної гри. Геймплей побудований навколо дослідження відкритого світу, поступової прокачки, збирання екіпірування та проходження сюжетних і побічних завдань. Простота управління та приємна атмосфера роблять гру чудовим прикладом балансування між доступністю та задоволенням від процесу.

Рисунок 1.1 – Гра «Cat Quest II»

Інший приклад — Unsouled, піксельна 2D action-RPG з високою динамікою боїв. Головна ставка тут зроблена на складну бойову систему, яка вимагає від гравця реакції, точності та освоєння комбо-прийомів. Гра не має акценту на сюжет, однак вирізняється візуальними ефектами та камерою, яка реагує на події в реальному часі.

Рисунок 1.2 – Гра «Unsouled»

Unsouled чудово демонструє, як за допомогою мінімалістичної графіки можна створити напружену та насичену діями атмосферу. Водночас висока складність гри не завжди сприяє ширшій популярності, але приваблює гравців, які цінують бойову глибину.

Rogue Heroes: Ruins of Tasos — ще один приклад інді-гри з елементами action-

RPG і roguelike. Гра поєднує кооперативний геймплей, генерацію підземель, будівництво власного міста та поступову прокачку персонажів. Гравці досліджують локації, змагаються з ворогами, вирішують головоломки, що робить гру схожою на класичні Zelda-подібні проєкти.

Рисунок 1.3 – Гра «Rogue Heroes: Ruins of Tasos»

Візуально гра витримана в стилі піксель-арт, що відповідає духу класичних аркад і водночас дозволяє розробникам зосередитися на вдосконаленні механік, а не лише графіки. Це також робить гру доступною з точки зору технічних вимог, розширюючи її потенційну аудиторію.

Rogue Heroes вирізняється колективним аспектом гри, дозволяючи до чотирьох гравців одночасно брати участь у проходженні. Її геймплей менш зосереджений на індивідуальному розвитку персонажа, проте надає широкі можливості в рамках командної взаємодії та дослідження світу.

Ще одним важливим прикладом є Chronicon — класична 2D action-RPG з видом зверху, яка більше орієнтована на "олдскульний" стиль гри у дусі Diablo. Проєкт пропонує глибоку систему прокачки, величезну кількість предметів, заклинань, дерев

навичок і класів. Гра створена одним розробником, але має вражаючий обсяг контенту.

Рисунок 1.4 – Гра «Chronicon»

Chronicon вирізняється великою тривалістю, системою луту та реіграбельністю, що досягається через процедурну генерацію та різноманіття ворогів. При цьому графіка лишається досить простою, і основна ставка робиться саме на механіки.

Таблиця 1.1 – Порівняльна характеристика action-RPG ігор

Назва	Основні функції	Переваги	Недоліки
Cat Quest II	Простий і приємний візуальний стиль	Доступний ігровий процес Дуже зрозуміла механіка	Легка складність (майже без виклику) Мала кількість геймплейних новинок
Unsouled	Динамічна бойова система, стилізована графіка, без зайвих деталей	Дуже гнучкий контроль бою. Гарна анімація. Відчуття динаміки	Висока складність для новачків. Іноді втрачається фокус у боях.

Кінець таблиці 1.1

Rogue Heroes: Ruins of Tasos	Підземелля з пастками, прокачка селища + героя	Атмосфера ретро-Zelda Кастомізація класу	Повторюваність підземель, AI ворогів не завжди адекватний
Chronicon	Величезна кількість предметів і скілів Рандомізовані підземелля	Глибока RPG-система Варіативність персонажів	Стара графіка Інтерфейс місцями перевантажений

У сучасній науковій літературі, присвяченій розробці ігор, значна увага приділяється вивченню та аналізу принципів проєктування ігрових механік, а також моделюванню поведінки користувача в інтерактивному середовищі. Особливий інтерес дослідників викликають ті аспекти, що впливають на ступінь залучення гравця до процесу, забезпечують належний рівень інтуїтивності керування, а також формують глибину ігрового досвіду.

Наприклад, у фундаментальній праці "Rules of Play: Game Design Fundamentals" авторів Кеті Сален та Еріка Циммермана висвітлюється концепція триєдиного підходу до дизайну ігор — механіки, динаміки та естетики (так званий MDA-фреймворк). Цей підхід передбачає взаємопов'язане проєктування систем правил (механіки), поведінкових патернів гравців (динаміки) та емоційного відгуку (естетики), що разом формують повноцінний та захопливий геймплей [1].

Крім того, в наукових публікаціях, пов’язаних з галуззю ігрової інженерії, часто порушуються питання технічної реалізації ключових компонентів гри, зокрема складних систем штучного інтелекту, які керують поведінкою NPC, алгоритмів бойової логіки та методів динамічного балансування складності.

Наприклад, у праці Millington, I., Funge, J. — "Artificial Intelligence for Games" (2nd ed., 2009) розглядаються різні підходи до створення адаптивної поведінки

персонажів, включаючи finite state machines, behavior trees, та планування на основі GOAP (Goal-Oriented Action Planning), що є надзвичайно актуальними для action-RPG, де важливо створити ілюзію «розумного» супротивника.

Ще одним важливим джерелом є публікація "Dynamic Difficulty Adjustment in Games" (Hunicke, 2005), в якій авторка пропонує механізми адаптивного балансування складності, що автоматично змінює виклики у грі в залежності від рівня майстерності або поточних дій гравця. Такий підхід дозволяє підтримувати стабільний інтерес і уникати фрустрації, що є важливим аспектом при реалізації бойових механік.

Також варто згадати роботу Yannakakis, G. N., Togelius, J. — "Artificial Intelligence and Games" (2018), яка описує сучасні підходи до генеративного дизайну рівнів, поведінкової аналітики та еволюційних алгоритмів у контексті комп’ютерних ігор. Автори приділяють значну увагу емоційній адаптації ігор та персоналізації геймплею, що відкриває можливості для створення більш глибокого занурення у світ гри.

Усе це особливо актуально для жанру action-RPG, де гравець очікує гнучкої адаптації рівня виклику, інтелектуальної поведінки супротивників, а також логічного й послідовного реагування світу гри на дії користувача. Використання сучасних алгоритмів штучного інтелекту дозволяє створити більш живий, динамічний світ, що змінюється разом із гравцем та його рішеннями.

Крім того, в наукових публікаціях, пов’язаних з галуззю ігрової інженерії, часто порушуються питання технічної реалізації ключових компонентів гри, зокрема складних систем штучного інтелекту, які керують поведінкою NPC, алгоритмів бойової логіки та методів динамічного балансування складності. Усе це особливо актуально для жанру action-RPG, де гравець очікує гнучкої адаптації рівня виклику, інтелектуальної поведінки супротивників, а також логічного й послідовного реагування світу гри на дії користувача.

Таким чином, науковий підхід до проєктування ігор не лише слугує

теоретичним підґрунтям, але й має прикладне значення у створенні якісних ігрових продуктів. Врахування таких досліджень при розробці власного проєкту дозволяє глибше осмислити особливості жанру, зосередитись на створенні узгоджених ігрових систем і забезпечити високий рівень задоволеності кінцевого користувача.

Таким чином, науковий підхід до проєктування ігор не лише слугує теоретичним підґрунтям, але й має прикладне значення у створенні якісних ігрових продуктів. Врахування таких досліджень при розробці власного проєкту дозволяє глибше осмислити особливості жанру, зосередитись на створенні узгоджених ігрових систем і забезпечити високий рівень задоволеності кінцевого користувача.

Проаналізувавши аналогічні проєкти, можна зробити висновок, що основними складовими гри в цьому жанрі є:

- захоплююча бойова система;
- варіативність у стилях гри;
- система прокачки персонажа;
- виразний візуальний стиль;
- зрозумілий інтерфейс.

Ці спостереження стали основою для формування концепції власного проєкту, який має поєднати в собі як традиційні, так і оригінальні рішення, орієнтовані на зручність та задоволення користувача.

1.3 Постановка задачі

На основі проведеного аналізу ринку, вивчення актуальних тенденцій та порівняння аналогів в жанрі action-RPG, було зроблено висновок, що розробка власного ігрового продукту в цьому напрямі повинна ґрунтуватися на поєднанні як традиційних, так і інноваційних механік. Метою дослідження є створення сучасного, технічно стабільного та з точки зору користувача привабливого прототипу гри, який

реалізує базові характеристики жанру та дозволяє гравцеві зануритися у віртуальний світ з високим рівнем інтерактивності.

Одна з ключових задач полягає у формуванні збалансованої та в той же час динамічної ігрової механіки, яка включає в себе як бойову систему, так і елементи прокачування персонажа, дослідження ігрового середовища, взаємодію з об’єктами, NPC та іншими компонентами світу. Створення такої системи потребує ретельного опрацювання внутрішньої логіки гри, врахування очікувань цільової аудиторії, а також дотримання принципів оптимальної складності, що не перевантажує користувача, але й водночас викликає постійний інтерес до прогресу.

Мета цієї кваліфікаційної роботи полягає у практичному втіленні концепції гри в жанрі action-RPG, в якій гравцеві буде надано можливість повноцінного управління ігровим персонажем у режимі реального часу. Гравець зможе не лише брати участь у бойових сутичках, але й виконувати сюжетні та побічні завдання, збирати ресурси, покращувати характеристики свого героя, а також взаємодіяти з різними елементами світу гри.

Проєктна реалізація передбачає розробку гри із просторими ігровими локаціями або навіть базовими елементами відкритого світу, які забезпечують свободу пересування та дослідження. Окрема увага приділяється механіці розвитку персонажа, що повинна мотивувати гравця до тривалого проходження гри завдяки системі винагород, накопичення досвіду, відкриття нових здібностей і використання інвентаря.

Для досягнення поставленої мети було визначено наступні завдання, які потрібно вирішити в процесі розробки:

– розробити концепцію гри: що включає в себе не лише загальний стиль, жанрову специфіку та візуальні рішення, а й ключові механіки, які будуть визначати геймплей;

– створити базове ігрове середовище з використанням рушія Unity — вибраного за його гнучкість, підтримку 2D/3D графіки та широкі можливості для інтеграції анімації, звуків та UI;

– реалізувати управління персонажем, включаючи його переміщення, стрибки, атаки та інші дії, які є базовими для інтерактивності;

– розробити просту систему противників, які будуть володіти базовим штучним інтелектом, атакувати гравця та взаємодіяти з ним у бою;

– забезпечити взаємодію з об’єктами середовища, включно з відкриттям скринь, збором предметів, діалогами з NPC, активацією тригерів тощо;

– продумати базовий інтерфейс користувача (UI): інвентар, здоров’я, меню;

– провести тестування, щоб переконатися у відсутності критичних помилок, забезпечити плавність ігрового процесу, оптимізацію продуктивності, а також належний користувацький досвід.

Таким чином, основною метою розробки є створення прототипу гри в жанрі action-RPG, який не лише демонструє технічні можливості розробника, але й служить платформою для подальшого розширення та вдосконалення проєкту. Проєкт повинен відповідати вимогам сучасного геймдизайну, бути функціональним, стабільним та мати потенціал для розвитку як комерційного, так і навчального продукту.

Висновки до першого розділу

У першому розділі було здійснено детальний аналіз сучасного стану розробки комп’ютерних ігор, зокрема у жанрі action-RPG, а також досліджено ключові особливості та тенденції ігрової індустрії, що роблять цей жанр актуальним та популярним. Було підкреслено значення поєднання динамічного бойового процесу з рольовими елементами, що забезпечує гравцям глибокий та інтерактивний ігровий досвід.

Проведено огляд і порівняння відомих проєктів у жанрі action-RPG, таких як Cat

Quest II, Unsouled, Rogue Heroes та Chronicon, що дозволило виявити їх сильні сторони, а також потенційні недоліки. Цей аналіз дав змогу окреслити основні компоненти, які мають бути враховані при розробці власного ігрового проєкту, зокрема: якісна бойова система, система прокачки персонажа, різноманітність стилів гри, інтуїтивний інтерфейс та адаптивність ігрових механік.

Також було розглянуто наукові підходи до проектування ігрових систем, зокрема концепцію MDA-фреймворку (механіки, динаміки, естетики), а також методи штучного інтелекту, які забезпечують гнучкість поведінки NPC та адаптацію складності, що є критично важливим для жанру action-RPG.

Отже, на основі проведеного аналізу зроблено висновок про необхідність комплексного підходу до розробки ігрового проєкту, який би інтегрував у собі як технічні, так і дизайнерські аспекти, забезпечуючи збалансований і захоплюючий геймплей. Це стане основою для подальшої розробки ігрового прототипу на рушії Unity, який поєднуватиме ключові елементи жанру action-RPG та відповідатиме сучасним стандартам ігрової індустрії.

2 АНАЛІЗ ІНСТРУМЕНТІВ РОЗРОБКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

2.1 Технології та методи для вирішення поставленої задачі

Для реалізації проєкту розробки гри в жанрі action-RPG було обрано сучасні інструменти та підходи, які дозволяють ефективно створювати 2D ігри з використанням рушія Unity. Вибір технологій базувався на таких критеріях, як функціональність, підтримка спільноти, зручність інтеграції та відповідність вимогам проєкту.

Основним рушієм для розробки став Unity, який забезпечує підтримку як 2D, так і 3D графіки, має зручний візуальний редактор сцен, розвинуту систему компонентів та потужну інфраструктуру для обробки фізики, анімації, користувацького інтерфейсу (UI) та роботи зі звуком. Перевагою Unity є також можливість кросплатформенної збірки, що дозволяє адаптувати гру під різні операційні системи та пристрої без значних змін у коді.

Для написання логіки гри було використано мову програмування C#, яка є основною для Unity. Цей вибір дозволяє реалізувати об'єктно-орієнтовану архітектуру проєкту, структурувати код у вигляді незалежних класів і скриптів, а також забезпечити легке тестування та масштабування ігрових систем.

Серед ключових технічних рішень, які використовувались у процесі розробки:

– система керування персонажем — реалізована за допомогою обробки вхідних подій з клавіатури, фізики RigidBody2D та використання анімаційного контролера Animator;

– бойова система — передбачає ближній бій, використання зброї, а також обробку зіткнень з противниками за допомогою колайдерів та подій типу OnTriggerEnter2D;

– штучний інтелект ворогів — побудований на базі finite state machine (скінченних автоматів), які дозволяють реалізувати різні стани поведінки (патрулювання, атака, переслідування, очікування);

– система квестів та діалогів — реалізована через власні скрипти, що обробляють завдання, їх прогрес та змінюють поведінку NPC після виконання умови;

– інвентар та система предметів — базується на використанні списків (List<T>), структури даних ScriptableObject для зберігання інформації про предмети, а також взаємодії з інтерфейсом UI;

– ігровий інтерфейс (HUD) — створений за допомогою Unity UI Toolkit, який дозволяє динамічно оновлювати дані (здоров’я, кількість ресурсів тощо) та забезпечує зручну взаємодію з користувачем.

Для візуального оформлення були використані спрайти у форматі PNG, адаптовані для 2D-середовища, а для анімації застосовувався вбудований аніматор Unity з підтримкою спрайт-сетів (Sprite Sheets).

Застосування згаданих технологій та методів дозволяє комплексно вирішити поставлену задачу створення ігрового прототипу, який включає основні механіки жанру action-RPG, зберігаючи при цьому баланс між якістю, продуктивністю та зручністю реалізації.

2.2 Аналіз інструментів розробки

Для розробки проекту було вирішено використати ігровий двигун Unity, який належить до числа найпопулярніших платформ для створення ігор, як двовимірних, так і тривимірних. Значна поширеність двигуна зумовлена численними перевагами, серед яких варто відзначити кросплатформеність, що забезпечує просте експортування проекту для різних ОС та пристроїв. Додатково, Unity може

похвалитися гнучкою архітектурою, що полегшує організацію коду та логіки гри, а також інтегрує широкий спектр інструментів для фізики, анімації, інтерфейсів (UI), візуальних ефектів та інших елементів ігрового процесу. Ще однією ключовою перевагою є активна спільнота розробників і великий обсяг офіційних і сторонніх матеріалів, що значно полегшує процес розробки, особливо для початківців або при вирішенні специфічних завдань.

У межах реалізації 2D action-RPG гри були використані такі технології та інструменти:

– Unity 2022+ — рушій, у якому реалізовано всі основні компоненти гри: ігрова логіка, взаємодія з користувачем, система анімацій та бойова механіка;

– Visual Studio 2022 — як основне середовище для написання та налагодження коду;

– C# — Головна мова програмування в Unity. Її застосовують для написання скриптів, що відповідають за пересування персонажа, обробку подій, логіку бою,управління інвентарем та інші системи;

– Tilemap Editor — інструмент Unity для створення ігрових карт на основі тайлів, що особливо зручно для 2D проєктів;

– Rigidbody — для фізичної симуляції об’єктів;

– Animator і Animation — компоненти, що дозволяють створювати плавні переходи між станами персонажа (наприклад, ходьба, атака, отримання ушкоджень, смерть);

– Cinemachine — використовується для налаштування поведінки камери, наприклад, її слідування за гравцем;

– TextMesh Pro — для якісного виводу тексту (наприклад, діалоги, HUD, повідомлення);

– Sprite Atlas — оптимізує роботу з великою кількістю спрайтів, зменшуючи використання пам’яті та прискорюючи рендеринг.

Також у процесі розробки були застосовані основи патернів програмування (наприклад, State Machine для управління поведінкою персонажа).

2.2.1 Visual Studio 2022

Visual Studio 2022 — це найоптимальніше інтегроване середовище для розробки, що дозволяє створювати функціональні, привабливі багатоплатформні додатки для Windows, Mac, Linux, iOS та Android. Розробляйте повноцінні клієнтські застосунки, використовуючи такі технології, як WinForms, WPF, WinUI, MAUI чи Xamarin. Кожна з них має конструктори Visual Studio, котрі дають змогу керувати додатком та переглядати його, використовуючи різні інструменти, що полегшують створення комплексних макетів [2].

До того ж, Visual Studio активно застосовується для розробки програмного забезпечення, зокрема ігор на Unity. Вона підтримує мову програмування C#, котра є ключовою для створення ігрової логіки у проектах Unity, зокрема і для 2D, і для 3D ігор.

Основні можливості Visual Studio 2022:

– розширене автодоповнення коду (IntelliSense), яке допомагає уникати помилок і пришвидшує розробку;

– інтегрована система дебагінгу для пошуку та виправлення логічних і синтаксичних помилок;

– зручна робота з файлами та структурами проєктів;

– вбудована підтримка Git — для контролю версій, створення комітів, гілок та злиття змін;

– підтримка розширень, що дозволяють додавати нові можливості до середовища;

– глибока інтеграція з Unity, що дозволяє відкривати й редагувати скрипти безпосередньо з редактора сцени.

Visual Studio 2022 працює на Windows і дозволяє працювати з великими проєктами без втрати продуктивності. Середовище надає всі необхідні засоби для ефективної роботи з кодом, створення обробників подій, керування анімацією, фізикою, об’єктами сцени та іншими ігровими компонентами.

Рисунок 2.1 – Логотип Visual Studio [23]

Таким чином, Visual Studio 2022 виступає як ключовий інструмент у розробці 2D-гри, забезпечуючи стабільність і зручність у програмуванні, тестуванні та налагодженні ігрових механік

2.2.2 Unity

Unity — це сьогочасна ітерація одного з найбільш поширених ігрових рушіїв, яким послуговуються як новачки, так і досвідчені команди розробників. Unity здатний

оперувати як з двовимірною, так і з тривимірною графікою, але саме для розробки 2D-проєктів, як той, що представлений в цій роботі, він пропонує комфортний арсенал інструментів та складових [3].

Серед ключових плюсів Unity — його інтуїтивний інтерфейс, що дозволяє оперативно взаємодіяти зі сценами, об’єктами, анімаціями та фізичними явищами. Unity 2022 може похвалитися поліпшеною продуктивністю, вдосконаленим вікном редактора і оновленими компонентами UI Toolkit, що спрощує розробку користувацького інтерфейсу.

Рисунок 2.2 – Логотип Unity [22]

Основні особливості Unity 2022:

– підтримка двовимірної фізики (2D Rigidbody, Collider, Physics Materials),

– інструменти анімації, які дозволяють створювати плавні рухи об’єктів і персонажів;

– вбудований менеджер сцен і таймлайн, що спрощує контроль за подіями в грі;

– інтеграція з Visual Studio — дозволяє швидко відкривати скрипти та тестувати логіку;

– Asset Store, де можна знайти готові ресурси: персонажі, шейдери, UI-елементи, аудіо тощо;

– можливість збірки гри для різних платформ — Windows, Android, WebGL тощо [3].

У процесі роботи з Unity 2022 було створено ігрові сцени, додано ігрові об’єкти, налаштовано колізії та обробку взаємодій між персонажем і середовищем. Unity дозволяє швидко протестувати зміни в реальному часі, що суттєво пришвидшує цикл розробки

2.2.3 Tilemap Editor

При створенні двовимірних сцен надзвичайно важливим є Tilemap Editor, вбудований у Unity. Він надає інструменти для формування рівнів шляхом розміщення тайлів на заданій сітці. Це значно економить час при створенні ігрових світів, особливо коли сцена насичена повторюваними елементами, такими як платформи, шляхи чи стіни.

Ключова перевага Tilemap полягає у можливості розробника працювати з рівнем як з мозаїкою, використовуючи готові зображення-тайли. Для зручності ці тайли зберігаються у Tile Palette, звідки їх легко переміщувати на сцену. Завдяки цьому процес редагування рівня нагадує малювання у графічному редакторі, що робить роботу інтуїтивно зрозумілою [5].

Tilemap підтримує ряд корисних функцій:

– роботу з багатошаровими картами (фон, об'єкти, передній план);

– просте налаштування колізій для тайлів;

– підтримку автоматичних переходів між тайлами;

– збереження Tilemap як компонента, який можна редагувати незалежно від решти сцени [5].

В рамках проєкту Tilemap Editor застосовувався для розробки ключової ігрової локації, включно з навколишнім середовищем, перепонами та ландшафтними складовими. Це забезпечило значну економію часу та надало карті адаптивності до змін навіть на пізніх етапах процесу розробки.

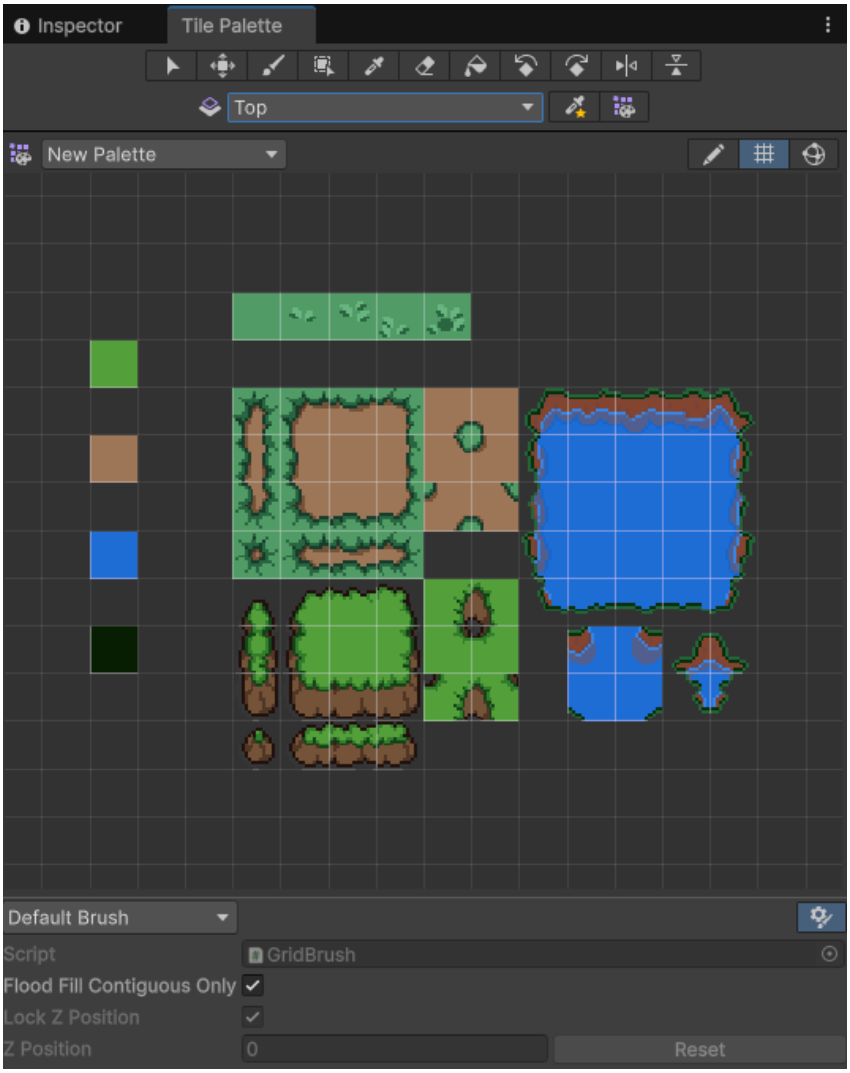

Рисунок 2.3 – вікно Tilemap Editor вбудований в Unity

Завдяки Tilemap, реалізація візуальної частини рівнів відбулася швидко та без ускладнень, що є важливим фактором для невеликих 2D-проєктів.

2.2.4 Animator і Animation

Для реалізації анімацій в ігровому проєкті було використано систему Animator у поєднанні з інструментом Animation, які вбудовані в Unity. Ці компоненти дозволяють оживити персонажів та об’єкти гри, надаючи їм більш реалістичну поведінку.

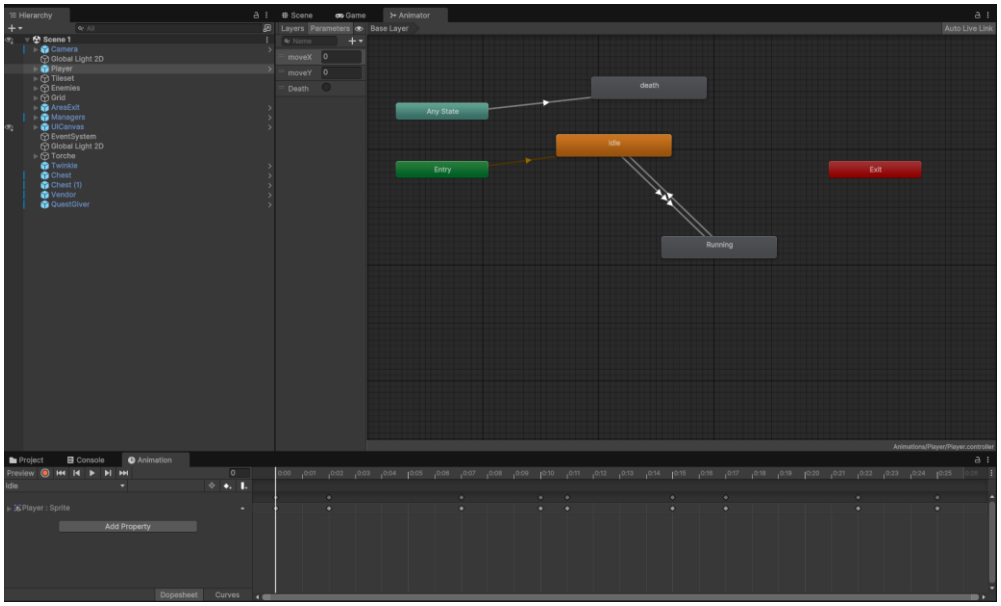

Рисунок 2.4 – вікно Animator і Animation з логікою анімації персонажу

Інструмент Animation слугує для створення окремих анімаційних фрагментів, наприклад, ходьби, стрибка, атаки чи отримання ушкоджень. Анімації робили як вручну в редакторі, так і на основі спрайт-листів, де кожний кадр задає конкретне положення об'єкта у часі. Це дає можливість досягти реалістичного руху персонажів без потреби у громіздких 3D-моделях.

Animator, у свою чергу, керує логікою перемикання між анімаціями. Його система станів (state machine) дає змогу визначити, за яких обставин має програватися та чи інша анімація. Приміром, коли гравець натискає клавішу руху – запускається стан бігу, а під час атаки – відповідна анімація нападу. Перехід між станами можна

налаштовувати за допомогою умов, параметрів (тригерів, булевих значень, чисел), що робить поведінку персонажа динамічною та здатною адаптуватися [6].

Перевага такої системи – її гнучкість: можна формувати складні взаємозв'язки між анімаціями, не прописуючи їх вручну в коді. В межах цього проєкту Animator використовувався для управління станами головного героя та противників, що значно поліпшило візуальну складову гри.

Завдяки поєднанню Animation і Animator вийшло створити плавні переходи між різними діями персонажів, що позитивно впливає на загальне сприйняття гри.

2.2.5 Cinemachine та TextMesh Pro

У проєкті було використано два потужних інструменти Unity — Cinemachine та TextMesh Pro, які значно покращують візуальну частину гри та якість взаємодії з користувачем.

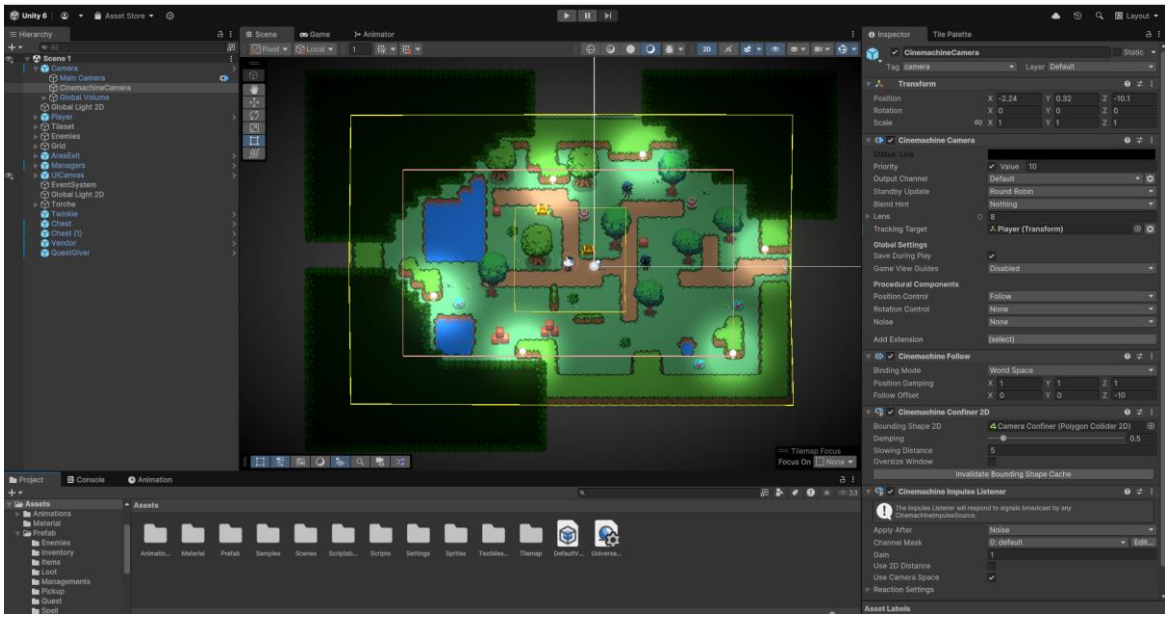

Рисунок 2.5 – приклад як виглядає Cinemachine камера в проєкті

Cinemachine — це система віртуальних камер, яка дає змогу реалізувати гнучке та плавне стеження за об’єктами у грі. Вона автоматично адаптує положення та рух камери залежно від положення гравця або інших важливих подій. Наприклад, під час пересування головного героя, камера не просто фіксовано слідує за ним, а додає ефекти затримки та згладження, завдяки чому гравець не відчуває різких ривків. Це особливо актуально для 2D action-RPG, де важливо утримувати гравця в центрі уваги без втрати динаміки. Також було використано конфігурацію Confiner, що дозволяє обмежити рух камери межами рівня [7].

Другий компонент — TextMesh Pro — застосовувався для виводу текстової інформації, зокрема діалогів, інтерфейсних повідомлень, лічильників здоров’я, очок тощо. TextMesh Pro надає розширені можливості стилізації тексту, включаючи підтримку тіней, контурів, кольорових градієнтів, шрифтів високої чіткості. Це дозволило зробити текстову інформацію у грі не тільки функціональною, а й естетично привабливою. Крім того, компонент добре оптимізований для Unity та підтримує локалізацію, що буде корисно при подальшому розширенні гри на інші мови [8].

Завдяки інтеграції Cinemachine та TextMesh Pro вдалося суттєво покращити користувацький досвід, зробивши гру більш плавною, динамічною та візуально приємною.

Висновки до другого розділу

У другому розділі було проведено комплексний аналіз інструментів та технологій, які застосовувались під час виконання завдань переддипломної практики. Вибір рушія Unity як основної платформи для розробки 2D action-RPG гри був зумовлений його широкими функціональними можливостями, кросплатформеністю, підтримкою анімацій, фізики та гнучкою архітектурою. Завдяки інтеграції з Visual Studio 2022 та використанню мови програмування C#, було досягнуто високого рівня

керованості логікою гри, а також забезпечено легкість налагодження та масштабування проєкту.

Застосування таких інструментів, як Tilemap Editor, Animator, TextMesh Pro та Unity UI Toolkit, дозволило ефективно реалізувати ключові механіки гри: керування персонажем, бойову систему, інтерфейс, інвентар та штучний інтелект ворогів. Особливу роль відіграли об’єктно-орієнтовані принципи програмування та використання патернів, таких як скінченні автомати, що сприяло модульності та читабельності коду.

Комбінація сучасного інструментарію, методів та підходів надала змогу створити гнучкий і функціональний ігровий прототип, що відповідає базовим вимогам жанру action-RPG, а також заклала основу для подальшого розвитку та вдосконалення гри.

3 РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

3.1 Опис вхідних даних та структури системи

Розробка гри передбачає наявність чітко визначеної структури даних та організації ігрових систем, які взаємодіють між собою у реальному часі. Для повноцінної роботи проєкту були визначені ключові типи вхідних даних, їх формат, а також логічна структура системи, що забезпечує належну обробку цих даних.

Вхідні дані

Основними вхідними даними у контексті гри виступають:

– користувацьке введення – сигнали з клавіатури, які керують рухами персонажа, активацією здібностей, атаками та взаємодією з об'єктами довкілля. Для цього використовується система Input System Manager;

– ігрові конфігурації – набори параметрів, які містять інформацію про властивості зброї. Це дозволяє централізовано керувати характеристиками без жорсткого закодування значень у скриптах. Інші об’єкти скриптовані, але піддаються легкому редагуванню;

– ресурси користувацького інтерфейсу (UI) – спрайти, текстові елементи, іконки предметів, що динамічно підвантажуються в HUD під час гри;

– картографічні дані – тайлові карти рівнів, створені за допомогою Tilemap Editor та додано до Tile Palette. Вони визначають структуру середовища, включно з платформами, стінами, зонами зіткнень, спавнами ворогів тощо;

– анімаційні дані – набори спрайтів та анімаційні кліпи, що використовуються системою Animator для створення рухів, атак, ушкоджень та інших станів персонажів.

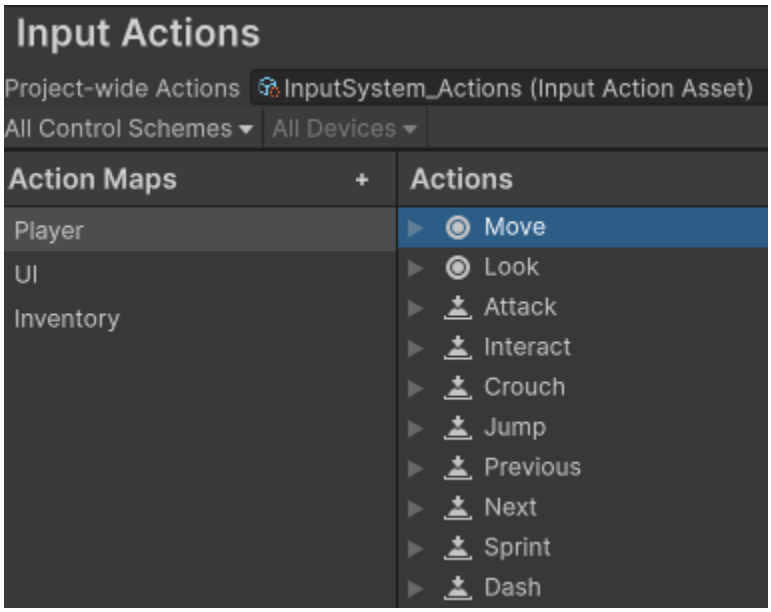

Рисунок 3.1 – Система Input System Manager

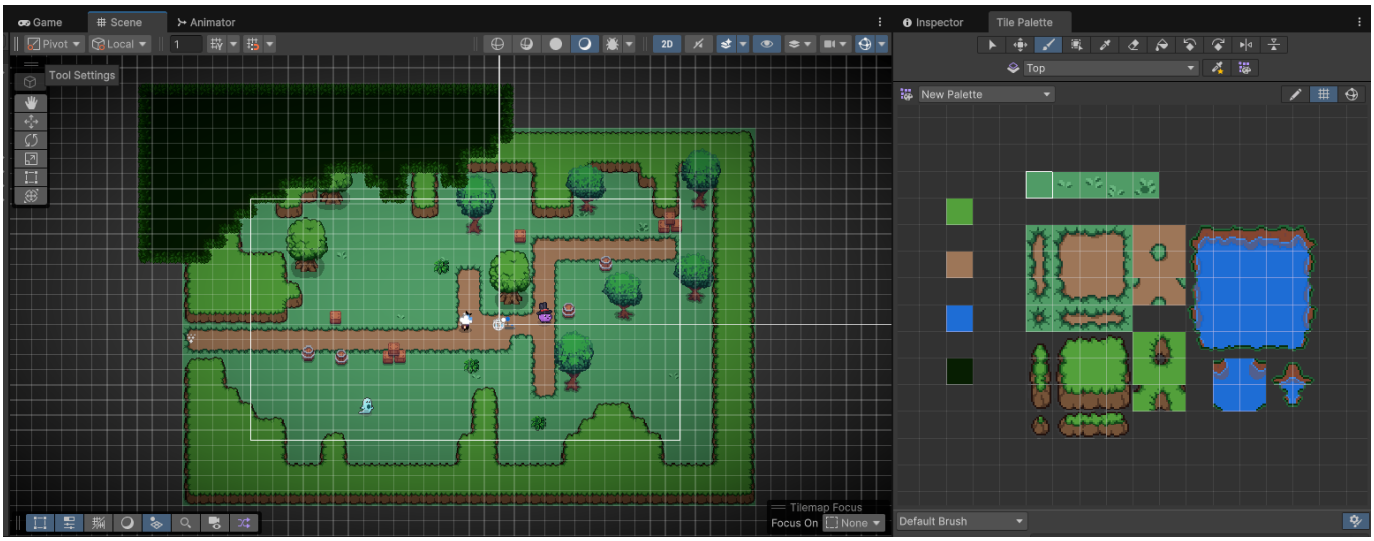

Рисунок 3.2 – Створена локація за допомогою Tile Palette

Структура системи. Структура гри побудована за об'єктно-орієнтованим підходом і складається з декількох модулів:

– модуль управління гравцем (Player Controller) – відповідає за обробку введення, фізичну взаємодію (через Rigidbody2D), запуск анімацій та відображення станів (здоров’я, мана, тощо);

– модуль штучного інтелекту (AI System) – включає в себе логіку поведінки ворогів, реалізовану через finite state machine. Кожен стан (патрулювання, атака, переслідування) має відповідні скрипти, які активуються на основі умов;

– модуль бою (Combat System) – обробляє атаки гравця і ворогів, зіткнення, розрахунок шкоди, а також реакції на отримання ушкоджень, включаючи ефекти та анімації;

– інвентар і предмети (Inventory System) – дозволяє зберігати, переміщувати та використовувати ігрові предмети;

– система квестів та діалогів – відповідає за запуск діалогів із NPC, зміну поведінки персонажів після виконання завдань, а також оновлення статусу квестів;

– UI-система (HUD) – відображає дані про гравця, повідомлення, інвентар, квести тощо. Реалізована на основі Unity UI Toolkit та TextMeshPro;

– система управління сценами (Scene Manager) – відповідає за завантаження та перемикання сцен, збереження прогресу та ініціалізацію об'єктів на рівні.

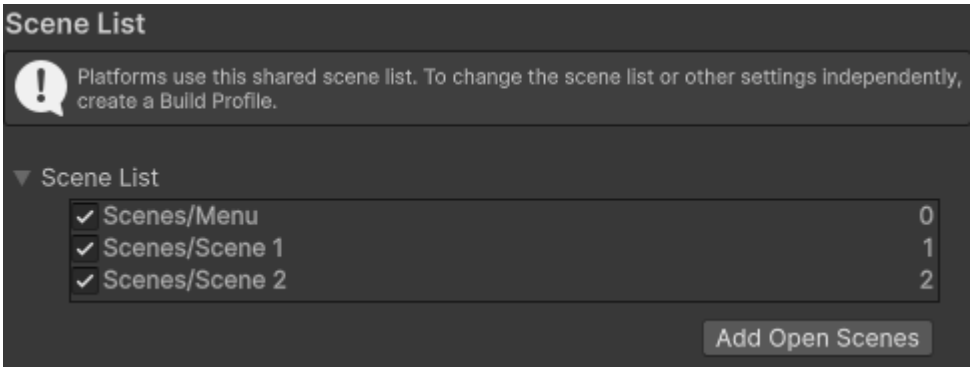

Рисунок 3.3 – Сцени із гри

Усі компоненти гри взаємодіють через публічні методи, події або делегати, що забезпечує модульність і легкість у підтримці та розширенні функціоналу.

3.2 Проєктування/моделювання

Етап проєктування є ключовим у процесі розробки програмного продукту, оскільки саме на ньому визначається структура системи, основні компоненти, їх взаємодія, а також загальний підхід до реалізації логіки гри. У процесі проєктування використовувались принципи модульності, повторного використання коду, а також розділення відповідальностей між компонентами.

Загальна структура.

Процес розробки ігрової системи базувався на принципах об’єктно-орієнтованого програмування, що забезпечує високий рівень модульності, масштабованості та повторного використання коду. Основна архітектура гри була побудована з урахуванням чіткої структуризації коду, що дозволяє розділити проєкт на незалежні логічні компоненти, кожен з яких відповідає за певний аспект функціонування гри.

У рамках реалізації програмної логіки було виокремлено ключові ігрові сутності, серед яких: персонаж гравця, ворожі юніти (супротивники), взаємодіючі об’єкти ігрового світу, елементи середовища, система квестів, інвентаризаційна система, бойова механіка, система прокачування та розвитку навичок, а також інтерфейс користувача (UI). Кожна з перелічених сутностей була реалізована у вигляді окремого класу або групи класів із чітко визначеними функціональними обов’язками та межами відповідальності.

Завдяки такому підходу стало можливим ефективно управляти взаємодією між компонентами, а також забезпечити легку модифікацію і подальший розвиток окремих модулів без необхідності втручання у загальну структуру проєкту. Наприклад, зміни в бойовій системі або алгоритмах штучного інтелекту

супротивників не потребують значного переписування інших частин гри, що є ключовою перевагою об’єктно-орієнтованої архітектури.

Особливу увагу було приділено уніфікації підходів до обробки подій та керування станами ігрових об’єктів. Було впроваджено механізми делегування, подієвого програмування та стан-машин, що дозволяє зручно контролювати поведінку ігрових елементів залежно від поточних умов або дій гравця.

Загалом, загальна структура гри була розроблена з урахуванням не лише технічних вимог, але й принципів зручності супроводу та майбутнього масштабування проєкту. Такий підхід не тільки полегшив етап реалізації основного функціоналу, але й забезпечив необхідну гнучкість для інтеграції нових можливостей, розширення контенту, а також покращення продуктивності й оптимізації гри загалом.

Система спроєктована таким чином, щоб підтримувати масштабування та гнучке розширення. Наприклад, можливо без значних змін додавати нові типи ворогів, нові предмети чи квести завдяки використанню шаблонів і абстрактних класів.

Вибір архітектури.

Кожен ігровий об'єкт складається з набору скриптів-компонентів, які реалізують конкретну функціональність — рух, атака, виявлення гравця, взаємодія з об'єктами тощо.

Умовно можна виділити наступні групи компонентів:

- компоненти керування – обробка вводу, переміщення, керування камерою;
- ігрова логіка – бойова система, система досвіду, квести, інвентар;
- візуальні елементи – UI, анімації, ефекти;
- допоміжні системи – керування, обробка подій.

Структура даних.

Дані про наприклад про зброю, зберігаються у вигляді окремих ресурсів, які легко редагуються через інтерфейс Unity. Для цього використовувались

ScriptableObject, що дозволяє створювати конфігураційні файли безпосередньо в редакторі.

Рисунок 3.4 – ScriptableObject які реалізовані в проєкті

У цілому, структура проєкту була реалізована з орієнтацією на простоту, зручність у користуванні та логічну впорядкованість усіх елементів. Усі ресурси, необхідні для функціонування гри, зберігаються в основній директорії Assets, яка є стандартною та базовою для проєктів, створених у середовищі Unity. Саме в цій папці містяться всі ключові компоненти: скрипти, сцени, префаби, графічні ресурси, анімації, матеріали, та інші елементи, необхідні для повноцінного функціонування гри.

Незважаючи на відсутність глибоко вкладеної ієрархії чи складної структури директорій, така організація забезпечує достатній рівень зручності, особливо на початкових етапах розробки, коли важливо зберігати гнучкість і мати швидкий доступ до всіх основних елементів. Простота структури дозволяє ефективно орієнтуватися в проєкті навіть без попереднього ознайомлення з його деталями, що особливо важливо у випадках розширення функціоналу, рефакторингу чи командної співпраці

Надалі, за потреби масштабування гри чи збільшення обсягу контенту, така структура може бути легко адаптована — шляхом створення підкаталогів для окремих функціональних блоків, що дозволить зберегти впорядкованість без порушення цілісності проєкту.

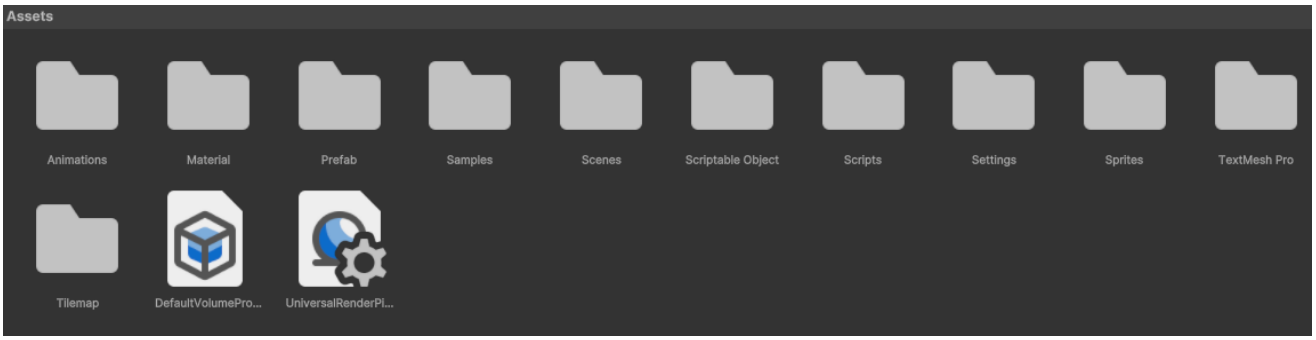

Рисунок 3.5 – Всі файли гри

Проєктування інтерфейсу.

Особлива увага в процесі розробки приділялася забезпеченню зручності користувача та створенню інтуїтивно зрозумілого інтерфейсу. Основна мета полягала в тому, щоб надати гравцю змогу взаємодіяти з ігровими системами максимально природно, без необхідності тривалого навчання чи пошуку функцій у складних меню. З урахуванням принципів юзабіліті та на основі аналізу подібних ігор жанру action-RPG, було розроблено адаптивний інтерфейс, який ефективно поєднує естетичну привабливість з функціональністю.

Рисунок 3.6 – Інтерфейс гри

Зокрема, було спроєктовано окремі інтерфейсні панелі для ключових підсистем гри: інвентарю, характеристик персонажа, системи діалогів, карти світу та системи квестів. Кожна з цих панелей виконує чітко визначену роль і організована у спосіб, що дозволяє швидко орієнтуватися в необхідній інформації. Наприклад, панель інвентарю дає змогу переглядати та керувати предметами, що належать гравцеві; панель характеристик відображає рівень здоров'я, досвіду, атаки, захисту тощо; система діалогів дозволяє вести спілкування з неігровими персонажами у зручному форматі, а карта забезпечує навігацію в ігровому світі.

Рисунок 3.7 – Елементи які будуть частиною інтерфейсу спорядження

Рисунок 3.8 – Елементи які будуть частиною інвентарю

Рисунок 3.9 – Елементи які будуть частиною інтерфейсу квестів

Всі елементи інтерфейсу були розташовані таким чином, щоб бути постійно доступними у потрібний момент. Було також реалізовано логіку відкриття та закриття панелей за допомогою гарячих клавіш, що забезпечує динамічну взаємодію з UI під час активної гри. Усе це сприяє комфортному користувацькому досвіду та підвищує загальну якість гри.

3.3 Аналіз отриманих результатів розробки гри

У результаті виконання поставлених завдань було реалізовано частковий, але функціонально наповнений прототип гри у жанрі action-RPG, що ілюструє основні ігрові механіки, характерні для цього напряму. Розроблений прототип дає змогу комплексно оцінити ефективність застосування теоретичних знань на практиці та демонструє можливість їх реалізації засобами сучасних інструментів розробки, зокрема Unity та мови програмування C#.

Рисунок 3.10 – Запуск гри, головне меню гри

Під час реалізації прототипу було створено базове ігрове середовище, яке включає стартову ігрову локацію, наповнену різними об’єктами, з якими може взаємодіяти гравець. Основна увага була зосереджена на впровадженні базової механіки руху персонажа, його взаємодії з середовищем, атаками, ухиленням та іншими базовими діями, без яких неможливе повноцінне функціонування жодної action-RPG гри. Управління реалізоване на основі стандартних схем керування, що забезпечують інтуїтивно зрозуміле користування та комфортну взаємодію з грою.

Рисунок 3.11 – Приклад бою з ворогами

Окрему увагу було приділено взаємодії гравця з ворогами та іншими NPC (неігровими персонажами). В грі реалізовано просту, але ефективну систему бою, де вороги реагують на присутність гравця, атакують його та завдають шкоди. Кожен ворог має власний запас здоров’я, а також передбачено динамічне зменшення цього запасу в результаті успішних атак гравця. Після знищення ворогів реалізована система випадіння здобичі, що додає грі елемент мотивації до дослідження й боїв.

Рисунок 3.12 – Здобич з ворогів

Важливою складовою проєкту стала реалізація системи квестів, що включає NPC, з якими можна вести діалоги, приймати завдання, виконувати їх і отримувати винагороду. Це забезпечує додаткову глибину геймплею та підвищує занурення гравця у віртуальний світ. Створено діалогову систему, яка дозволяє обирати варіанти відповідей, що може впливати на подальший перебіг взаємодії з персонажем. Реалізовано можливість отримання нового квесту після завершення попереднього, що дає змогу підтримувати ігрову активність.

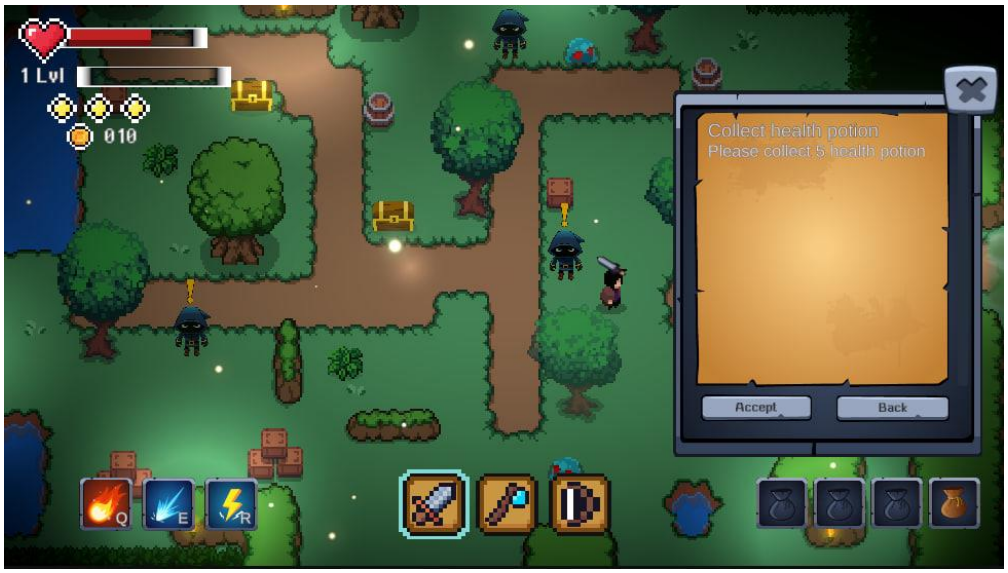

Рисунок 3.13 – Приклад обраного квесту

Рисунок 3.14 – Приклад повідомлення про виконання квесту

У рамках проєкту також було розроблено базовий інтерфейс користувача, що містить усі ключові елементи, необхідні для взаємодії з грою. Зокрема, було реалізовано індикатори рівня здоров’я персонажа, рівня розвитку, кількості накопичених монет, а також створено функціональний інвентар, що дозволяє зберігати предмети, здобуті під час гри. Крім того, розроблено меню спорядження, що

дозволяє змінювати озброєння та екіпірування, інтерфейс магазину для придбання та продажу предметів, головне меню.

Рисунок 3.15 – Система магазину за монети добуті з ворогів

Рисунок 3.16 – Вікно інвентарю та спорядження

Гравець має можливість досліджувати спеціально спроєктовані ігрові локації, створені для демонстрації ключових аспектів геймплею та функціональності систем.

Ці локації не лише візуально варіативні, а й інтерактивні — вони включають різноманітні об’єкти, з якими гравець може взаємодіяти. Наприклад, при знищенні або активації певних об'єктів можуть випадати ресурси, спорядження або унікальні квестові предмети, що необхідні для подальшого прогресу. Такий підхід стимулює гравця до уважного дослідження кожного куточка віртуального світу, формує відчуття винагороди за дослідницьку активність і суттєво підвищує інтерес до проходження.

Окрема увага була приділена реалізації системи випадіння здобичі з переможених супротивників. Цей механізм враховує різні параметри, тип противника та ймовірнісні випадіння предметів, у майбутньому буде враховано зокрема рівень ворога. Така система додає грі не лише елемент непередбачуваності, а й важливу складову реіграбельності, оскільки кожне зіткнення з ворогом може завершитися різними результатами. Це спонукає гравця до повторного проходження і створює додатковий мотиваційний чинник.

Загалом, дослідження локацій та здобич є важливою складовою загального геймплейного циклу, оскільки забезпечують постійну зміну контексту, підтримують динаміку подій та сприяють глибшому зануренню у гру. Такий підхід дозволяє підвищити рівень залученості гравця та робить проходження більш насиченим і варіативним.

Підібрано графічні елементи, стилістично узгоджені між собою та витримані в єдиній візуальній концепції, яка відповідає жанру фантастичного action-RPG. Візуальне оформлення проєкту базується на комбінації 2D-спрайтів з напівреалістичним ілюстративним стилем, що дозволяє передати атмосферу загадкового, дещо похмурого, але водночас захоплюючого ігрового світу. Кольорова палітра, тіні, деталізація об’єктів середовища та персонажів були обрані таким чином, щоб підсилити емоційне сприйняття подій, які розгортаються на екрані.

Для досягнення більшого ефекту занурення в ігровий світ, до сцени було додано

систему освітлення.

Також була інтегрована система кінематографічної камери Cinemachine — потужного інструменту Unity, що дозволяє створювати м’яке, плавне слідкування за гравцем, ефекти масштабування та зміщення в залежності від ситуацій у грі. Це не лише робить рух персонажа більш природним та приємним для сприйняття, а й дозволяє формувати епічність у сценах боїв або сюжетних подій. Камера автоматично адаптується до змін у просторі, зберігаючи композицію кадру та забезпечуючи гравцю комфортне спостереження за процесом гри.

Рисунок 3.17 – Скрині з речами, в яких можна знайти здобич та покласти її

Завдяки всім вищезгаданим елементам було створено гармонійну та естетично привабливу атмосферу, яка підсилює фантастичний наратив гри та забезпечує глибоке візуальне занурення у створений світ.

Система бою передбачає і ближній, і дальній бій. Ближній бій представлений ударом мечем, який фізично відкидає ворогів. Дальній бій реалізовано за допомогою

заклинань, чарівної палички, а також лука зі стрілами.

У грі також присутня механіка дешу (англ. Dash – ривок вперед) на клавішу пробілу, що дає змогу краще контролювати ситуацію на полі бою та уникати поранень від атак ворогів. Ця механіка використовує 1 з 3 зарядів дешу, які поступово відновлюються або випадають з переможених ворогів та об’єктів.

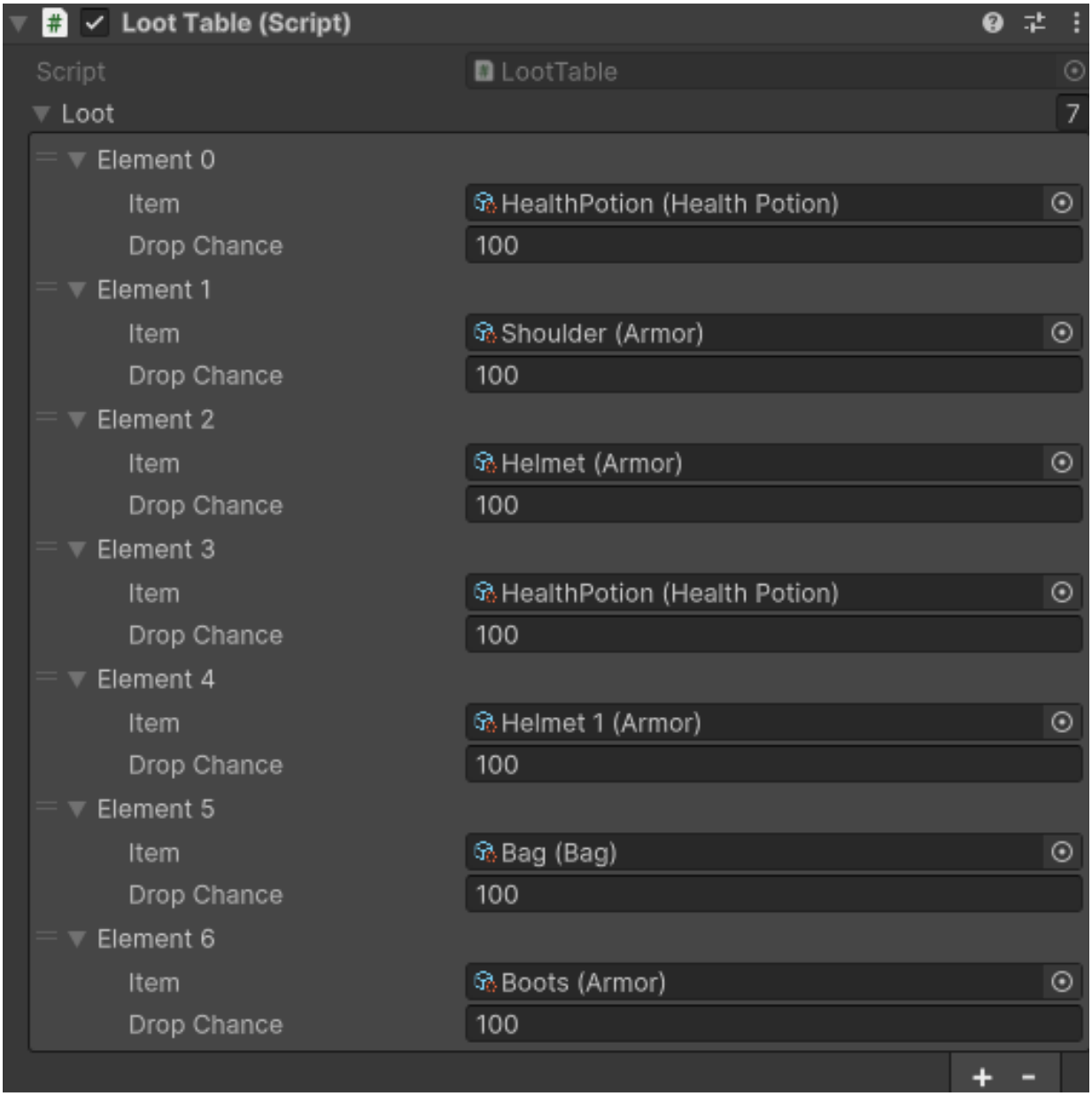

Рисунок 3.18 – Налаштування системи луту, який може випасти

Загалом, розроблений прототип, незважаючи на свій експериментальний статус, виконує важливу функцію перевірки життєздатності основних ігрових механік. Він дозволяє комплексно оцінити як технічну реалізацію закладених систем, так і їхню взаємодію між собою в умовах реального ігрового процесу. Прототип забезпечує базову іграбельність, включаючи рух персонажа, бойову систему, інвентар, елементи інтерфейсу, взаємодію з оточенням та базову логіку квестів, що вже дає змогу сформувати уявлення про майбутній геймплей.

Окрім цього, створений прототип є цінним етапом у реалізації повноцінного проєкту, адже дозволяє виявити можливі недоліки на ранній стадії, а також протестувати користувацький досвід, що в подальшому сприяє прийняттю обґрунтованих рішень щодо оптимізації або переробки окремих компонентів. Модульна структура, використана в процесі розробки, відкриває широкі можливості для масштабування проєкту — як у плані розширення функціоналу, так і в плані інтеграції нових механік або графічних елементів.

Таким чином, навіть у рамках обмеженої функціональності, реалізований прототип уже виконує роль фундаменту для подальшої розробки. Його створення дозволило закласти основи повноцінної ARPG-гри, яка у подальшому може бути перетворена на завершений комерційний або аматорський продукт із залученням розширеного контенту, складніших сценаріїв та покращеної графіки.

Висновки до третього розділу

У третьому розділі дипломної роботи було здійснено практичну реалізацію основних етапів створення інформаційної системи у вигляді частково реалізованої гри в жанрі action-RPG. У результаті розробки вдалося сформувати не лише технічну базу для побудови прототипу, а й на практиці перевірити дієвість обраних архітектурних рішень, структур даних, підходів до проєктування та реалізації ігрових механік, характерних для подібного роду ігор.

У рамках цього розділу було детально проаналізовано вхідні дані, що використовуються в ігровому середовищі, включно з користувацьким введенням, конфігураціями об'єктів, ресурсами інтерфейсу, картографічними елементами та анімаціями. На основі цих даних було побудовано модульну структуру системи, яка забезпечує взаємодію між компонентами гри через делегати, події та публічні методи. Такий підхід забезпечив високу гнучкість системи, її адаптивність до змін і потенційного розширення.

Особливу увагу було приділено процесу проєктування, де реалізовано чітке розділення відповідальностей між модулями: керування гравцем, бойова система, штучний інтелект ворогів, інвентар, квести, інтерфейс користувача та система сцен. Це дало змогу досягти високого рівня структурованості, модульності та масштабованості коду. Застосування об'єктно-орієнтованих принципів, зокрема інкапсуляції, спадкування та поліморфізму, стало важливим фактором для досягнення організованої та логічно завершеної системи.

У ході реалізації прототипу особливий акцент було зроблено на впровадженні основних геймплейних механік. Успішно реалізовано базову систему руху персонажа, атаки, ухилення, бойові взаємодії з супротивниками, а також просту, але функціональну систему квестів та діалогів із неігровими персонажами. Було впроваджено систему луту, яка забезпечує випадіння здобичі з ворогів або об'єктів, та дозволяє гравцю збирати корисні ресурси, предмети й спорядження.

Окремо варто відзначити реалізацію користувацького інтерфейсу, який охоплює всі необхідні елементи: від показників здоров’я, досвіду, монет до вікон інвентарю, спорядження, магазину, повідомлень та системи діалогів. Візуальна частина була організована з урахуванням принципів юзабіліті, що забезпечує зручну та інтуїтивно зрозумілу взаємодію гравця з грою.

Розроблене середовище доповнено графічними елементами в єдиному стилі, базовим освітленням, а також динамічною камерою з Cinemachine, що забезпечує

естетично привабливу візуалізацію. У гру було інтегровано як ближній, так і дальній типи бою, різні типи озброєння та магічні атаки, що додає різноманіття у геймплей та розширює тактичні можливості гравця.

Впровадження механіки ривка (дешу) стало додатковим ігровим елементом, який підвищує динаміку бою та глибину керування персонажем. Обмеження в кількості використань і можливість поповнення зарядів додають тактичної складової в управлінні ресурсами під час гри.

Таким чином, розроблений прототип, незважаючи на експериментальний статус, демонструє успішне поєднання теоретичних принципів проєктування програмного забезпечення з їх практичною реалізацією у вигляді інтерактивного, частково функціонального ігрового середовища. Він є важливою основою для подальшого розширення проєкту, дозволяє виявити сильні та слабкі сторони обраного підходу, а також окреслити шляхи оптимізації та вдосконалення майбутніх версій гри.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА СХЕМИ

4.1 Опис програмної реалізації, схеми та діаграми

Зараз наведу приклади ключових модулів скриптів, які реалізовані у проєкті. Вони охоплюють основні складові ігрового процесу, необхідні для ігрового процесу.

До них належать Система управління персонажем.

Приклад элементу коду PlayerControlls.cs:

```
public class PlayerController : Singleton<PlayerController>
{
    public bool FacingLeft { get { return facingLeft; } set { facingLeft = value; } }

    [SerializeField] private float moveSpeed = 1f;
    [SerializeField] private float dashSpeed = 4f;
    [SerializeField] private TrailRenderer myTrailRenderer;
    [SerializeField] private Transform weaponCollider;

    private InputSystem_Actions playerControls;
    private Vector2 movement;
    private Rigidbody2D rb;
    private Animator myAnimator;
    private Knockback knockback;
    private SpriteRenderer mySpriteRender;
    private float startingMoveSpeed;

    private bool facingLeft = false;
    private bool isDashing = false;
```

Цей фрагмент коду містить оголошення приватних змінних, які забезпечують функціональність управління персонажем у 2D-грі. Змінна myAnimator зберігає посилання на компонент Animator, який відповідає за запуск анімацій персонажа, таких як ходьба, атака, стрибок тощо. Компонент Knockback зберігається у змінній

knockback і, ймовірно, відповідає за реалізацію ефекту відштовхування персонажа при отриманні шкоди або взаємодії з ворогами. mySpriteRender містить компонент SpriteRenderer, який керує відображенням спрайта персонажа, зокрема — його віддзеркаленням (flip) залежно від напрямку руху.

Змінна startingMoveSpeed використовується для збереження початкової швидкості руху гравця, щоб її можна було повернути після закінчення дашу (ривка). Логічна змінна facingLeft вказує, чи дивиться персонаж ліворуч — це використовується для правильного відображення спрайта або орієнтації зброї. Змінна isDashing використовується для контролю, чи персонаж зараз виконує ривок (dash), щоб уникнути повторного виклику цієї дії, поки вона ще триває. У сукупності ці змінні утворюють базу для руху, анімацій, динаміки та візуальної взаємодії персонажа з оточенням.

Фрагмент коду з реалізацією механіки ривка (dash) персонажа у грі:

```
private void Dash()
{
     if (!isDashing && Stamina.Instance.CurrentStamina > 0)
    {
        Stamina.Instance.UseStamina();
        isDashing = true;
        moveSpeed *= dashSpeed;
        myTrailRenderer.emitting = true;
        StartCoroutine(EndDashRoutine());
    }
}
private IEnumerator EndDashRoutine()
{
    float dashTime = .2f;
    float dashCD = .25f;
    yield return new WaitForSeconds(dashTime);
    moveSpeed = startingMoveSpeed;
    myTrailRenderer.emitting = false;
    yield return new WaitForSeconds(dashCD);
    isDashing = false;
}
```

Метод Dash() викликається, коли гравець натискає відповідну клавішу для ривка. Спочатку перевіряється, чи персонаж не виконує ривок (!isDashing) та чи має достатньо витривалості (Stamina.Instance.CurrentStamina > 0). Якщо умови виконуються, витривалість зменшується методом UseStamina(), активується режим ривка (isDashing = true), збільшується швидкість руху персонажа (moveSpeed *= dashSpeed), а також активується візуальний ефект — слід за персонажем (myTrailRenderer.emitting = true). Далі запускається корутина EndDashRoutine().

У EndDashRoutine() реалізовано часову логіку: персонаж ривком рухається протягом dashTime (0.2 секунди), після чого його швидкість повертається до початкової (moveSpeed = startingMoveSpeed) і вимикається слід. Потім ще протягом dashCD (0.25 секунди) персонаж не може робити новий ривок — це свого роду затримка між ривками (cooldown). Після цього isDashing встановлюється у false, і гравець може знову використати ривок. Така реалізація забезпечує баланс у геймплеї та дозволяє створити динамічні бойові сцени.

Бойова система включає механіку нанесення шкоди, виявлення колізій під час ударів, таймінг атак, а також виведення візуального відображення шкоди. Додатково реалізовано відштовхування ворогів під час удару, а також параметри різних видів зброї (швидкість, шкода, радіус дії).

Приклад коду логіки завдання шкоди ворогам під час атаки:

```
private int damageAmount;

private void Start()
{
    MonoBehaviour currentActiveWeapon =
ActiveWeapon.Instance.CurrentActiveWeapon;
    damageAmount = (currentActiveWeapon as
IWeapon).GetWeaponInfo().weaponDamage;
}

private void OnTriggerEnter2D(Collider2D other)
```

```
{
        EnemyHealth enemyHealth =
other.gameObject.GetComponent<EnemyHealth>();
        enemyHealth?.TakeDamage(damageAmount);
}
```

У методі Start() відбувається ініціалізація змінної damageAmount, яка зберігає кількість шкоди, що буде завдана ворогу. Для цього отримується поточна активна зброя гравця через ActiveWeapon.Instance.CurrentActiveWeapon, яка реалізує інтерфейс IWeapon. Через цей інтерфейс отримується структура WeaponInfo, з якої витягується значення weaponDamage.

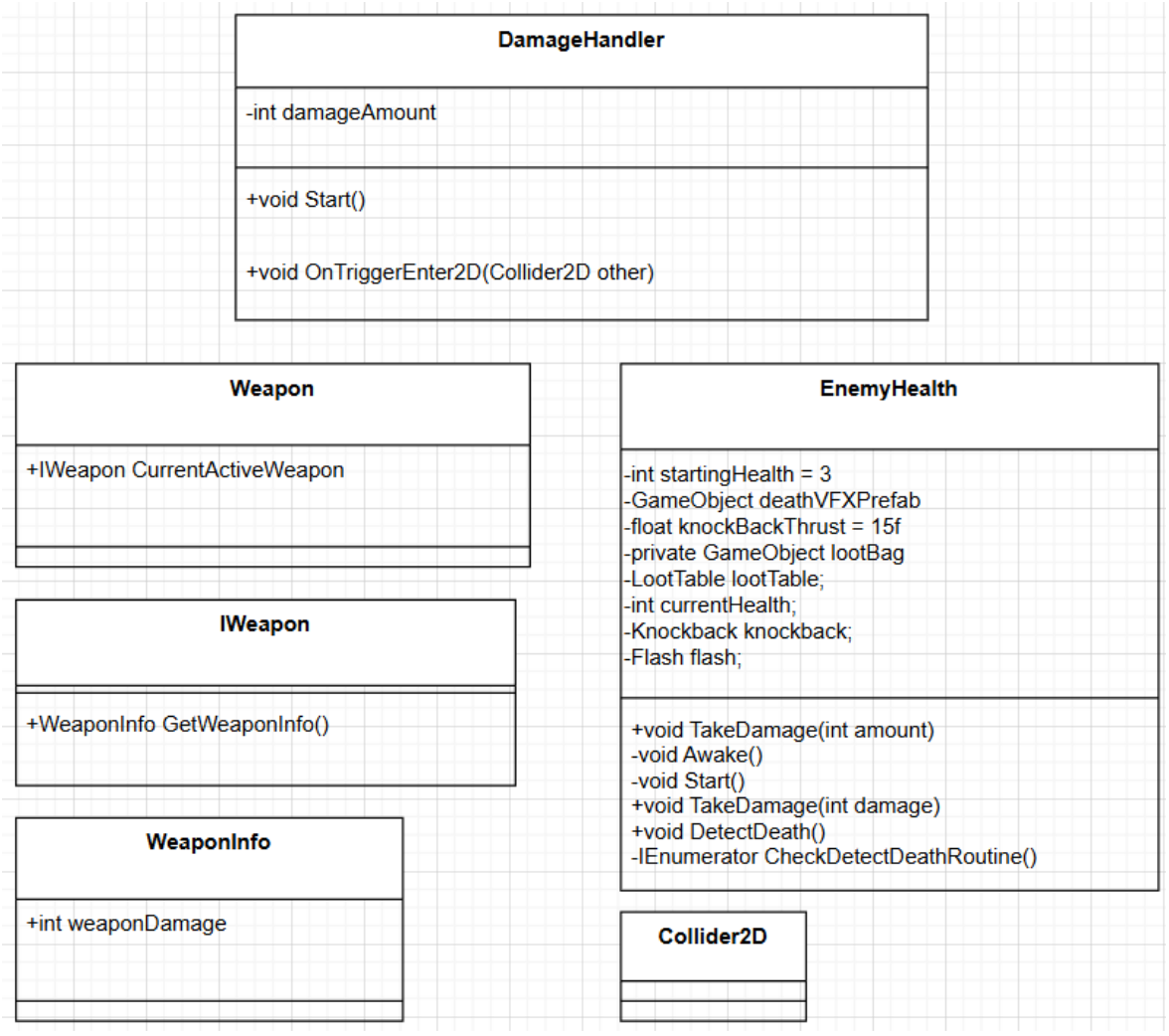

Рисунок 4.1 – Скрипт з логікою завдання шкоди ворогам

Метод OnTriggerEnter2D(Collider2D other) автоматично викликається, коли об'єкт, до якого прикріплений цей скрипт, стикається з іншим об'єктом у 2D-просторі. В середині методу перевіряється, чи об'єкт, з яким відбулось зіткнення, має компонент EnemyHealth. Якщо так — викликається метод TakeDamage(damageAmount), який зменшує здоров’я ворога на визначену кількість. Такий підхід дозволяє легко керувати шкодою від різних видів зброї, зберігаючи при цьому гнучкість і масштабованість коду.

Система штучного інтелекту (AI) ворогів реалізована у вигляді самостійного модуля, який керує поведінкою ворожих істот залежно від ігрової ситуації. В основі логіки AI лежить машина станів (Finite State Machine), яка перемикається між двома основними станами: патрулювання (Roaming) та атака (Attacking).

Приклад елементу коду EnemyAI.cs:

```
private void Roaming()
{
    timeRoaming += Time.deltaTime;

    enemyPathFinding.MoveTo(roamPosition);

    if (Vector2.Distance(transform.position,
PlayerController.Instance.transform.position) < attackRange)
    {
        state = State.Attacking;
    }

    if (timeRoaming > roamChangeDirFloat)
    {
        roamPosition = GetRoamingPosition();
    }
}

private void Attacking()
{
```

```
    if (Vector2.Distance(transform.position,
PlayerController.Instance.transform.position) > attackRange)
    {
        state = State.Roaming;
    }

    if (attackRange != 0 && canAttack)
    {
        canAttack = false;
        (enemyType as IEnemy).Attack();

        if (stopMovingWhileAttacking)
        {
            enemyPathFinding.StopMoving();
        }
        else
        {
            enemyPathFinding.MoveTo(roamPosition);
        }

        StartCoroutine(AttackCooldownRoutine());
    }
}
```

Цей код реалізує штучний інтелект (ШІ) для ворога, який перемикається між двома станами: патрулювання (Roaming) та атака (Attacking), залежно від відстані до гравця та внутрішніх умов.

У методі Roaming() ворог переміщується до випадкової позиції roamPosition, яку обирає заздалегідь. Водночас накопичується час у змінній timeRoaming. Якщо ворог підходить ближче до гравця, ніж дозволяє attackRange, стан змінюється на Attacking. Якщо минуло більше часу, ніж вказано у roamChangeDirFloat, вибирається нова позиція для патрулювання. Це забезпечує динамічну зміну напрямку руху, що імітує "блукання".

Метод Attacking() спрацьовує, коли ворог знаходиться в бойовому стані. Якщо гравець віддаляється поза зону атаки (attackRange), ворог повертається в стан

патрулювання. Якщо ворог готовий до атаки (canAttack == true), він викликає метод Attack() інтерфейсу IEnemy. Якщо вказано stopMovingWhileAttacking, ворог зупиняється під час атаки, інакше продовжує рух до цілі. Після цього запускається затримка між атаками через корутину AttackCooldownRoutine(), яка дозволяє керувати частотою атак.

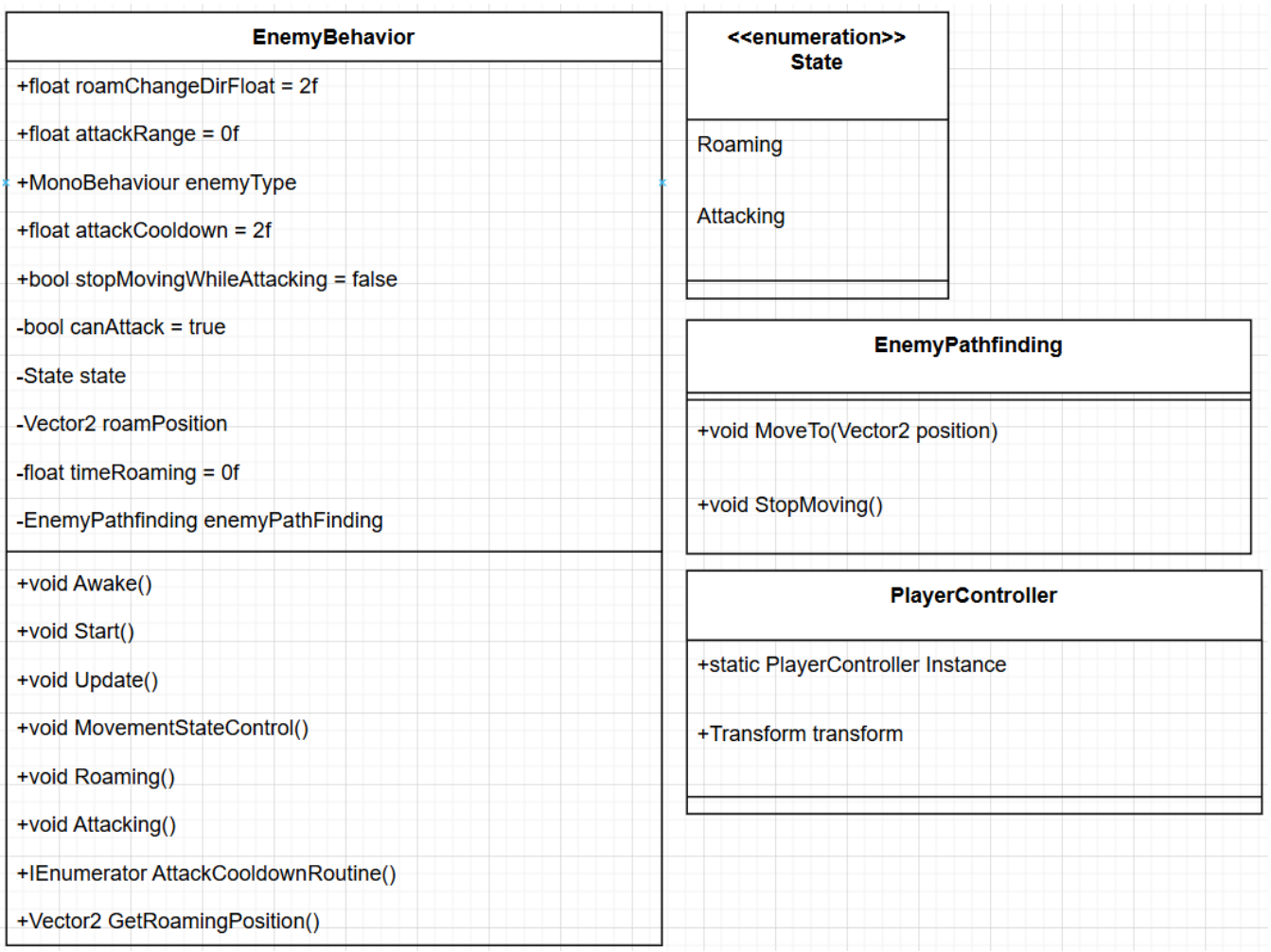

Рисунок 4.2 – Скрипт EnemyAI.cs

Цей підхід дозволяє створити ворогів, які адаптуються до с78итуації та ведуть себе природно — патрулюють, помічають гравця і вступають у бій.

Система інвентарю, реалізовано механізм збору предметів, збереження їх у слотах. Можливість використання предметів.

Приклад коду переміщення предметів:

```
private bool SwapItems(SlotScript from)
{
    if (IsEmpty)
    {
        return false;
    }
    if(from.MyItem.GetType() != MyItem.GetType() ||
from.MyCount+MyCount > MyItem.MyStackSize)
    {
        ObservableStack<Item> tmpFrom = new
ObservableStack<Item>(from.items);

        from.items.Clear();

        from.AddItems(items);

        items.Clear();

        AddItems(tmpFrom);

        return true;
    }

    return false;
   }
```

Метод SwapItems(SlotScript from) виконує обмін вмістом між двома слотами інвентаря. Його основна роль — вирішити, чи можна обмінятися предметами між поточним слотом і переданим у параметрі, а також, якщо можливо, провести сам обмін. Метод спочатку перевіряє, чи порожній поточний слот (IsEmpty). Якщо так — обмін неможливий, і повертається false.

Далі йде перевірка умов обміну: якщо предмети в слотах різного типу або якщо загальна кількість предметів перевищує максимальний допустимий стек, відбувається перестановка — вміст слотів міняється місцями. Для цього створюється тимчасовий стек, щоб зберегти вміст одного слота, після чого обидва слоти очищуються й

наповнюються новими значеннями. Якщо ж предмети однакові за типом і їх можна скласти разом — метод повертає false, бо обмін не потрібен (ймовірно, буде викликана інша логіка для складання).

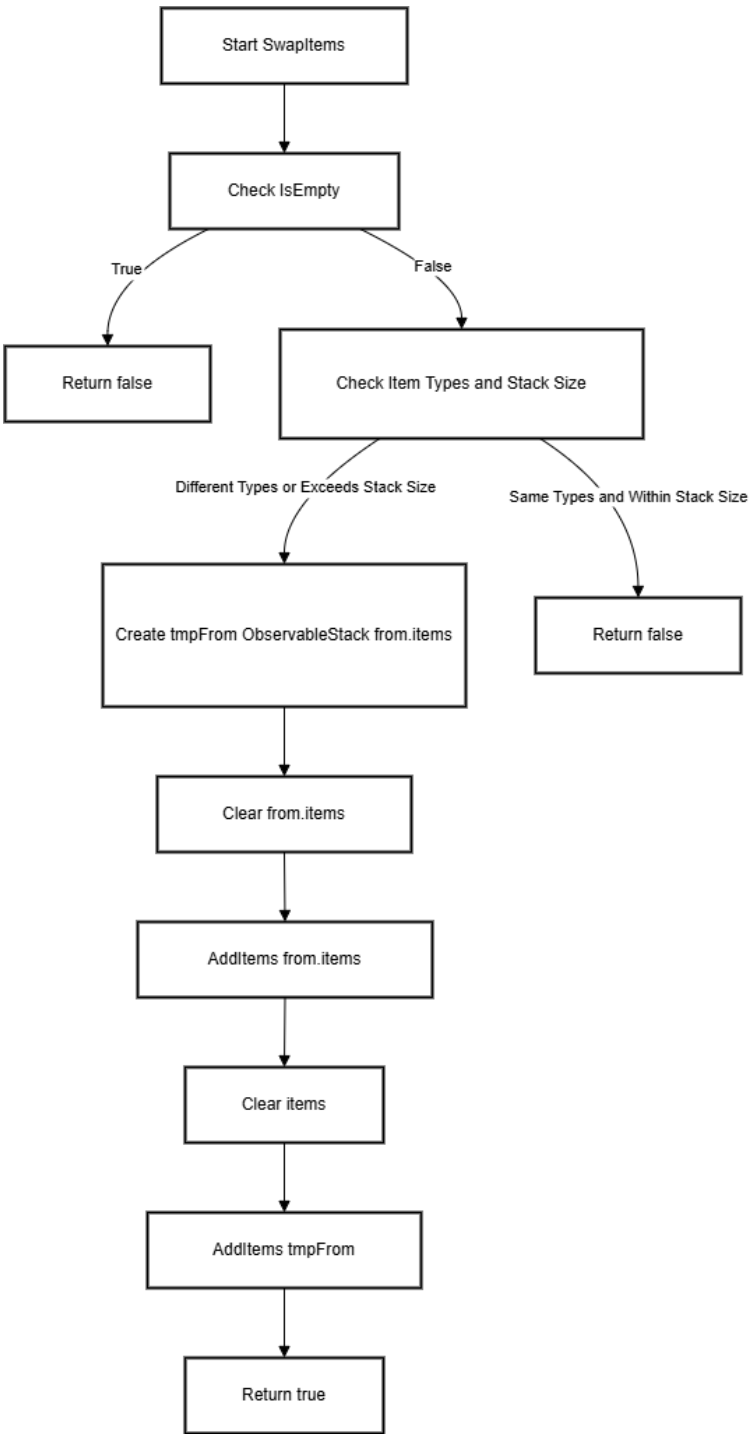

Рисунок 4.3 – Блок-схема переміщення предметів

Інвентар інтегровано з системою UI та квестами, де певні предмети можуть бути необхідними для виконання завдань.

Усі скрипти проєкту структуровано відповідно до призначення, що забезпечує логічну організацію коду та полегшує навігацію у проєкті.

4.2 Керівництво користувача

Даний розділ містить інструкції щодо взаємодії користувача з розробленим прототипом гри. Враховуючи наявний функціонал, реалізовано зручний інтерфейс керування персонажем, системою інвентарю, квестами, а також бойовими діями.

Запуск гри (користувацький досвід):

– запустити виконуваний файл гри (Build);

– при запуску завантажується головне меню, в якому можна обрати «Грати», «Завантажити гру», «Зберегти гру», «Налаштування», «Закрити гру»;

– після запуску гри завантажується ігрове середовище з головним героєм, базовим UI, противниками та інтерактивними об'єктами.

Основне керування.

Таблиця 1.2 – Керування гри

Дія	Клавіша / Комбінація клавіш
Рух персонажа	W, A, S, D або стрілки
Атака	Ліва кнопка миші (LMB)
Меч	1
Чарівна палка	2
Лук	3
Книга спелів	P
Огняна куля	Q

Кінець таблиці 1.2

Льодяний постріл	E
Пострій молній	R
Відкрити інвентар	B
Відкрити спорядження	I
Перегляд квестів	M
Взаємодія з об’єктом	F

Ігровий процес.

Персонаж гравця автоматично орієнтується в напрямку курсора миші.

При натисканні ЛКМ виконується анімація атаки та завдається шкода ворогу (за умови перебування у зоні атаки).

З повалених ворогів випадково випадає лут: зілля, предмети екіпірування, або квестові об'єкти.

Користувач може збирати ці об'єкти, наближуючись до них та натискаючи E відкривши вікно луту ворога.

Також можна переходити на інші локації, та досліджувати їх.

Інвентар.

В інвентарі гравець може переглядати отримані речі: зброю, броню, зілля.

Доступна можливість використати або обладнати обраний предмет.

Квести.

Система квестів дозволяє приймати завдання від NPC.

Активний квест можна відкрити на "M", відкриється квест лог з описом мети та прогресом виконання.

Після виконання завдання — гравець отримує винагороду та досвід.

Інтерфейс користувача (UI).

Панель здоров’я гравця, прогресія рівню та сам рівень гравця, кількість монет,

стаміна відображається у верхньому лівому куті.

Спели у нижньому лівому куті.

Снизу доступна зброя.

и правий кут, інвентар.

Висновки до четвертого розділу.

У четвертому розділі було розглянуто ключові аспекти програмної реалізації гри у жанрі action-RPG, а також надано керівництво користувача для взаємодії з прототипом. Реалізація гри охоплює всі основні елементи, необхідні для забезпечення функціонального, інтерактивного та захоплюючого геймплею.

Було розроблено і впроваджено модулі управління персонажем, бойової системи, інвентарю, системи квестів та штучного інтелекту ворогів. Значна увага приділялася деталізації механік: переміщення, атаки, використання вмінь, а також візуалізації ефектів і зворотного зв’язку з користувачем. Код кожного з компонентів структуровано логічно та підтримує принципи об’єктно-орієнтованого програмування, що полегшує супровід і масштабування проєкту.

Описаний у підрозділі 4.2 інтерфейс користувача та схема керування дозволяють гравцю інтуїтивно взаємодіяти з усіма елементами гри — від переміщення до використання предметів та виконання завдань. Це забезпечує комфортну ігрову взаємодію навіть для нових користувачів.

У цьому розділі також наведено ключові схеми, що ілюструють роботу ігрових механік, використання шаблонів проєктування, а також послідовність виконання основних процесів.

Таким чином, програмна реалізація є повноцінним і функціональним прототипом гри, який демонструє засвоєння основ розробки ігрових систем в середовищі Unity та готовність до подальшого розвитку й удосконалення проєкту.

ВИСНОВКИ

У межах виконання кваліфікаційної роботи було реалізовано індивідуальний ігровий проєкт у жанрі 2D action-RPG з використанням рушія Unity. В результаті було створено повноцінний прототип гри, що включає базову бойову систему ближнього бою, елементи штучного інтелекту ворогів, систему взаємодії з ігровим середовищем, а також реалізацію інтерфейсу користувача (UI), анімацій та системи прогресу персонажа.

У процесі розробки поглиблено практичні навички роботи з середовищем Unity, мовою програмування C#, редактором Visual Studio 2022, а також з інструментами для створення анімацій, налаштування фізики, систем колізій, організації сцен та побудови рівнів. Було сформовано розуміння принципів побудови архітектури ігрових проєктів, зокрема через компонентний підхід та ефективну структуру коду.

Отриманий результат засвідчив спроможність застосовувати набуті знання у реальному проєкті, що охоплює повний цикл розробки гри: від концептуального задуму до реалізації ключових геймплейних механік. Виконання цієї кваліфікаційної роботи стало важливим етапом у професійному зростанні та підтвердило актуальність обраної теми як з точки зору освітньої підготовки, так і з перспектив подальшої діяльності в галузі розробки комп’ютерних ігор.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1) "Rules of Play: Game Design Fundamentals" (Salen & Zimmerman): вебсайт. URL: https://gamifique.wordpress.com/wp-content/uploads/2011/11/1-rules-of-play-game-design-fundamentals.pdf
2) Документация по Visual Studio: вебсайт. URL: https://learn.microsoft.com/ru-ru/visualstudio/windows/?view=vs-2022
3) Unity Documentation — офіційна документація Unity. Unity Technologies: вебсайт. URL: https://docs.unity3d.com/Manual/index.html
4) C# Programming Guide — Microsoft Docs: вебсайт. URL: https://learn.microsoft.com/en-us/dotnet/csharp/
5) Tilemap — 2D Tilemap Documentation. Unity: вебсайт. URL: https://docs.unity3d.com/Manual/Tilemap.html
6) Animator and Animation — Unity Manual: вебсайт. URL: https://docs.unity3d.com/Manual/AnimationSection.html
7) Cinemachine — Unity Documentation: вебсайт. URL: https://docs.unity3d.com/Packages/com.unity.cinemachine@2.8/manual/index.html
8) TextMesh Pro — Unity Manual: вебсайт. URL: https://docs.unity3d.com/Packages/com.unity.textmeshpro@3.0/manual/index.html
9) Unity Learn — офіційна навчальна платформа Unity: вебсайт. URL: https://learn.unity.com/
10) Brackeys YouTube Channel — навчальні відео з розробки ігор на Unity. URL: https://www.youtube.com/user/Brackeys
11) "Game Programming Patterns" (Robert Nystrom) — книга про шаблони програмування у розробці ігор. URL: https://gameprogrammingpatterns.com/

12) Microsoft Learn — курси по C# і .NET Framework. URL: https://learn.microsoft.com/en-us/training/dotnet/

13) Unity Asset Store — офіційний магазин активів для Unity. URL: https://assetstore.unity.com/

14) GitHub — платформа для розміщення коду та спільної роботи над проєктами. URL: https://github.com/

15) "The Art of Game Design: A Book of Lenses" (Jesse Schell) — книга по геймдизайну. URL: https://www.amazon.com/Art-Game-Design-Book-Lenses/dp/0123694965

16) Unity Physics Documentation — офіційна документація по фізиці Unity. URL: https://docs.unity3d.com/Manual/Physics.html

17) "Introduction to Algorithms" (Cormen, Leiserson, Rivest, Stein) — книга для поглибленого розуміння алгоритмів. URL: https://mitpress.mit.edu/books/introduction-algorithms-third-edition

18) Unity Forum — офіційний форум спільноти Unity. URL: https://forum.unity.com/

19) "Clean Code" (Robert C. Martin) — книга по написанню чистого коду. URL: https://www.oreilly.com/library/view/clean-code-a/9780136083238/

20) "Effective C#" (Bill Wagner) — книга по ефективному використанню мови C#. URL: https://www.oreilly.com/library/view/effective-c-50/9780134579551/

21) Unity Manual — Lighting — офіційна документація по освітленню в Unity. URL: https://docs.unity3d.com/Manual/LightingSection.html

22) Картинка емблеми Unity URL: https://upload.wikimedia.org/wikipedia/commons/thumb/c/c4/Unity_2021.svg/250px-Unity_2021.svg.png

23) Картинка емблеми Visual Studio 2022 URL: https://upload.wikimedia.org/wikipedia/commons/thumb/2/2c/Visual_Studio_Icon_2022.svg/120px-Visual_Studio_Icon_2022.svg.png

24) "Level Up! The Guide to Great Video Game Design" (Scott Rogers) — книга з практичними порадами для геймдизайнерів. URL: https://www.amazon.com/Level-Up-Guide-Video-Design/dp/1118877160/

25) Stack Overflow — спільнота програмістів, часто використовувана для вирішення проблем з C#, Unity, .NET тощо.URL: https://stackoverflow.com/

26) Raywenderlich Game Tutorials — гайди з розробки ігор, зокрема на Unity.URL: https://www.raywenderlich.com/unity

27) Bhadeshia H. K. D. H. Neural Networks in Materials Science : ISIJ International 39, 1999. (10): 966–979 с.

28) Gold J. Object-Oriented Game Development. UK : Pearson Education Limited, 2004. 404 с

29) Wolf M. J. P. Encyclopedia of Video Games: The Culture, Technology, and Art of Gaming / Ed. by Mark J. P. Wolf. Santa Barbara, CA :Greenwood, 2012. 740 с.

30) Learn C# by Microsoft — безкоштовний курс для вивчення C# від Microsoft. URL: https://learn.microsoft.com/en-us/training/paths/csharp-first-steps/

ДОДАТОК

КОД ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Лістинг коду PlayerControlls.cs:

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

public class PlayerController : Singleton<PlayerController>
{
    public bool FacingLeft { get { return facingLeft; } set { facingLeft = value; } }

    [SerializeField] private float moveSpeed = 1f;
    [SerializeField] private float dashSpeed = 4f;
    [SerializeField] private TrailRenderer myTrailRenderer;
    [SerializeField] private Transform weaponCollider;

    private InputSystem_Actions playerControls;
    private Vector2 movement;
    private Rigidbody2D rb;
    private Animator myAnimator;
    private Knockback knockback;
    private SpriteRenderer mySpriteRender;
    private float startingMoveSpeed;

    private bool facingLeft = false;
    private bool isDashing = false;

    private void Start()
    {
        playerControls.Player.Dash.performed += _ => Dash();

        startingMoveSpeed = moveSpeed;

        ActiveInventory.Instance.EquipStartingWeapon();
    }

    protected override void Awake()
    {
        base.Awake();
        playerControls = new InputSystem_Actions();
        rb = GetComponent<Rigidbody2D>();
        myAnimator = GetComponent<Animator>();
        mySpriteRender = GetComponent<SpriteRenderer>();
        knockback = GetComponent<Knockback>();
    }

    private void OnEnable()
    {
        playerControls.Enable();
    }

    private void OnDisable()
    {
        playerControls.Disable();
```

```
    }

    private void Update()
    {
        PlayerInput();

        if (Input.GetKeyDown(KeyCode.Q))
        {
            StartCoroutine(CastSpell(0));
        }
        if (Input.GetKeyDown(KeyCode.E))
        {
            StartCoroutine(CastSpell(1));
        }
        if (Input.GetKeyDown(KeyCode.R))
        {
            StartCoroutine(CastSpell(2));
        }
    }

    private void FixedUpdate()
    {
        AdjustPlayerFacingDirection();
        Move();
    }

    public Transform GetWeaponCollider()
    {
        return weaponCollider;
    }

    private void PlayerInput()
    {
        movement = playerControls.Player.Move.ReadValue<Vector2>();

        myAnimator.SetFloat("moveX", movement.x);
        myAnimator.SetFloat("moveY", movement.y);
    }

    private void Move()
    {
        if(knockback.GettingKnockedBack || PlayerHealth.Instance.isDead ) { return; }

        rb.MovePosition(rb.position + movement * (moveSpeed * Time.fixedDeltaTime));
    }

    private void AdjustPlayerFacingDirection()
    {
        Vector3 mousePos = Input.mousePosition;
        Vector3 playerScreenPoint =
Camera.main.WorldToScreenPoint(transform.position);

        if (mousePos.x < playerScreenPoint.x)
        {
            mySpriteRender.flipX = true;
            FacingLeft = true;
        }
```

```
        else
        {
            mySpriteRender.flipX = false;
            FacingLeft = false;
        }
    }

    private void Dash()
    {
        if (!isDashing && Stamina.Instance.CurrentStamina > 0)
        {
            Stamina.Instance.UseStamina();
            isDashing = true;
            moveSpeed *= dashSpeed;
            myTrailRenderer.emitting = true;
            StartCoroutine(EndDashRoutine());
        }
    }

    private IEnumerator EndDashRoutine()
    {
        float dashTime = .2f;
        float dashCD = .25f;
        yield return new WaitForSeconds(dashTime);
        moveSpeed = startingMoveSpeed;
        myTrailRenderer.emitting = false;
        yield return new WaitForSeconds(dashCD);
        isDashing = false;
    }

    public IEnumerator CastSpell(int spellIndex)
    {
        Spell s = SpellBook.MyInstance.CastSpell(spellIndex);

        if (s == null) yield break;

        yield return new WaitForSeconds(s.MyCastTime);

        Instantiate(s.MySpellPrefab, transform.position, Quaternion.identity);

        SpellBook.MyInstance.StopCasting();
    }
}
```

Лістинг коду EnemyAI.cs:

```
using UnityEngine;
using System.Collections;

public class EnemyAI : MonoBehaviour
{
    [SerializeField] private float roamChangeDirFloat = 2f;
    [SerializeField] private float attackRange = 0f;
    [SerializeField] private MonoBehaviour enemyType;
    [SerializeField] private float attackCooldown = 2f;
    [SerializeField] private bool stopMovingWhileAttacking = false;

    private bool canAttack = true;
```

```
        private enum State
        {
            Roaming,
            Attacking
        }

        private Vector2 roamPosition;
        private float timeRoaming = 0f;

        private State state;
        private EnemyPathfinding enemyPathFinding;

        private void Awake()
        {
            enemyPathFinding = GetComponent<EnemyPathfinding>();
            state = State.Roaming;
        }

        private void Start()
        {
            roamPosition = GetRoamingPosition();
        }

        private void Update()
        {
            MovementStateControl();
        }

        private void MovementStateControl()
        {
            switch (state)
            {
                default:
                case State.Roaming:
                    Roaming();
                    break;

                case State.Attacking:
                    Attacking();
                    break;
            }
        }

        private void Roaming()
        {
            timeRoaming += Time.deltaTime;

            enemyPathFinding.MoveTo(roamPosition);

            if                                  (Vector2.Distance(transform.position,
PlayerController.Instance.transform.position) < attackRange)
            {
                state = State.Attacking;
            }

            if (timeRoaming > roamChangeDirFloat)
```

```
            {
                roamPosition = GetRoamingPosition();
            }
        }

        private void Attacking()
        {
            if                                  (Vector2.Distance(transform.position,
PlayerController.Instance.transform.position) > attackRange)
            {
                state = State.Roaming;
            }

            if (attackRange != 0 && canAttack)
            {
                canAttack = false;
                (enemyType as IEnemy).Attack();

                if (stopMovingWhileAttacking)
                {
                    enemyPathFinding.StopMoving();
                }
                else
                {
                    enemyPathFinding.MoveTo(roamPosition);
                }

                StartCoroutine(AttackCooldownRoutine());
            }
        }

        private IEnumerator AttackCooldownRoutine()
        {
            yield return new WaitForSeconds(attackCooldown);
            canAttack = true;
        }

        private Vector2 GetRoamingPosition()
        {
            timeRoaming = 0f;
            return    new    Vector2(Random.Range(-1f,    1f),    Random.Range(-1f,
1f)).normalized;
        }
    }
```

Лістинг коду EnemyPathfinding.cs:

```
using UnityEngine;

public class EnemyPathfinding : MonoBehaviour
{
    [SerializeField] private float moveSpeed = 2f;

    private Rigidbody2D rb;
    private Vector2 moveDir;
    private Knockback knockback;
    private SpriteRenderer spriteRenderer;
```

```
        private void Awake()
        {
            knockback = GetComponent<Knockback>();
            rb = GetComponent<Rigidbody2D>();
            spriteRenderer = GetComponent<SpriteRenderer>();
        }

        private void FixedUpdate()
        {
            if(knockback.GettingKnockedBack) { return;  }
            rb.MovePosition(rb.position     +     moveDir     *     (moveSpeed     *
Time.fixedDeltaTime));

            if(moveDir.x < 0)
            {
                spriteRenderer.flipX = true;
            }
            else if((moveDir.x > 0))
            {
                spriteRenderer.flipX = false;
            }
        }

        public void MoveTo(Vector2 targetPosition)
        {
            moveDir = targetPosition;
        }

        public void StopMoving()
        {
            moveDir = Vector3.zero;
        }
    }
```

Лістинг коду EnemyPathfinding.cs:

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

    [CreateAssetMenu(fileName = "quest", menuName = "quest/quest", order = 1)]
    [System.Serializable]
    public class Quest : ScriptableObject
    {
        [SerializeField]
        private string title;

        [SerializeField]
        private string description;

        [SerializeField]
        private int coinReward;

        [SerializeField]
        private int expReward;

        [SerializeField]
        private CollectObjective[] collectObjectives;
```

```
[SerializeField]
private KillObjective[] killObjectives;

public QuestScript MyQuestScript { get; set; }

public QuestGiver MyQuestGiver { get; set; }

public string MyTitle
{
    get
    {
        return title;
    }
    set
    {
        title = value;
    }
}

public string MyDescription
{
    get
    {
        return description;
    }
    set
    {
        description = value;
    }
}

public int MyCoinReward
{
    get
    {
        return coinReward;
    }
    set
    {
        coinReward = value;
    }
}

public int MyExpReward
{
    get
    {
        return expReward;
    }
    set
    {
        expReward = value;
    }
}

public CollectObjective[] MyCollectObjectives
```

```
    {
        get
        {
            return collectObjectives;
        }
    }

    public KillObjective[] MyKillObjectives
    {
        get
        {
            return killObjectives;
        }
    }

    public bool IsComplete
    {
        get
        {
            foreach (Objective obj in collectObjectives) {
                if (!obj.IsComplete)
                {
                    return false;
                }
            }

            foreach (Objective obj in killObjectives)
            {
                if (!obj.IsComplete)
                {
                    return false;
                }
            }

            return true;
        }
    }
}


[System.Serializable]
public abstract class Objective
{
    [SerializeField]
    public int amount;

    [SerializeField]
    private int currentAmount;

    [SerializeField]
    private string type;

    public int MyAmount
    {
        get
        {
            return amount;
```

```
            }
        }

        public int MyCurrentAmount
        {
            get
            {
                return currentAmount;
            }
            set
            {
                currentAmount = value;
            }
        }

        public string MyType
        {
            get
            {
                return type;
            }
            set
            {
                type = value;
            }
        }

        public bool IsComplete
        {
            get
            {
                return MyCurrentAmount >= MyAmount;
            }
        }
    }

    [System.Serializable]
    public class CollectObjective: Objective
    {
        public void UpdateItemCount(Item item)
        {
            if (MyType.ToLower() == item.MyTitle.ToLower())
            {
                MyCurrentAmount                                                    =
InventoryScript.MyInstance.GetItemCount(item.MyTitle);

                if(MyCurrentAmount <= MyAmount)
                {
                    MessageFeedManager.MyInstance.WriteMessage(string.Format("{0}:
{1}/{2}", item.MyTitle, MyCurrentAmount, MyAmount));
                }

                QuestLog.MyInstance.CheckCompletion();
                QuestLog.MyInstance.UpdateSelected();
            }
        }
```

```
        public void UpdateItemCount()
        {
            MyCurrentAmount = InventoryScript.MyInstance.GetItemCount(MyType);

            QuestLog.MyInstance.CheckCompletion();
            QuestLog.MyInstance.UpdateSelected();
        }

        public void Complete()
        {
            Stack<Item>    items    =    InventoryScript.MyInstance.GetItems(MyType,
MyAmount);

            foreach (Item item in items)
            {
                item.Remove();
            }
        }
    }


    [System.Serializable]
    public class KillObjective : Objective
    {
        public void UpdateKillCount(IEnemy enemy)
        {
            if(MyType == enemy.GetType())
            {
                MyCurrentAmount++;

                if (MyCurrentAmount <= MyAmount)
                {
                    MessageFeedManager.MyInstance.WriteMessage(string.Format("{0}:
{1}/{2}", enemy.GetType(), MyCurrentAmount, MyAmount));
                }

                QuestLog.MyInstance.CheckCompletion();
                QuestLog.MyInstance.UpdateSelected();
            }
        }
    }
```

Лістинг коду Quest.cs:

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

    [CreateAssetMenu(fileName = "quest", menuName = "quest/quest", order = 1)]
    [System.Serializable]
    public class Quest : ScriptableObject
    {
        [SerializeField]
        private string title;

        [SerializeField]
        private string description;
```

```
[SerializeField]
private int coinReward;

[SerializeField]
private int expReward;

[SerializeField]
private CollectObjective[] collectObjectives;

[SerializeField]
private KillObjective[] killObjectives;

public QuestScript MyQuestScript { get; set; }

public QuestGiver MyQuestGiver { get; set; }

public string MyTitle
{
    get
    {
        return title;
    }
    set
    {
        title = value;
    }
}

public string MyDescription
{
    get
    {
        return description;
    }
    set
    {
        description = value;
    }
}

public int MyCoinReward
{
    get
    {
        return coinReward;
    }
    set
    {
        coinReward = value;
    }
}

public int MyExpReward
{
    get
    {
        return expReward;
```

```
        }
        set
        {
            expReward = value;
        }
    }

    public CollectObjective[] MyCollectObjectives
    {
        get
        {
            return collectObjectives;
        }
    }

    public KillObjective[] MyKillObjectives
    {
        get
        {
            return killObjectives;
        }
    }

    public bool IsComplete
    {
        get
        {
            foreach (Objective obj in collectObjectives) {
                if (!obj.IsComplete)
                {
                    return false;
                }
            }

            foreach (Objective obj in killObjectives)
            {
                if (!obj.IsComplete)
                {
                    return false;
                }
            }

            return true;
        }
    }
}


[System.Serializable]
public abstract class Objective
{
    [SerializeField]
    public int amount;

    [SerializeField]
    private int currentAmount;
```

```
        [SerializeField]
        private string type;

        public int MyAmount
        {
            get
            {
                return amount;
            }
        }

        public int MyCurrentAmount
        {
            get
            {
                return currentAmount;
            }
            set
            {
                currentAmount = value;
            }
        }

        public string MyType
        {
            get
            {
                return type;
            }
            set
            {
                type = value;
            }
        }

        public bool IsComplete
        {
            get
            {
                return MyCurrentAmount >= MyAmount;
            }
        }
    }

    [System.Serializable]
    public class CollectObjective: Objective
    {
        public void UpdateItemCount(Item item)
        {
            if (MyType.ToLower() == item.MyTitle.ToLower())
            {
                MyCurrentAmount                                                    =
InventoryScript.MyInstance.GetItemCount(item.MyTitle);

                if(MyCurrentAmount <= MyAmount)
                {
                    MessageFeedManager.MyInstance.WriteMessage(string.Format("{0}:
```

```
{1}/{2}", item.MyTitle, MyCurrentAmount, MyAmount));
                }

                QuestLog.MyInstance.CheckCompletion();
                QuestLog.MyInstance.UpdateSelected();
            }
        }

        public void UpdateItemCount()
        {
            MyCurrentAmount = InventoryScript.MyInstance.GetItemCount(MyType);

            QuestLog.MyInstance.CheckCompletion();
            QuestLog.MyInstance.UpdateSelected();
        }

        public void Complete()
        {
            Stack<Item>    items    =    InventoryScript.MyInstance.GetItems(MyType,
MyAmount);

            foreach (Item item in items)
            {
                item.Remove();
            }
        }
    }


    [System.Serializable]
    public class KillObjective : Objective
    {
        public void UpdateKillCount(IEnemy enemy)
        {
            if(MyType == enemy.GetType())
            {
                MyCurrentAmount++;

                if (MyCurrentAmount <= MyAmount)
                {
                    MessageFeedManager.MyInstance.WriteMessage(string.Format("{0}:
{1}/{2}", enemy.GetType(), MyCurrentAmount, MyAmount));
                }

                QuestLog.MyInstance.CheckCompletion();
                QuestLog.MyInstance.UpdateSelected();
            }
        }
    }
```

Лістинг коду QuestLog.cs:

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

public class QuestLog : MonoBehaviour
```

```
{
    [SerializeField]
    private GameObject questPrefab;

    [SerializeField]
    private Transform questParent;

    private Quest selected;

    [SerializeField]
    private TMP_Text questDescription;

    [SerializeField]
    private CanvasGroup canvasGroup;

    [SerializeField]
    private TMP_Text questCountTxt;

    [SerializeField]
    private int maxCount;

    private int currentCount;

    private List<QuestScript> questScripts = new List<QuestScript>();

    private List<Quest> quests = new List<Quest>();

    private static QuestLog instance;

    public static QuestLog MyInstance
    {
        get
        {
            if(instance == null)
            {
                instance = FindObjectOfType<QuestLog>();
            }
            return instance;
        }
    }

    private void Start()
    {
        questCountTxt.text = currentCount.ToString() + "/" + maxCount.ToString();
    }

    public void Update()
    {
        if (Input.GetKeyDown(KeyCode.M))
        {
            OpenClose();
        }
    }

    public void AcceptQuest(Quest quest)
    {
        if (currentCount < maxCount)
```

```
            {
                currentCount++;

                questCountTxt.text = currentCount.ToString() + "/" +
maxCount.ToString();

                foreach (CollectObjective o in quest.MyCollectObjectives)
                {
                    InventoryScript.MyInstance.itemCountChangedEvent += new
ItemCountChanged(o.UpdateItemCount);
                    o.UpdateItemCount();
                }

                foreach (KillObjective o in quest.MyKillObjectives)
                {
                    GameManager.MyInstance.killConfirmedEvent += new
KillConfirmed(o.UpdateKillCount);
                    o.MyCurrentAmount = 0;
                }

                quests.Add(quest);

                GameObject go = Instantiate(questPrefab, questParent);

                QuestScript qs = go.GetComponent<QuestScript>();
                quest.MyQuestScript = qs;
                qs.MyQuest = quest;

                questScripts.Add(qs);

                go.GetComponent<TextMeshProUGUI>().text = quest.MyTitle;

                CheckCompletion();
            }
        }

        public void UpdateSelected()
        {
            ShowDescription(selected);
        }

        public void ShowDescription(Quest quest)
        {
            if (quest != null)
            {
                if (selected != null && selected != quest)
                {
                    selected.MyQuestScript.DeSelect();
                }

                string objective = string.Empty;

                selected = quest;

                foreach (Objective obj in quest.MyCollectObjectives)
                {
                    objective += obj.MyType + ": " + obj.MyCurrentAmount + "/" +
```

```
obj.MyAmount + "\n";
                }

                foreach (Objective obj in quest.MyKillObjectives)
                {
                    objective += obj.MyType + ": " + obj.MyCurrentAmount + "/" +
obj.MyAmount + "\n";
                }

                questDescription.text                                                =
string.Format("{0}\n<size=32>{1}\n\nObjectives\n{2}</size>",           quest.MyTitle,
quest.MyDescription, objective);
            }
        }

        public void CheckCompletion()
        {
            foreach(QuestScript qs in questScripts)
            {
                qs.MyQuest.MyQuestGiver.UpdateQuestStatus();
                qs.IsComplete();
            }
        }

        public void OpenClose() {
            if(canvasGroup.alpha == 1)
            {
                Close();
            }
            else{
                canvasGroup.alpha = 1;
                canvasGroup.blocksRaycasts = true;
            }
        }

        public void Close()
        {
            canvasGroup.alpha = 0;
            canvasGroup.blocksRaycasts = false;
        }

        public bool HasQuest(Quest quest)
        {
            return quests.Exists(x => x.MyTitle == quest.MyTitle);
        }

        public void AbandonQuests()
        {
            foreach (CollectObjective o in selected.MyCollectObjectives)
            {
                InventoryScript.MyInstance.itemCountChangedEvent       -=       new
ItemCountChanged(o.UpdateItemCount);
            }

            foreach (KillObjective o in selected.MyKillObjectives)
            {
                GameManager.MyInstance.killConfirmedEvent          -=          new
```

```
KillConfirmed(o.UpdateKillCount);
            }

            RemoveQuest(selected.MyQuestScript);
        }

        public void RemoveQuest(QuestScript qs)
        {
            questScripts.Remove(qs);
            Destroy(qs.gameObject);
            quests.Remove(qs.MyQuest);
            questDescription.text = string.Empty;
            selected = null;
            currentCount--;
            questCountTxt.text = currentCount.ToString() + "/" + maxCount.ToString();
            qs.MyQuest.MyQuestGiver.UpdateQuestStatus();
            qs = null;
        }
    };
```

Лістинг коду Vendor.cs:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Vendor : MonoBehaviour
{
    [SerializeField]
    private VendorWindow vendorWindow;

    [SerializeField] private int activeSceneIndex = 0;

    [SerializeField]
    private VendorItem[] items;

    private bool isPlayerInTrigger = false;

    public bool IsOpen { get; set; }

    private void Awake()
    {
        DontDestroyOnLoad(gameObject);
    }

    private void Start()
    {
        CheckScene(SceneManager.GetActiveScene());
        SceneManager.sceneLoaded += OnSceneLoaded;
    }

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

    private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
    {
        CheckScene(scene);
```

```
}

private void CheckScene(Scene scene)
{
    if (scene.buildIndex != activeSceneIndex)
    {
        gameObject.SetActive(false);
    }
    else
    {
        gameObject.SetActive(true);
    }
}


void Update()
{
    if (isPlayerInTrigger && Input.GetKeyDown(KeyCode.F))
    {
        Interact();
    }
}

private void Interact()
{
    if (!IsOpen)
    {
        IsOpen= true;
        vendorWindow.CreatePages(items);
        vendorWindow.Open(this);
    }
}

public void StopInteract()
{
    if (IsOpen)
    {
        IsOpen = false;
        vendorWindow.Close();
    }
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.name == "Player")
    {
        isPlayerInTrigger = true;
    }
}

private void OnCollisionExit2D(Collision2D collision)
{
    if (collision.gameObject.name == "Player")
    {
        isPlayerInTrigger = false;
    }
}
```

```
}
```

Лістинг коду InventoryScript.cs:

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

public delegate void ItemCountChanged(Item item);

public class InventoryScript : MonoBehaviour
{
    public event ItemCountChanged itemCountChangedEvent;

    private static InventoryScript instance;

    public static InventoryScript MyInstance
    {
        get
        {
            if(instance == null)
            {
                instance = FindObjectOfType<InventoryScript>();
            }

            return instance;
        }
    }

    [SerializeField]
    private Item[] items;

    [SerializeField]
    private BagButton[] bagButtons;

    private SlotScript fromSlot;

    private List<Bag> bags = new List<Bag>();

    public bool CanAddBag
    {
        get
        {
            return bags.Count < 4;
        }
    }

    public SlotScript FromSlot
    {
        get
        {
            return fromSlot;
        }

        set
        {
            fromSlot = value;
            if (value != null)
```

```
            {
                fromSlot.MyIcon.color = Color.grey;
            }
        }
    }

    public int MyEmptySlotCount
    {
        get
        {
            int count = 0;

            foreach (Bag bag in bags)
            {
                count += bag.MyBagScript.MyEmptySlotCount;
            }

            return count;
        }
    }

    public int MyTotalSlotCount
    {
        get
        {
            int count = 0;

            foreach(Bag bag in bags)
            {
                count += bag.MyBagScript.MySlots.Count;
            }

            return count;
        }
    }

    public int MyFullSlotCount
    {
        get
        {
            return MyTotalSlotCount - MyEmptySlotCount;
        }
    }

    public int GetItemCount(string type)
    {
        int itemCount = 0;

        foreach (Bag bag in bags) {
            foreach(SlotScript slot in bag.MyBagScript.MySlots)
            {
                if(!slot.IsEmpty && slot.MyItem.MyTitle == type)
                {
                    itemCount += slot.MyItems.Count;
                }
            }
        }
```

```
    return itemCount;
}

private void Awake()
{
    Bag bag = (Bag)Instantiate(items[0]);
    bag.Initialize(16);
    bag.Use();
}

private void Update()
{
    if (Input.GetKeyDown(KeyCode.J))
    {
        Bag bag = (Bag)Instantiate(items[0]);
        bag.Initialize(16);
        bag.Use();
    }
    if (Input.GetKeyDown(KeyCode.K))
    {
        Bag bag = (Bag)Instantiate(items[0]);
        bag.Initialize(12);
        AddItem(bag);
    }
    if (Input.GetKeyDown(KeyCode.L))
    {
        HealthPotion potion = (HealthPotion)Instantiate(items[1]);
        AddItem(potion);
    }
    if (Input.GetKeyDown(KeyCode.H))
    {
        Armor armor = (Armor)Instantiate(items[2]);
        AddItem(armor);
    }
    if (Input.GetKeyDown(KeyCode.O))
    {
        Armor armor = (Armor)Instantiate(items[3]);
        AddItem(armor);
    }
}

public void RemoveBag(Bag bag)
{
    bags.Remove(bag);
    Destroy(bag.MyBagScript.gameObject);
}

public void AddBag(Bag bag)
{
    foreach (BagButton bagbutton in bagButtons)
    {
        if(bagbutton.MyBag == null)
        {
            bagbutton.MyBag = bag;
            bags.Add(bag);
            bag.MyBagButton = bagbutton;
```

```
bag.MyBagScript.transform.SetSiblingIndex(bagbutton.MyBagIndex);
                    break;
                }
            }
        }

        public void AddBag(Bag bag, BagButton bagButton)
        {
            bags.Add(bag);
            bagButton.MyBag = bag;
        }

        public void SwapBags(Bag oldBag, Bag newBag)
        {
            int newSlotCount = (MyTotalSlotCount - oldBag.Slots) + newBag.Slots;

            if(newSlotCount - MyFullSlotCount >= 0)
            {
                List<Item> bagItems = oldBag.MyBagScript.GetItems();

                RemoveBag(oldBag);

                newBag.MyBagButton = oldBag.MyBagButton;

                newBag.Use();

                foreach (Item item in bagItems)
                {
                    if(item != newBag)
                    {
                        AddItem(item);
                    }
                }

                AddItem(oldBag);

                HandScript.MyInstance.Drop();

                MyInstance.fromSlot = null;
            }
        }

        private bool PlaceInEmpty(Item item)
        {
            foreach(Bag bag in bags)
            {
                if (bag.MyBagScript.AddItem(item))
                {
                    OnItemCountChanged(item);
                    return true;
                }
            }

            return false;
        }
```

```
private bool PlaceInStack(Item item)
{
    foreach(Bag bag in bags)
    {
        foreach(SlotScript slots in bag.MyBagScript.MySlots)
        {
            if (slots.StackItem(item))
            {
                OnItemCountChanged(item);
                return true;
            }
        }
    }

    return false;
}

public bool AddItem(Item item)
{
    if (item.MyStackSize > 0)
    {
        if (PlaceInStack(item))
        {
            return true;
        }
    }

    return PlaceInEmpty(item);
}

public void OpenClose()
{
    bool closedBags = bags.Find(x => !x.MyBagScript.IsOpen);

    foreach(Bag bag in bags)
    {
        if(bag.MyBagScript.IsOpen != closedBags)
        {
            bag.MyBagScript.OpenClose();
        }
    }
}

public Stack<Item> GetItems(string type, int count)
{
    Stack<Item> items = new Stack<Item>();

    foreach (Bag bag in bags)
    {
        foreach (SlotScript slot in bag.MyBagScript.MySlots)
        {
            if (!slot.IsEmpty && slot.MyItem.MyTitle == type)
            {
                foreach(Item item in slot.MyItems)
                {
                    items.Push(item);
```

```
                        if(items.Count == count)
                        {
                            return items;
                        }
                    }
                }
            }
        }

        return items;
    }

    public void OnItemCountChanged(Item item)
    {
        if (itemCountChangedEvent != null)
        {
            itemCountChangedEvent.Invoke(item);
        }
    }
}
```