Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА ГРИ «SPOOKY SCARY SKELETON» В ЖАНРІ SURVIVOR-LIKE З ВИКОРИСТАННЯМ UNITY

DEVELOPMENT OF THE GAME «SPOOKY SCARY SKELETON» IN THE SURVIVOR-LIKE GENRE USING UNITY

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗс-21
Давидюк Роман Миколайович

Керівник:
Сичук Віктор Анатолійович

Кваліфікаційну роботу
допущено до захисту
«10» 06 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *бакалавр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«20» 12 2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Давидюку Роману Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: «*Розробка гри "Spooky scary skeleton" в жанрі Survivor-like з використанням Unity»*
Керівник роботи: *к.т.н., доцент, Сичук В. А.*
затверджені наказом закладу вищої освіти від *«19» грудня 2024 р.* № *474/01–02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра *«10» червня 2025 р.*
3. Вихідні дані до роботи: *ігрові рушії, порівняльний аналіз ігрових рушіїв, Unity, UnrealEngine, Godot, GameEngine, Survivor-like, Vapire survivors, game development, розробка комп'ютерних ігор.*
4. Зміст розрахунково–пояснювальної записки (перелік питань, що потрібно розробити): *обгрунтування актуальності жанру Survivor-like, формулювання вимог і базового функціоналу гри, вибір інструментів розробки (ігрового рушія), розробка гри в жанрі Survivor-like з використанням обраного рушія, тестування роботи та основних механік гри, компіляція виконуваного файлу гри та створення інсталяційного пакету.*
5. Перелік графічного матеріалу: *16 рисунків, 4 таблиці, 4 додатки*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Сичук В. А.*		
Специфікація вимог до розробленої системи	*Сичук В. А.*		
Розробка об'єкта проектування	*Сичук В. А.*		
Нормоконтроль	*Вознюк А. В.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	2,76%		
Академічна доброчесність	*Сичук В. А.*		

7. Дата видачі завдання *«20» грудня 2024 р.*

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 04.02.2025 р.	Виконано
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 01.03.2025 р.	Виконано
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 15.03.2025 р.	Виконано
4	Розробка функціонально-структурної схеми роботи об'єкта проектування	до 29.03.2025 р.	Виконано
5	Практична реалізація об'єкта проектування	до 26.04.2025 р.	Виконано
6	Тестування та налагодження об'єкта проектування	до 03.05.2025 р.	Виконано
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 10.06.2025 р.	Виконано

Здобувач вищої освіти _____ Роман ДАВИДЮК

Керівник кваліфікаційної роботи _____ Віктор СИЧУК

АНОТАЦІЯ

Давидюк Р. М. Розробка гри «Spooky scary skeleton» в жанрі survivor-like з використанням Unity. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота бакалавра складається з вступу, трьох розділів, висновків і пропозицій, списку використаних джерел та додатків.

У першому розділі було здійснено комплексний аналіз сучасного стану індустрії розробки комп'ютерних ігор, зокрема жанру survivor-like. Досліджено ключові тенденції, успішні проєкти та перспективи розвитку цього напрямку. Проаналізовано практичне застосування технологій розробки ігор. На основі проведеного аналізу було сформульовано основні завдання на дану кваліфікаційну роботу бакалавра.

У другому розділі було проведено детальний аналіз технологій розробки ігор, з особливим акцентом на ігрові рушії. Розроблено базову архітектуру програмного забезпечення з урахуванням особливостей обраного жанру. Обґрунтовано вибір технологічного стеку, зокрема ігрового рушія Unity.

У третьому розділі описано практичну реалізацію гри в жанрі survivor-like на базі обраного ігрового рушія Unity. Представлено процес розробки основних механік гри. Проведене комплексне тестування основних механік гри. Описано процес компіляції виконуваного файлу гри. Було створено інсталяційний пакет. Гру було розміщено на спеціалізованому сервісі.

У висновках до роботи підсумовано реалізовану функціональність гри, обґрунтовано вибір технологій і підтверджено досягнення поставленої мети. Окрему увагу приділено аналізу вимог до системи, а також результатам тестування, які засвідчили стабільну роботу гри, відповідність її логіки обраному жанру та відсутність критичних помилок.

Ключові слова: розробка ігор, survivor-like, ігровий рушій Unity, UnrealEngine, Godot, GameEngine, геймдизайн, компіляція гри, Inno Setup.

ABSTRACT

Davydiuk R. Development of the game "Spooky Scary Skeleton" in the Survivor-like Genre Using Unity. Manuscript. Qualification work for bachelor's degree in Software Engineering, specialty "Software Engineering". Lutsk National Technical University. Lutsk, 2025.Bachelor's qualification thesis in Software Engineering. Lutsk National Technical University, Lutsk, 2025.

The bachelor's thesis consists of an introduction, three chapters, conclusions and recommendations, a list of references, and appendices.

The first chapter provides a comprehensive analysis of the current state of the computer game development industry, in particular the Survivor-like genre. The key trends, successful projects, and prospects for the development of this area are explored. The practical application of game development technologies is analyzed. Based on the analysis, the main tasks for this bachelor's thesis were formulated.

In the second chapter, a detailed analysis of game development technologies was conducted, with a special focus on game engines. The basic architecture of the software was developed taking into account the peculiarities of the chosen genre. The choice of the technology stack, in particular the Unity game engine, is substantiated.

The third chapter describes the practical implementation of a Survivor-like game based on the selected Unity game engine. The process of developing the main game mechanics is presented. The complex testing of the main game mechanics is carried out. The process of compiling the game executable file is described. An installation package was created. The game was placed on a specialized service.

The conclusions to the paper summarize the implemented functionality of the game, justify the choice of technologies, and confirm the achievement of the goal. Particular attention is paid to the analysis of system requirements, as well as to the test results, which showed the stable operation of the game, the correspondence of its logic to the chosen genre and the absence of critical errors.

Keywords: game development, Survivor-like, Unity game engine, UnrealEngine, Godot, GameEngine, game design, game compilation, Inno Setup.

ЗМІСТ

ВСТУП

Актуальність теми. Індустрія відеоігор постійно розвивається, і гравці демонструють стійкий інтерес до проєктів, що пропонують інтенсивний, динамічний і водночас непередбачуваний ігровий досвід. Жанр Survivor-like, з його акцентом на виживанні в умовах постійної загрози, прокачці персонажа під час ігрової сесії та високій реграбельності завдяки елементам випадковості, відповідає цим запитам. Популярність таких ігор, як Vampire Survivors, Brotato та інших, підкреслює актуальність дослідження та розробки в цьому напрямку.

У даній роботі поставлено за мету розробити повноцінну комп’ютерну гру в жанрі Survivor-like. Для досягнення цієї мети було необхідно вирішити низку завдань, включаючи обґрунтування актуальності вибраного жанру, формулювання вимог до гри та проєктування її базового функціоналу, вибір оптимальних інструментів для розробки, безпосередню розробку гри з урахуванням особливостей жанру та, нарешті, компіляцію готового до використання виконуваного файлу.

Об’єктом кваліфікаційної роботи є процес розробки комп’ютерної гри.

Предметом кваліфікаційної роботи є методологія та інструменти розробки ігор у жанрі Survivor-like, включаючи принципи геймдизайну, особливості використання ігрового рушія Unity, а також процес підготовки та компіляції фінального білду гри для подальшого розповсюдження. Засобом досягнення мети є вирішення наступних завдань:

- обґрунтувати актуальність вибраного жанру;
- сформулювати вимоги гри та спроектувати базовий функціонал гри;
- обрати інструменти для розробки гри;
- розробити гру в жанрі Survivor-like;
- скомпілювати виконуваний файл;
- протестувати розроблену гру;
- розмістити гру на спеціалізованому сервісі.

РОЗДІЛ 1
АНАЛІЗ ТЕХНОЛОГІЙ РОЗРОБКИ ВІДЕОІГОР І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

Розробка відеоігор – це багатоетапний процес, який вимагає організації, уяви, досліджень та уваги до деталей.

Індустрія відеоігор на сьогодні є однією з найдинамічніших і найбільш прибуткових галузей цифрового ринку. За останнє десятиріччя спостерігається стрімке зростання популярності інді-розробок та ігор, створених невеликими командами або окремими розробниками. Цьому сприяє поява доступних і потужних рушіїв, таких як Unity (рис. 1.1), що дозволяє реалізовувати складні ідеї навіть без великого бюджету [1].

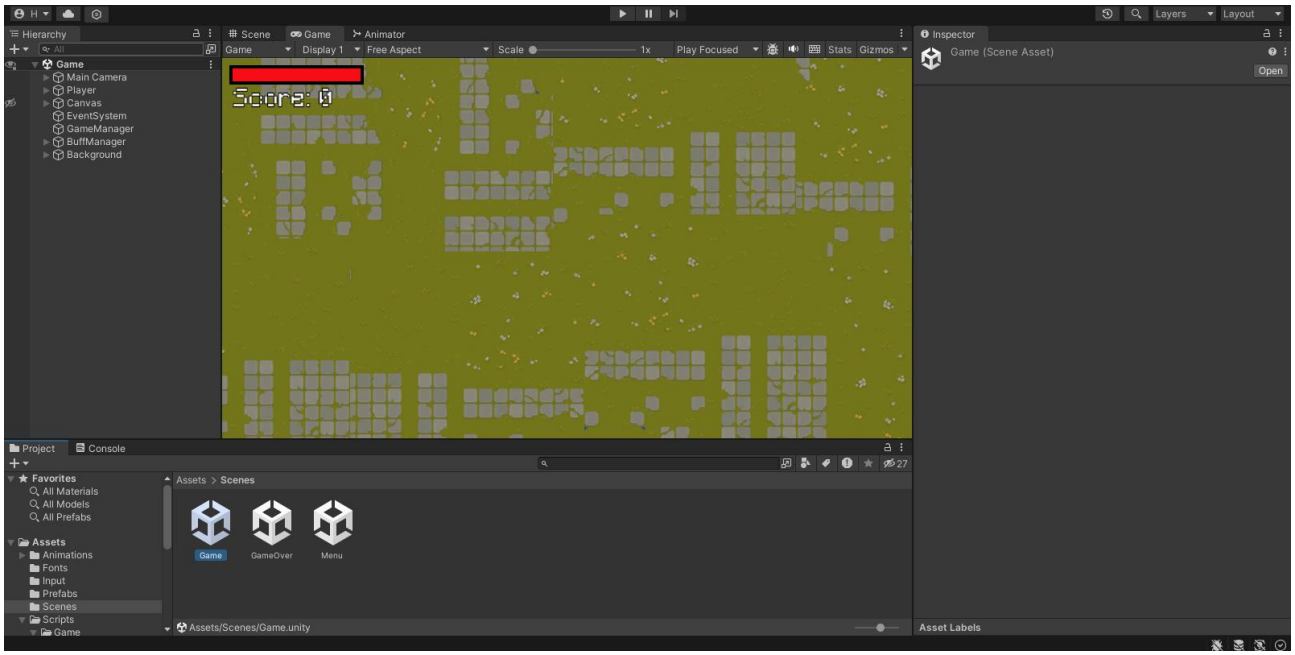

Рисунок 1.1 – Інтерфейс рушія Unity

Жанр Survivor-like є відносно новим піджанром в ігровій індустрії. Його зростання почалося з виходом гри «Vampire Survivors» у 2022 році, яка отримала високу оцінку гравців і критиків завдяки простому, але захоплюючому геймплею, а також елементам rogue-like. Особливістю цього жанру є

автоматична атака персонажа, нескінченні хвилі ворогів, які поступово ускладнюються, і система покрокового покращення характеристик [2].

Ігри у жанрі Survivor-like мають низку характерних особливостей, які вирізняють їх серед інших типів відеоігор і визначають специфіку геймплею.

Однією з ключових рис є автоматичний бій: гравець не керує атаками безпосередньо, а лише контролює пересування персонажа. Такий підхід зосереджує увагу гравця на ухиленні від ворогів, ефективному позиціонуванні на полі бою та прийнятті тактичних рішень.

Другою важливою характеристикою є постійне зростання складності: вороги з’являються у хвилях, з кожною хвилею стаючи все більш численними, швидкими та сильними, що забезпечує поступове підвищення напруження. У грі присутній елемент RPG-прогресії, який виражається у розвитку персонажа: знищення ворогів приносить досвід, що дозволяє відкривати нові покращення або посилювати вже наявні здібності.

Ще однією особливістю є обмежена тривалість ігрової сесії – зазвичай одна гра триває фіксований час або доти, доки персонаж не загине, що робить ігри цього жанру придатними як для швидких сеансів, так і для багаторазового проходження.

Жанр Survivor-like часто порівнюють із такими жанрами, як bullet hell, roguelike та twin-stick shooter. Попри певні спільні риси, Survivor-like має свої унікальні особливості. Наприклад, на відміну від roguelike-ігор, де значну роль відіграє генерація лабіринтів та ресурсоменеджмент, у Survivor-like основний акцент зміщено на динаміку бою та виживання в умовах постійного натиску. У порівнянні з bullet hell, у якому гравець уникає великої кількості снарядів, Survivor-like частіше покладається на щільність ворогів та прогресивну ескалацію небезпеки. Twin-stick шутери, хоча і мають схожий формат керування, зазвичай потребують ручної атаки, тоді як у Survivor-like атаки автоматизовані. Такий підхід знижує поріг входу для новачків і робить гру зручнішою для гри на мобільних пристроях.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Сформовано мету роботи, якою є розробка гри в жанрі Survivor-like з допомогою рушія Unity.

Відповідно до мети, визначено завдання, які виконуватимуться під час роботи над проєктом кваліфікаційної роботи бакалавра:

- обґрунтувати актуальність вибраного жанру;
- сформулювати вимоги гри та спроектувати базовий функціонал гри;
- обрати інструменти для розробки гри;
- розробити гру в жанрі Survivor-like;
- скомпілювати виконуваний файл;
- протестувати розроблену гру;
- розмістити гру на спеціалізованому сервісі.

Висновки до розділу 1

У першому розділі було здійснено аналіз сучасного стану розробки ігор у жанрі Survivor-like, а також розглянуто можливості використання рушія Unity для реалізації подібних проєктів. Визначено ключові риси жанру, його особливості та переваги.

Ігри жанру Survivor-like пропонують інтенсивний геймплей із поступовим зростанням складності, що поєднує елементи виживання, динамічної стратегії та RPG. Популярність цього жанру зумовлена доступністю, реіграбельністю та простотою освоєння. Застосування Unity як основного інструменту розробки надає широкі можливості завдяки кросплатформеності, розвиненій екосистемі та підтримці 2D/3D графіки.

Завдяки розвитку технологій і широкому доступу до потужних рушіїв, створення подібних ігор стало реальним навіть для невеликих команд. Це зумовило стрімке зростання кількості проєктів у цьому жанрі та потребу в якісному технічному рішенні для реалізації унікальних ідей.

На основі проведеного аналізу було визначено мету даної роботи – розробити повноцінну гру в жанрі Survivor-like з використанням Unity. Сформульовано завдання для подальших етапів розробки, включно з визначенням концепції гри, вибором інструментів, реалізацією ключових механік, розробкою візуального оформлення, а також тестуванням ігрового процесу.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваної гри, та її проектування

Перед розробкою будь-якого ігрового проєкту важливо сформулювати його концепцію та зрозуміти, яким чином гра буде відповідати очікуванням користувачів. У випадку з проєктом у жанрі Survivor-like, основний акцент робиться на інтенсивному геймплеї з нескінченними хвилями ворогів і поступовим розвитком персонажа. Гравець керує персонажем, що має вижити в умовах постійного натиску з боку противників. Водночас, на відміну від класичних action-ігор, у Survivor-like (рис. 2.1) атаки персонажа виконуються автоматично, а увага гравця зосереджується переважно на пересуванні, ухиленні від ворогів та виборі покращень після підвищення рівня [4].

Рисунок 2.1 – Гра Vampire Survivors [5]

Протягом ігрової сесії гравець набирає досвід, знищуючи ворогів, і кожне нове підвищення рівня відкриває доступ до нових навичок або покращень (рис. 2.2). Такі елементи дають змогу створити цікавий цикл розвитку, в якому кожна нова гра може відрізнятися за стилем – залежно від вибраних комбінацій покращень. Особливої уваги потребує баланс складності: із часом гра повинна ускладнюватися, вороги – ставати швидшими та сильнішими, а на певних етапах – з'являтися боси, що виконують роль перевірки навичок гравця.

Таким чином, концепція гри базується на комбінації простого керування, постійного розвитку персонажа та зростання виклику, що робить кожну сесію унікальною й потенційно реграбельною, також конструкція системи забезпечує можливості для розширення в майбутньому [6].

Рисунок 2.2 – Вікно підвищення рівня в грі Vampire Survivors [7]

Функціональні вимоги формуються відповідно до жанрових особливостей, досвіду аналогічних ігор і очікувань аудиторії. Гравець повинен мати змогу керувати персонажем у реальному часі, використовуючи стандартне для ПК

керування клавішами WASD. При цьому атаки персонажа відбуваються автоматично з певною періодичністю, а їхня частота, сила та форма залежать від поточного набору активних покращень.

Після вбивства ворога гравець отримує певну кількість досвіду. Накопичення певної кількості досвіду підвищує рівень персонажа, що відкриває вікно вибору покращення – нової здатності або посилення існуючих характеристик. Це можуть бути додаткові снаряди, збільшення радіусу шкоди, скорочення інтервалу між пострілами, тощо. Гравець приймає рішення, яке безпосередньо впливатиме на стратегію виживання в подальшому.

Протягом гри на екрані з’являються вороги, кількість і складність яких зростає з часом. Їхня генерація відбувається хвилями, або ж безперервно із поступовим ускладненням. Кожні кілька хвилин або при досягненні певного рівня з’являється особливо сильний супротивник – бос. Це ворог із підвищеним рівнем здоров’я, посиленим розміром та можливо – унікальними здібностями. Перемога над босом дає додаткову нагороду і відкриває шлях до наступного етапу гри [8].

Серед інших важливих функцій варто виділити систему здоров’я персонажа. Гравець починає з фіксованим запасом HP, який зменшується при зіткненні з ворогами. Після вичерпання здоров’я гра завершується. Після поразки відображається фінальний результат – наприклад, час виживання, досягнутий рівень або кількість знищених ворогів.

Структура проєкту має передбачати можливість розширення: додавання нових типів ворогів, зброї, покращень, механік без потреби у глибокому переписуванні базового коду. Це досягається шляхом правильного розділення логіки на окремі класи, використання інтерфейсів та шаблонів проєктування.

Крім того, інтерфейс гри має бути інтуїтивно зрозумілим та зручним. Гравець повинен з перших секунд орієнтуватися у всіх елементах керування та візуальній інформації на екрані. Важливо дотримуватись сучасних принципів UI/UX дизайну: використання контрастних кольорів, зручне розташування панелей, відсутність зайвих елементів, які відволікають увагу [9].

Для реалізації гри буде використано об’єктно-орієнтований підхід, що передбачає логічне розділення обов’язків між компонентами та повторне використання коду (рис. 2.3). Основними складовими ігрового проєкту є: система керування гравцем, система стрільби, логіка ворогів, менеджер рівня, система прокачування, спавнер ворогів і базовий GameManager, який координує всі процеси гри [10].

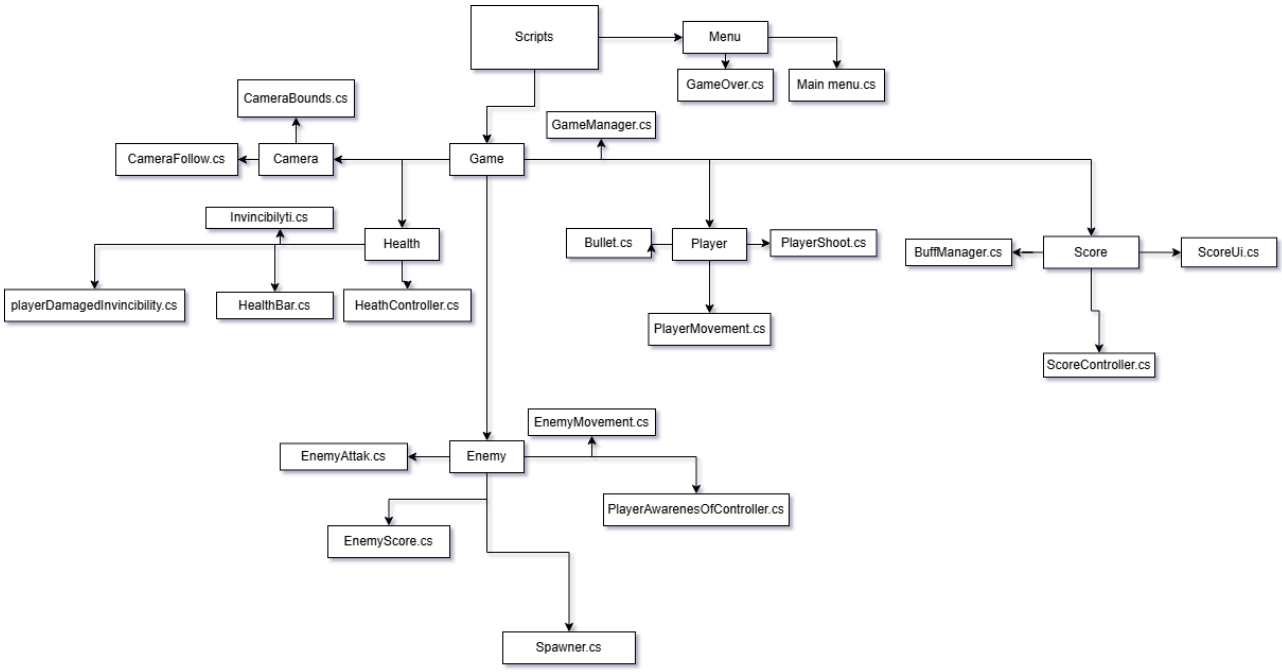

Рисунок 2.3 – Ієрархія внутрішніх папок гри

Персонаж гравця реалізується через окремий скрипт, що відповідає за пересування у двовимірному просторі та отримання шкоди. Стрілянина реалізована у вигляді компонента, що генерує снаряди із заданим інтервалом. Ці снаряди взаємодіють із ворогами через колізії та передають їм інформацію про нанесену шкоду. Вороги реалізуються як окремі об’єкти з власними параметрами: здоров’ям, швидкістю, типом поведінки. Боси, у свою чергу, є підкласом ворогів із масштабованими характеристиками.

Всі події в грі координуються через GameManager – центральний клас, що ініціалізує гру, обробляє стан Game Over, керує потоком хвиль ворогів, а також взаємодіє з UI. Інтерфейс реалізується окремим менеджером, що відслідковує

зміни у здоров'ї, досвіді, рівні та часу. Комунікація між елементами гри відбувається через події або патерн Observer, що дозволяє динамічно оновлювати дані без жорстких залежностей [11].

Загальна структура сцени передбачає наявність окремої головної сцени гри, стартового меню та екрана завершення. Це дозволяє логічно відділити різні стани гри та реалізувати плавні переходи між ними.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Вибір правильного ігрового рушія має вирішальне значення для будь-якого розробника ігор, незалежно від його досвіду. З такою великою кількістю доступних варіантів, кожен з яких має свої сильні та слабкі сторони, важливо розуміти, що робить кожен рушій унікальним.

Ігровий рушій – це програмний фреймворк для створення та розробки відеоігор. Він надає набір інструментів і бібліотек, які виконують типові завдання розробки ігор, такі як рендеринг графіки, моделювання фізики, керування звуком тощо. Використовуючи ігровий рушій, розробники можуть зосередитися на створенні унікальних аспектів своєї гри, а не винаходити велосипед для цих базових елементів.

Було розглянуто 4 рушія для використання у даному проєкті, а саме Unity, Unreal Engine, Godot, та GameMaker.

Unity (табл. 2.1) став надійним помічником в індустрії розробки ігор, відомий своєю універсальністю та простотою використання. Він підходить для розробки як 2D, так і 3D ігор, що робить його привабливим вибором для широкого спектру проектів. Широке сховище ресурсів Unity та спільнота підтримки сприяють його популярності, а можливості крос-платформного розгортання підвищують його доступність [12].

Таблиця 2.1 – Переваги та недоліки рушія Unity

Переваги	Недоліки
Зручність	Графічні обмеження
Кросс платформенність	Проблеми з продуктивністю
Мова програмування: C#	Фрагментація між версіями
Розвинена спільнота	

Розроблений Epic Games, рушій Unreal Engine (табл. 2.2) є титаном індустрії, відомим своєю приголомшливою графікою та реалістичними візуальними ефектами. Він ідеально підходить для розробки AAA-ігор, завдяки потужному рушію рендерингу та інтуїтивно зрозумілій системі візуальних сценаріїв Blueprints. Майстерність Unreal Engine полягає в його здатності створювати першокласну графіку та кінематографічні ефекти [13].

Таблиця 2.2 – Переваги та недоліки рушія Unreal Engine

Переваги	Недоліки
Високоякісна графіка	Крута крива навчання
Візуальний скрипт Blueprint	Ресурсомісткість
Велика спільнота та ресурси	Повільніший час ітерацій
Масштабованість та продуктивність	Менш гнучкий для 2D ігор
Підтримка технологій наступного покоління	

Godot (табл. 2.3) здобув популярність як ігровий рушій з відкритим вихідним кодом, що поєднує в собі простоту та універсальність. Godot підтримує розробку як 2D, так і 3D ігор, має візуальну систему написання сценаріїв, а сайт заохочує до співпраці спільноти. Завдяки відсутності ліцензійних платежів та активній спільноті, він став привабливим вибором для інді-розробників [14].

Таблиця 2.3 – Переваги та недоліки рушія Godot

Переваги	Недоліки
Відкритий вихідний код	Невелика спільнота
Повністю безкоштовний	Менша функціональність
Підтримка багатьох мов програмування	Невелика інтегрованість в індустрію
Окремі версії рушія для 2D та 3D розробки	

GameMakerStudio (табл. 2.4) – це 2D-орієнтований ігровий рушій, відомий своєю простотою та легкістю у використанні. Він особливо популярний серед інді-розробників та аматорів створення піксельних ігор і використовувався для розробки таких ігор, як Undertale, Spelunky, Hyper Light Drifter, Hotline Miami та Katana ZERO [14].

Таблиця 2.4 – Переваги та недоліки рушія GameMaker

Переваги	Недоліки
Зручність	Обмежені 3D можливості
2D - спеціалізація	Плата за ліцензію
Активна спільнота	Продуктивність
Крос-платформний експорт	Мова використання - GML
	Немає вбудованої підтримки розширеної багатокористувацької мережі

Для розробки застосунку, який дозволятиме взаємодіяти з комп’ютером за допомогою жестів, можна використовувати різні мови програмування і відповідні їм бібліотеки. Щоб визначити, які інструменти є найбільш оптимальними та ефективними для обробки й розпізнавання жестів, проведено огляд кількох таких мов, що дозволяють реалізувати дане програмне рішення.

Висновки до розділу 2

На основі аналізу можливих платформ і засобів розробки для створення гри в жанрі Survivor-like, обрано рушій Unity, який надає широкі можливості для реалізації 2D-графіки, підтримує масштабовану архітектуру проєкту та забезпечує кросплатформенність. Середовище розробки Visual Studio було використано для написання коду на мові C#, яка є основною для Unity і дозволяє ефективно структурувати логіку гри, використовуючи об’єктно-орієнтований підхід і патерни проєктування.

Для побудови логіки ігрового процесу обрано алгоритми, які відповідають жанру: спавн ворогів з поступовим ускладненням, накопичення досвіду та вибір

покращень гравцем. Всі ці елементи організовано у вигляді модулів із чіткими відповідальностями: окремо для гравця, ворогів, менеджера гри та UI. В основу побудови взаємодії між компонентами покладено подієву систему, що підвищує гнучкість і розширюваність коду.

Порівнявши популярні рушії (Unity, Unreal Engine, Godot, GameMaker Studio), встановлено, що Unity є найбільш оптимальним вибором завдяки зручності для 2D-розробки, великій кількості документації, активній спільноті та підтримці основних платформ.

Таким чином, обраний набір технологій, методів і інструментів дозволяє ефективно реалізувати функціональність, притаманну сучасним іграм у жанрі Survivor-like, із забезпеченням гнучкості подальшого розширення, оптимізації та виводу продукту на різні платформи.

РОЗДІЛ 3
РОЗРОБКА ГРИ В ЖАНРІ SURVIVOR-LIKE З ДОПОМОГОЮ РУШІЯ UNITY

3.1 Практична реалізація об’єкта проєктування

У процесі розробки гри жанру Survivor-like було використано ігровий рушій Unity, який надає потужний набір інструментів для створення інтерактивного тривимірного середовища. Основним середовищем розробки обрано Unity Editor версії 2022.3 LTS, а як інтегроване середовище для редагування коду – Visual Studio 2022, з вбудованим підтриманням мови C# [16].

Перед початком безпосередньої розробки було встановлено необхідні компоненти Unity Editor, налаштовано платформу розгортання та створено базову структуру проєкту. Робота над проєктом розпочалася зі створення сцени меню та основної ігрової сцени, що зображено на рисунку 3.1.

Рисунок 3.1 – Початковий екран гри Spooky scary skeleton

На головному екрані створено кнопки «Play» та «Exit» що відповідають за запуск гри та її закриття відповідно.

Логіка керування кнопками наведена в лістингу 3.1.

Лістинг 3.1 – Контролер керування кнопок

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

public class Mainmenu : MonoBehaviour
{
    public void Play()
    {
        SceneManager.LoadScene(«Game»);
    }

    public void Main_menu()
    {
        SceneManager.LoadScene(«Menu»);
    }

    public void Exit()
    {
        Application.Quit();
    }
}
```

кінець лістингу 3.1

Далі було створено гравця (рис. 3.2) і прописано його логіку пересування та дій. Цей скрипт реалізував систему руху гравця у 2D-просторі з урахуванням обмежень по координатах, згладженого введення та інтеграції з анімацією. Він використовує компоненти «Rigidbody2D» для фізичного пересування та «Animator» для візуального відображення стану руху. Гравець може рухатися в межах заданих координат, що запобігає виходу за межі ігрового простору.

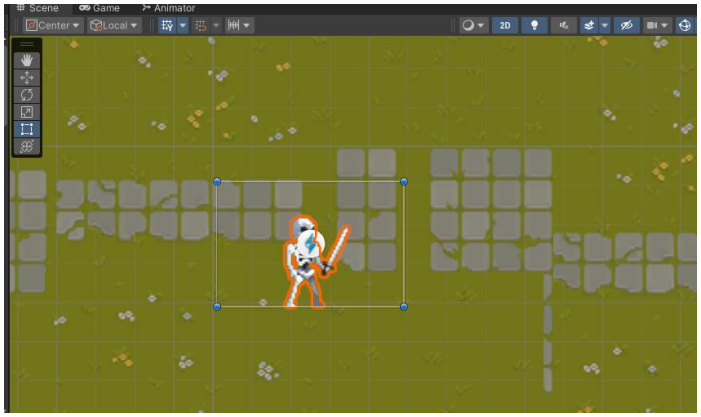

Рисунок 3.2 – Спрайт гравця в редакторі Unity

Крім того, скрипт автоматично змінює орієнтацію персонажа в залежності від напрямку руху, що створює більш природне візуальне враження. Для зчитування введення використовується нова система вводу Unity. яка дозволяє більш гнучко обробляти дії гравця (додаток А).

Префаби дозволяють зберегти конфігурацію об’єкта та легко створювати його копії. Також було створено логіку поведінки кулі (ліст. 3.2).

Лістинг 3.2 – Логіка поведінки кулі

```
using UnityEngine;

public class Bullet : MonoBehaviour
{
    [SerializeField] private float _lifetime = 3f;
    [SerializeField] private float _damage = 10f;
    private void Start()
    {
        Destroy(gameObject, _lifetime);
    }
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag(«Player»)) return;
        if (other.CompareTag(«Enemy»))
        {
            if (other.TryGetComponent<HeathController>(out var health))
            {
                health.TakeDamage(_damage);
            }
            Destroy(gameObject);
        }
    }
}
```

кінець лістингу 3.2

Було створено префаб кулі (рис. 3.3). Префаб – це шаблон або заготовка ігрового об’єкта, яку можна створити один раз і потім багаторазово використовувати у сценах. Префаби є одним із основних інструментів Unity для оптимізації, організації та масштабування процесу розробки гри. Вони зберігають заздалегідь налаштований об’єкт із усіма його властивостями, компонентами та вкладеною ієрархією.

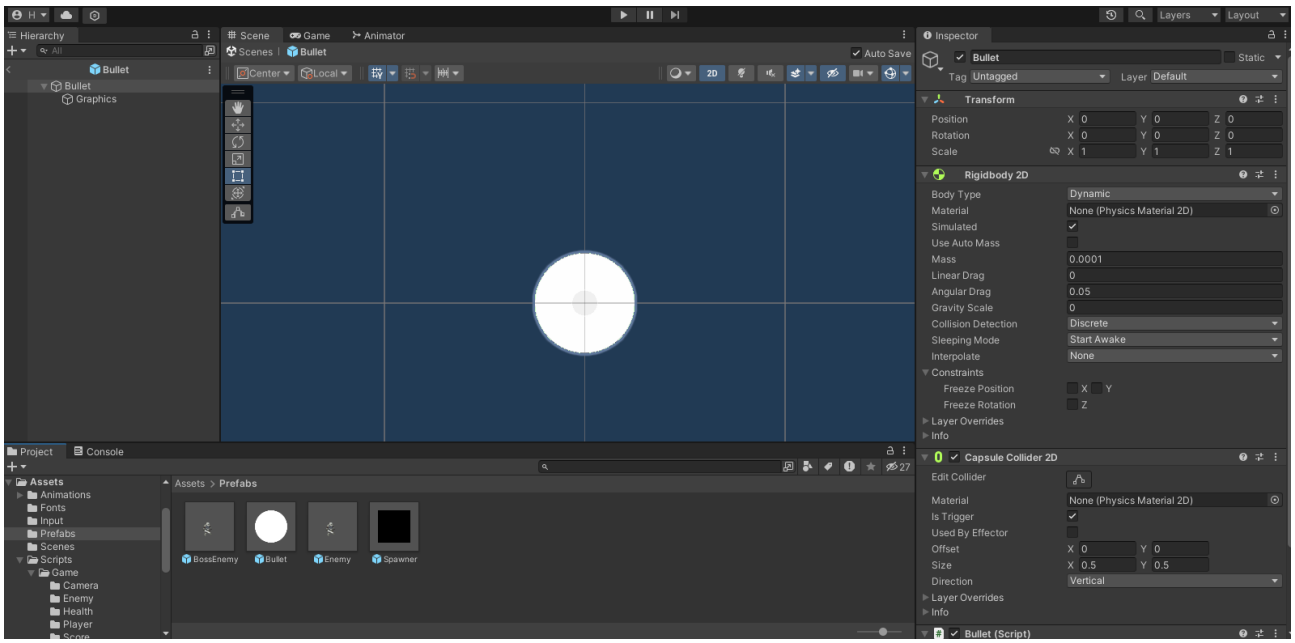

Рисунок 3.3 – Префаб кулі в редакторі Unity

Було створено скрипт що реалізує механізм тимчасової невразливості гравця після отримання ним пошкодження або в інших випадках, що вимагають захисту персонажа від втрати здоров’я протягом певного проміжку часу. Такий підхід є особливо актуальним у вибраному жанрі, де гравець може зазнати миттєвих або частих атак, де гравець має отримати можливість на короткий час уникати нових пошкоджень, щоб не втратити всі очки здоров’я одразу після одного невдалого зіткнення з ворогом чи перешкодою.

Скрипт взаємодіє з іншим компонентом під назвою «HeathController», який відповідає за загальне керування здоров’ям персонажа, зокрема за збереження його поточного стану, зменшення при пошкодженнях та, в даному випадку, активацію стану невразливості.

Така реалізація дозволяє досягти балансу між складністю гри та комфортом гравця, даючи останньому короткочасну можливість оговтатись після критичної ситуації або уникнути миттєвого завершення гри в результаті непередбачуваних атак (ліст. 3.3).

Лістинг 3.3 – Скрипт для вводу гравця у стан невразливості у файлі «Inivicibility.cs»

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

public class Inivicibility : MonoBehaviour
{
    private HeathController _healthController;
    private void Awake()
    {
        _healthController = GetComponent<HeathController>();
    }
    public void startInvicibile(float invincibilityDuration)
    {
        StartCoroutine(InvincCour(invincibilityDuration));
    }
    private IEnumerator InvincCour(float invincibilityDuration)
    {
        _healthController.isInvicible = true;
        yield return new WaitForSeconds(invincibilityDuration);
        _healthController.isInvicible = false;
    }
}
```

кінець лістингу 3.3

Створено скрипт призначений для візуального відображення рівня здоров’я персонажа на інтерфейсі гри (ліст. 3.4). Він оновлює заповнення графічного елементу смуги здоров’я відповідно до поточного відсотка здоров’я, який надає компонент «HeathController». Таким чином, гравець отримує наочну інформацію про стан персонажа під час гри.

Лістинг 3.4 – Скрипт для виводу кількості здоров’я гравця «healthbar.cs»

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

public class HealthBar : MonoBehaviour
{
    [SerializeField]
    private UnityEngine.UI.Image _healthFG;

    public void barUpdate(HeathController HealthConrtroller)
```

```
    {
        _healthFG.fillAmount = HealthConrtroller.RemainHealthPercent;
    }
}
```

кінець лістингу 3.4

Одним із ключових елементів цієї системи стала реалізація механізму виявлення гравця, за допомогою якого ворог отримує інформацію про присутність гравця поблизу (ліст. 3.5). Вона забезпечує базовий рівень обізнаності ворога про навколишнє середовище та дозволяє визначати напрямок до гравця і відстань до нього, що в подальшому використовується для прийняття рішень про переслідування чи атаку.

Для забезпечення динамічності та природності поведінки ворога у просторі сцени, окремо був створений скрипт, який відповідає за його рух у напрямку до гравця (додаток Б), дозволяє ворогові активно змінювати своє положення у просторі, реагуючи на зміну позиції гравця, що в сукупності з системою виявлення формує основу інтелектуального переслідування.

Лістинг 3.5 – Логіка пошуку гравця ворогами у файлі «PlayerAwareness.cs»

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

public class PlayerAwarenesOfController : MonoBehaviour
{
        public bool AwareOfPlayer { get; private set; }

    public Vector2 DirectionToPlayer { get; private set; }
    [SerializeField]
    private float _playerAwarenessDistance;
    private Transform _player;
    private void Awake()
    {
        var playerMovement = FindObjectOfType<PlayerMovement>();
        if (playerMovement != null)
        {
            _player = playerMovement.transform;
        }
    }
    // Update is called once per frame
```

```
    void Update()
    {
        if (_player == null)
            return;
        Vector2     enemyToPlayerVector     =     _player.position     -
transform.position;
        DirectionToPlayer = enemyToPlayerVector.normalized;

        AwareOfPlayer      =      enemyToPlayerVector.magnitude      <=
_playerAwarenessDistance;
    }
    void Start()
    {
        this.enabled = true; // гарантія увімкнення
    }

}
```

кінець лістингу 3.5

Для реалізації механіки безперервного зростання складності гри та створення враження постійного тиску з боку противників було розроблено спеціальний об’єкт, відповідальний за генерацію хвиль ворогів (додаток В).

Однією з ключових особливостей цього об’єкта є його здатність переміщуватись синхронно з камерою гравця.

Також було реалізовано логіку взаємодії з гравцем (ліст. 3.6). Вона охоплює обробку ситуацій зіткнення або наближення до гравця на критичну дистанцію, після чого ворог здатен нанести шкоду. Таким чином, було закладено фундамент для побудови базової бойової системи з боку супротивника.

Лістинг 3.6 – Скрипт з описаною логікою взаємодії гравця і ворога в файлі «EnemyAttack.cs»

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

public class EnemyAttak : MonoBehaviour
{
    [SerializeField]
    private float _damageAmount;

    private void OnCollisionStay2D(Collision2D collision)
```

```
    {
        if (collision.gameObject.GetComponent<PlayerMovement>())
        {
            var                          healthController                          =
collision.gameObject.GetComponent<HeathController>();
            healthController.TakeDamage(_damageAmount);
        };
    }
}
```

кінець лістингу 3.6

Такий підхід дозволяє підтримувати тиск на гравця незалежно від його пересування і гарантує, що хвилі ворогів не залишаються поза межами видимості або активної частини сцени. Це рішення особливо ефективне для ігор з нескінченним або поступово розширюваним ігровим простором, де рух гравця не обмежений конкретною зоною.

У процесі реалізації штучного інтелекту противників у грі було створено повноцінний префаб ворога (рис. 3.4), що містить у собі всі необхідні компоненти для його функціонування: візуальне представлення, фізичні властивості, а також скрипти, що визначають його поведінку.

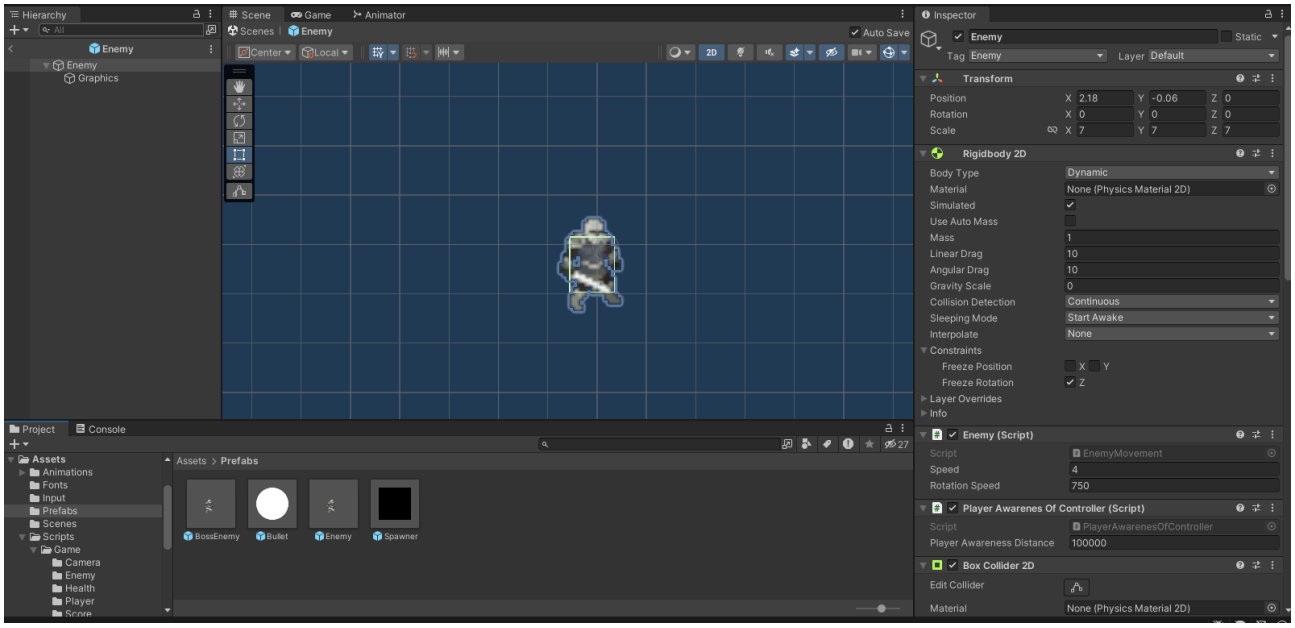

Рисунок 3.4 – Префаб ворога в редакторі Unity

Окрім цього, була реалізована система нарахування очок за знищення кожного ворога. Це забезпечує зворотний зв’язок для гравця, підсилює мотивацію до знищення супротивників та є частиною загального ігрового процесу. Кількість очок, яку отримує гравець за кожного переможеного ворога, регулюється програмно, і приклад відповідної реалізації наведено у лістингу 3.7. Цей механізм може бути легко адаптований для подальшого розширення, наприклад, для врахування рівня складності ворога або поточного прогресу гравця в грі.

Лістинг 3.7 – Логіка підрахунку кількості очок

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

public class EnemyScore : MonoBehaviour
{

    [SerializeField]
    private int _killScore;

    private ScoreController _scoreController;

    private void Awake()
    {
        _scoreController = FindObjectOfType<ScoreController>();
    }

    public void allocateScore()
    {
        Debug.Log($»Додано {_killScore} очок»);
        _scoreController.AddScore(_killScore);
    }

}
```

кінець лістингу 3.7

У межах розробки ігрового середовища була вручну створена велика карта, (рис. 3.5) яка стала основною ареною для ігрового процесу. Для її побудови активно використовувались безкоштовні ассети, що дозволило значно пришвидшити процес розробки та зосередитися на створенні зручного й

візуально привабливого простору для гравця. Завдяки масштабності карти гравець отримує можливість вільно пересуватись, уникати зіткнень з ворогами, обирати оптимальні маршрути та застосовувати тактичні рішення залежно від ситуації.

Таке розширене ігрове поле не лише підвищує загальну динаміку гри, а й сприяє кращому зануренню у геймплей, створюючи враження відкритого світу, навіть за умови використання обмеженого ігрового простору.

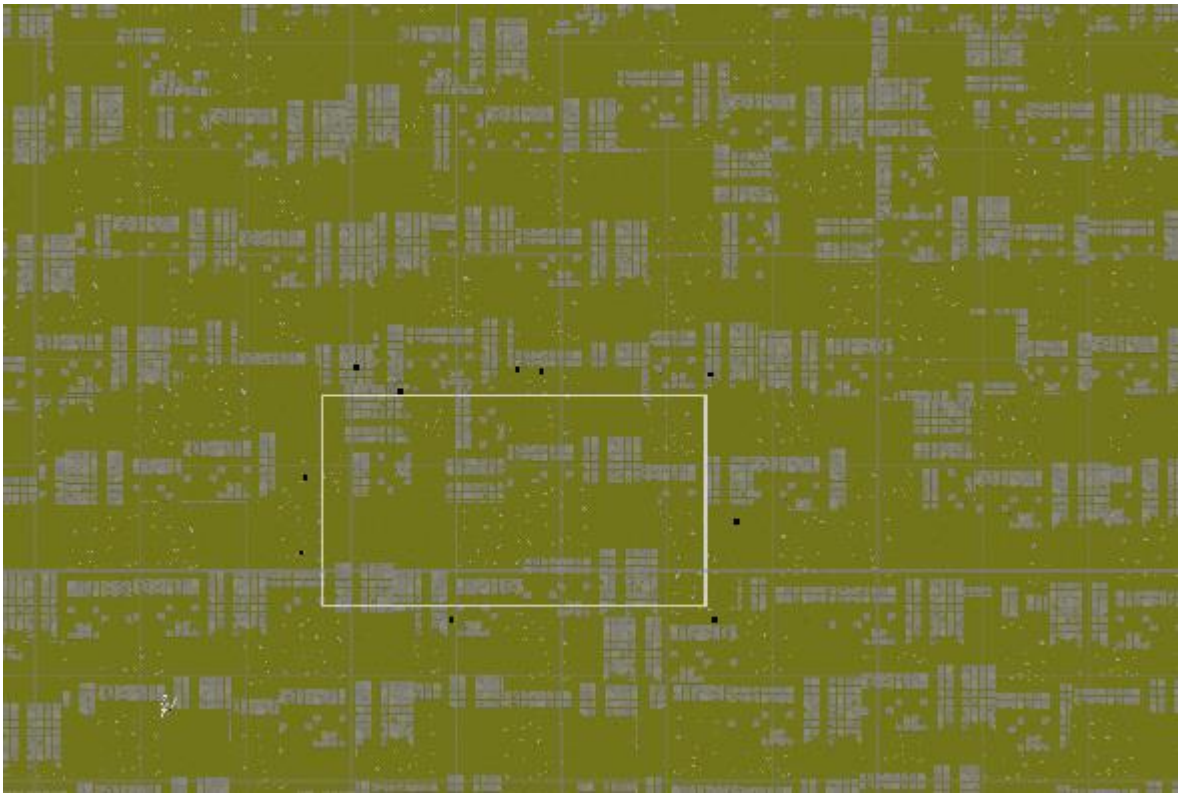

Рисунок 3.5 – Карта світу в редакторі Unity

У зв’язку з масштабністю створеної ігрової карти та необхідністю забезпечити гравцеві постійний контроль над ситуацією навколо, було створено окремий скрипт, який відповідає за фокусування камери на персонажі гравця. Його основне завдання полягає в тому, щоб камера автоматично слідувала за гравцем, утримуючи його в центрі кадру незалежно від напрямку руху чи швидкості пересування. Це дозволяє не лише покращити орієнтацію в просторі, а й гарантує, що гравець завжди перебуває у полі зору, що є критично важливим для ефективного керування, ухилення від ворогів і планування подальших дій.

Така реалізація камери значно підвищує комфорт гри, особливо в умовах динамічного геймплею (ліст. 3.8).

Лістинг 3.8 – Конфігурація гучності у файлі «volume_control.py»

```
using UnityEngine;

public class CameraFollow : MonoBehaviour
{
    [SerializeField] private Transform _target; // гравець
    [SerializeField] private Vector2 _minBounds;
    [SerializeField] private Vector2 _maxBounds;

    private float _halfHeight;
    private float _halfWidth;

    private Camera _cam;

    private void Start()
    {
        _cam = Camera.main;
        _halfHeight = _cam.orthographicSize;
        _halfWidth = _halfHeight * _cam.aspect;
    }

    private void LateUpdate()
    {
        if (_target == null) return;

        float clampedX = Mathf.Clamp(_target.position.x, _minBounds.x +
_halfWidth, _maxBounds.x – _halfWidth);
        float clampedY = Mathf.Clamp(_target.position.y, _minBounds.y +
_halfHeight, _maxBounds.y – _halfHeight);

        transform.position    =    new    Vector3(clampedX,    clampedY,
transform.position.z);
    }
}
```

кінець лістингу 3.8

У межах розвитку ігрової механіки було реалізовано систему рівнів, яка дозволяє гравцеві поступово покращувати характеристики свого персонажа. З кожним підвищенням рівня, що досягається шляхом накопичення досвіду, гравцю надається можливість обрати одне з кількох доступних покращень.

Такий підхід дозволяє формувати індивідуальний стиль проходження, надаючи гравцеві свободу у виборі стратегії та підвищуючи загальну варіативність ігрового процесу. Система рівнів виконує також важливу мотиваційну функцію, заохочуючи гравця до активних дій у грі та забезпечуючи відчутний прогрес у розвитку персонажа (додаток Г).

Коли гравець програє або завершує гру, завантажується сцена GameOver (рис. 3.6). Вона показує, що гра завершена, дає змогу почати заново або повернутися в меню чи вийти з гри.

Рисунок 3.6 – Сцена «Game Over» в редакторі Unity

3.2 Тестування та налагодження інформаційно-комп’ютерної системи

Після завершення основного етапу розробки гри надзвичайно важливо приділити увагу процесу її тестування. Це завершальний, проте критично важливий етап, який забезпечує належну якість програмного продукту та дозволяє уникнути багатьох потенційних проблем у подальшому. У сфері геймдизайну та розробки інтерактивних застосунків якісне тестування визначає не лише стабільність функціонування гри, але й враження користувача, його задоволення від процесу гри та загальний рівень довіри до продукту.

Гра, навіть якщо вона виглядає візуально привабливо і має цікаву механіку, не може вважатися завершеною без проходження повноцінного етапу тестування. Саме завдяки ретельній перевірці вдається знайти як явні, так і приховані помилки, які могли бути допущені у процесі створення логіки, написання коду або при інтеграції різних компонентів. Важливо також враховувати, що деякі помилки можуть не проявлятися відразу, а виникати лише за певних умов, що робить процес тестування ще більш відповідальним і багатогранним.

У процесі перевірки гри були проаналізовані всі її ключові елементи. Особливу увагу було приділено базовій функціональності, а саме: чи може гравець належним чином пересуватись по ігровому полю, чи відбувається стрільба так, як передбачено задумом, чи вірно враховується шкала здоров’я ворогів, і чи з’являються боси на відповідних рівнях. Не менш важливим було й те, як працює система нарахування очок, адже саме на її основі реалізується підвищення рівня гравця.

Крім цього, було проведено оцінку інтерфейсу користувача. Окремо тестувалися меню, кнопки, відображення текстових повідомлень, а також коректність перемикання сцен. Було необхідно впевнитися, що жоден елемент інтерфейсу не виходить за межі екрану та не перекриває важливі ділянки ігрового поля. Також здійснювалась перевірка реакції гри на натискання кнопок при різних роздільних здатностях екрана, щоб уникнути графічних артефактів або неправильного масштабування.

Не менш важливою частиною тестування стало виявлення проблем продуктивності. Навіть найкраща гра може стати непридатною до використання, якщо вона надмірно навантажує систему або знижує кількість кадрів на секунду. У зв’язку з цим було змодельовано ситуації, коли на екрані одночасно з’являється велика кількість ворогів, снарядів та візуальних ефектів. У результаті вдалося виявити ряд критичних місць, які були своєчасно оптимізовані, зокрема заміна надлишкового створення об’єктів на систему об’єктного пулінгу, яка дозволяє повторно використовувати вже створені об’єкти без зайвих витрат

ресурсів.

Загалом, у процесі тестування були виявлені та виправлені наступні категорії несправностей:

– логічні помилки в ігровому коді (наприклад, некоректна поява босів або неправильне нарахування очок);

– графічні проблеми інтерфейсу (зміщення тексту, неправильне масштабування на деяких пристроях);

– функціональні несправності (стрільба під час паузи, відсутність знищення об’єктів після досягнення певної межі);

– продуктивні збої (зависання або різке зниження FPS при великій кількості об’єктів на сцені).

Варто зазначити, що тестування гри не обмежувалося лише ручним переглядом та проходженням. В окремих випадках застосовувалися також базові елементи автоматизованого тестування, зокрема для перевірки функцій, які відповідають за підрахунок очок та підвищення рівня. Це дозволило впевнитися, що навіть після багаторазових змін у коді ці ключові механізми зберігають свою працездатність.

Таким чином, проведене тестування дозволило досягти високого рівня якості гри, усунути помилки, що могли негативно вплинути на досвід користувача, та створити стабільний ігровий процес, який відповідає початковим очікуванням і поставленим технічним завданням. Це ще раз підкреслює, що завершення розробки будь-якого програмного продукту неможливе без ретельного й багаторівневого тестування, яке є невіддільною складовою сучасної практики розробки ігор.

3.3 Розробка виконуваного файлу та інсталяційного пакета

Налаштування фінального білду гри перед компіляцією є ключовим етапом у процесі розробки проєкту в Unity, оскільки саме ці параметри визначають, як саме гра буде підготовлена до розповсюдження, тестування або

випуску. У цьому розділі детально розглянуто процес налаштування білду гри, виходячи з конкретного прикладу, зафіксованого на скріншотах, що демонструють типову конфігурацію Unity-проєкту для платформи Windows.

Насамперед варто звернути увагу на вікно Build Settings (рис. 3.7) у якому вибрано сцени, що включатимуться до фінального білду. Додано три сцени: «Menu», «Game» і «GameOver», що є типовим розділенням для більшості ігор, які мають головне меню, основний ігровий процес і фінальний екран завершення. Це дозволяє забезпечити структурованість і гнучкість проєкту, а також зручно керувати переходами між сценами за допомогою системи завантаження сцен Unity.

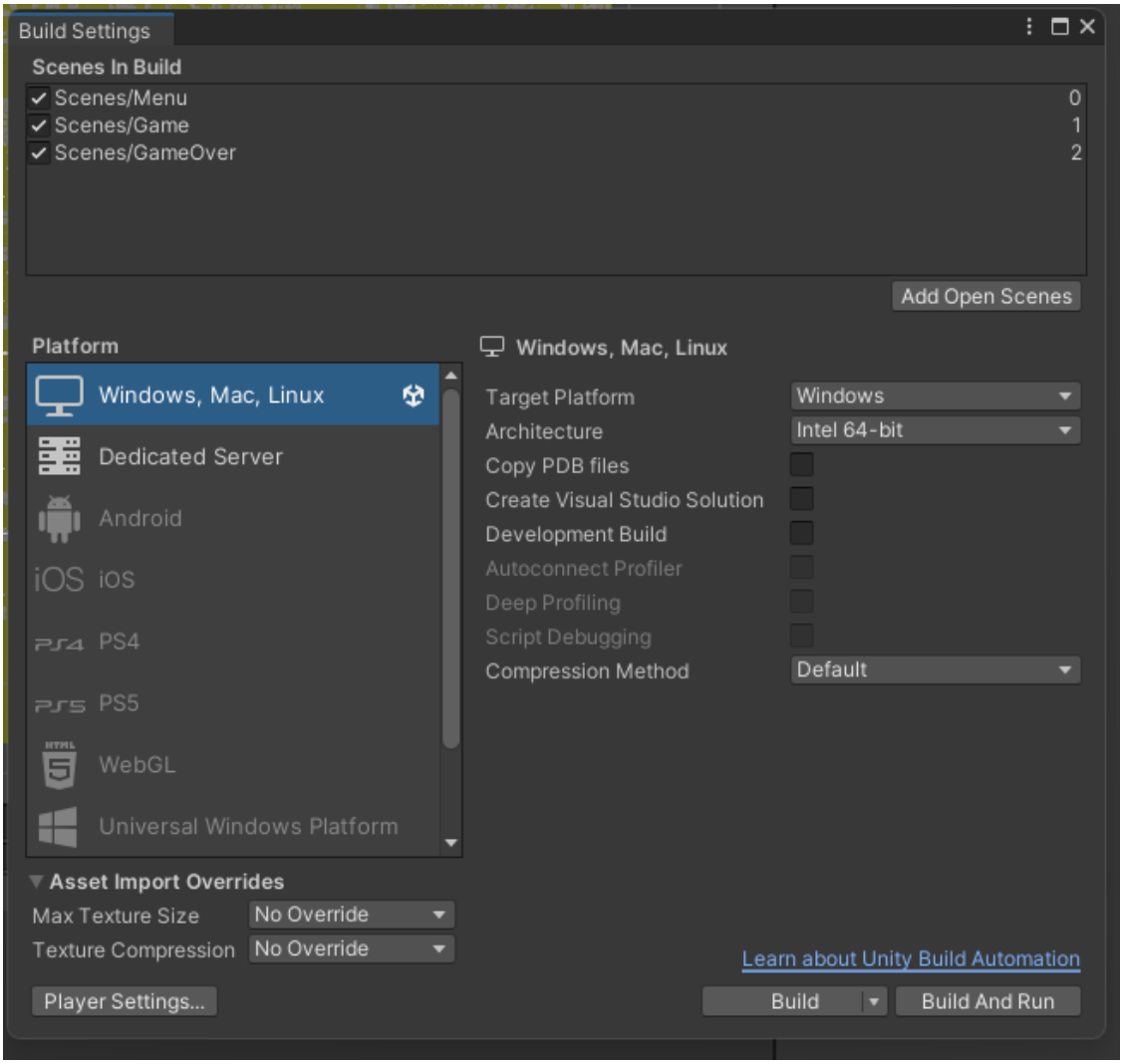

Рисунок 3.7 – вікно «Build Settings» в редакторі Unity

Платформа цільової збірки обрана як «Windows, Mac, Linux», а саме – Windows (Intel 64-bit), що є найпоширенішою платформою для десктопної гри. Додаткові параметри, як-от «Development Build», «Autoconnect Profiler», «Deep Profiling» та «Script Debugging», залишаються вимкненими, що означає підготовку білду для фінального релізу, а не для внутрішнього тестування. Це дозволяє зменшити розмір гри, прискорити завантаження та уникнути витоку технічної інформації про гру.

Далі налаштування переходять до секції «Player» (рис. 3.8), де визначаються глобальні характеристики додатку.

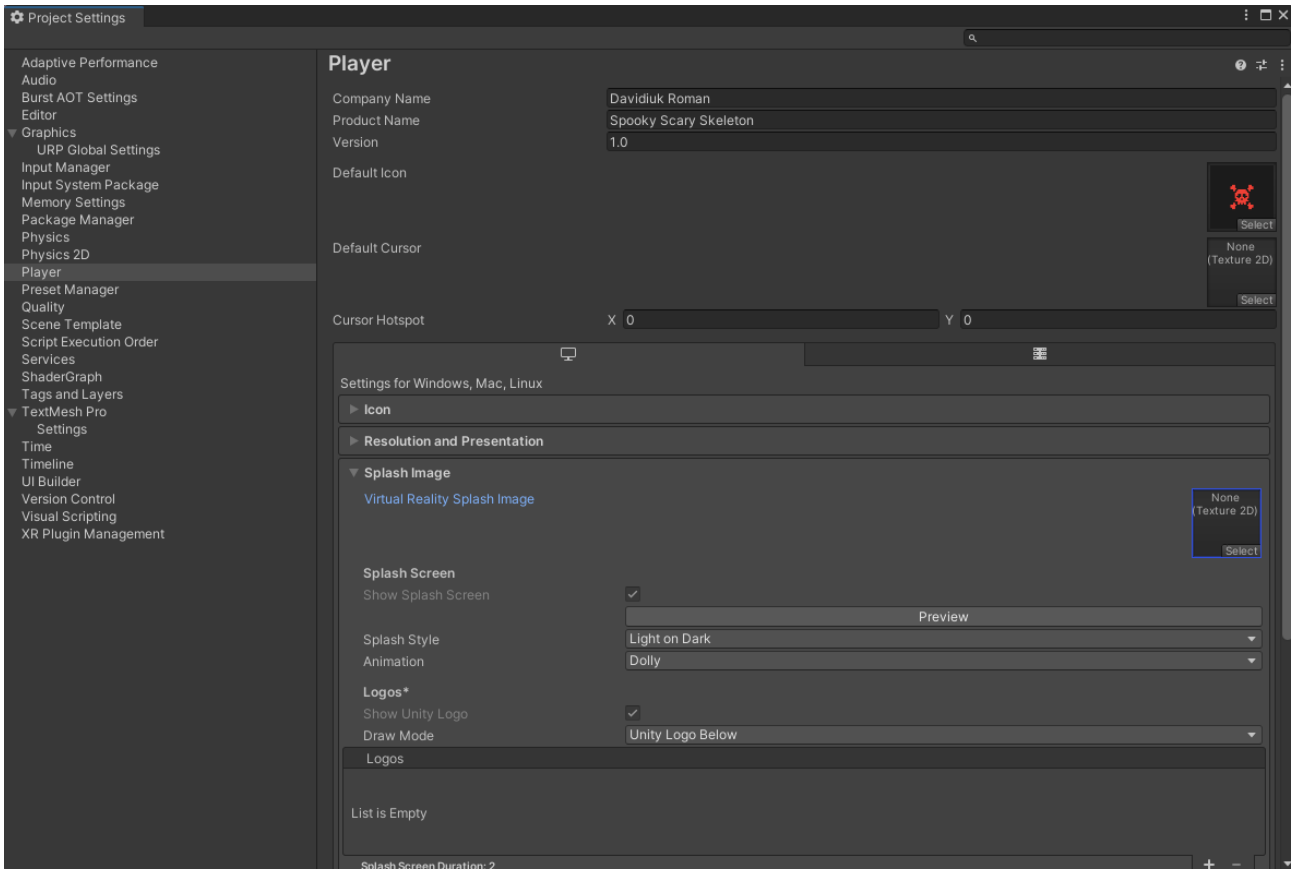

Рисунок 3.8 – вкладка «Player» вікна «Project settings» в редакторі Unity

Іконка гри (рис. 3.9) задана як кастомізоване зображення черепа червоного кольору, що символічно відповідає тематиці гри. Це зображення буде використовуватись як іконка виконуваного файлу, а також у ярликах системи.

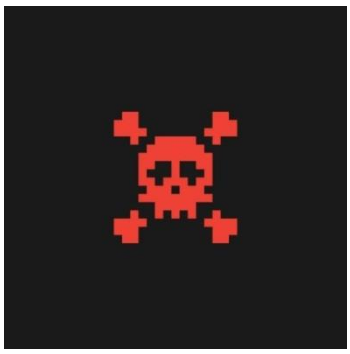

Рисунок 3.9 – Іконка гри [15]

Не менш важливим є розділ налаштувань якості графіки (рис. 3.10). У представленому прикладі активовано три рівні якості: «Low», «High» і «Ultra», серед яких «Ultra» встановлено як рівень за замовчуванням. Це означає, що гра при запуску використовуватиме максимальну якість графіки, якщо користувач не змінить налаштування вручну.

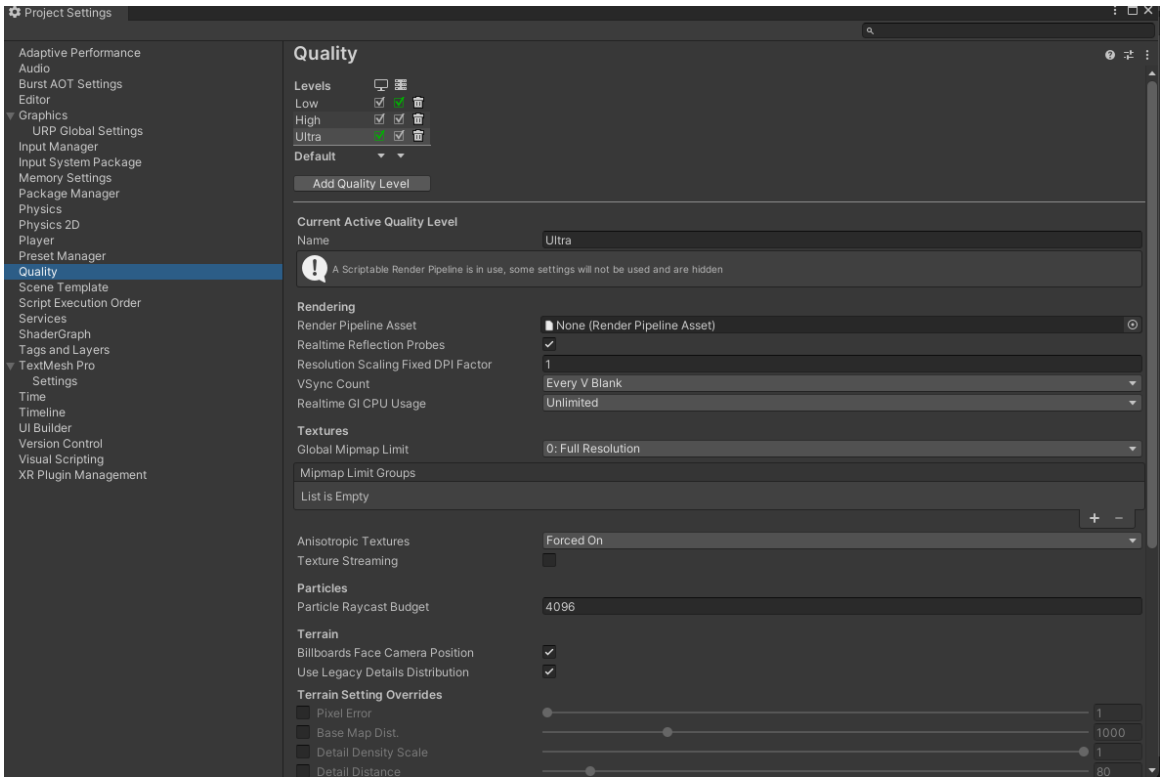

Рисунок 3.10 – Вкладка «Quality» вікна «Project settings» в редакторі Unity

У налаштуваннях білду також відсутні будь-які зміни для «Asset Import Overrides», тобто використовується глобальна конфігурація текстур без

обмежень на максимальний розмір або тип стиснення. Це дає змогу уникнути деградації якості візуальних матеріалів у процесі компіляції. «Compression Method» встановлено як «Default», що передбачає використання стандартного алгоритму стискання ресурсів, оптимізованого для балансу між розміром і продуктивністю.

Таким чином, налаштування білду гри в Unity є комплексним процесом, що охоплює визначення цільової платформи, обрання сцен, налаштування продуктивності, якості візуалізації, брендування продукту та поведінки програми при запуску. Продемонстровані параметри свідчать про підготовку до фінального випуску гри на платформу Windows із акцентом на високу графічну якість, збереження візуального стилю, зручну сценову структуру та відповідну презентацію продукту при запуску. Усі ці аспекти створюють завершений і професійний вигляд для кінцевого користувача, забезпечуючи стабільність, естетику й оптимальну продуктивність гри.

Провівши тестування роботи програми, функціонування основних команд та упевнившись у коректному виконанні завдань, наступним етапом є компіляція усіх файлів проєкту в єдиний виконуваний файл формату «.exe», для цього буде використовуватись програма Inno Setup (рис. 3.11).

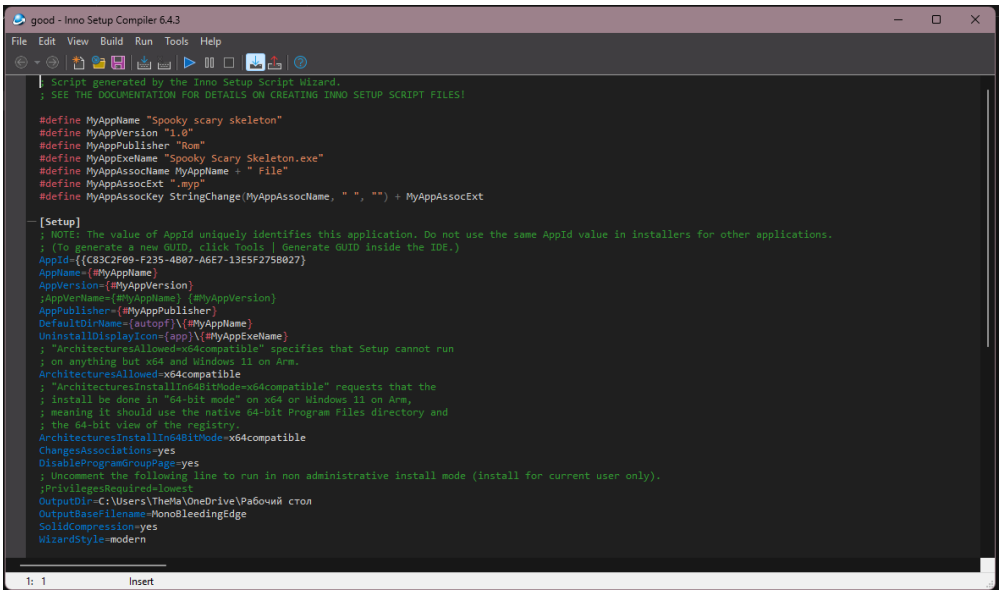

Рисунок 3.11 – Інтерфейс програми Inno Setup

Inno Setup – це безкоштовний та потужний інструмент для створення інсталяторів Windows-додатків, який широко використовується розробниками для упакування своїх програм у зручний і професійно оформлений інсталяційний процес. Незалежно від рівня досвіду, володіння цим інструментом є важливим етапом для ефективного розповсюдження програмного забезпечення.

Inno Setup є відкритим засобом створення інсталяторів, який надає широкі можливості, включно з підтримкою стиснення файлів, налаштуванням інтерфейсу встановлення, створенням ярликів, редагуванням системного реєстру та функцією видалення встановленої програми. Завдяки своїй гнучкості та простоті використання, цей інструмент дозволяє швидко створювати інсталятори навіть початківцям, при цьому залишаючи простір для просунутої кастомізації більш досвідченим користувачам.

Створення інсталятора починається з встановлення Inno Setup на комп’ютер та підготовки всіх необхідних файлів програми. Далі потрібно відкрити програму і за допомогою майстра створення скриптів налаштувати базові параметри інсталяції, зокрема назву програми, її версію та інформацію про видавця. Наступним кроком є вказання директорії з файлами програми, включаючи головний виконуваний файл, після чого задається стандартна папка встановлення, наприклад, Program Files. Також визначається, чи потрібно створювати ярлик у меню «Пуск» або на робочому столі, і додаються додаткові файли, такі як ліцензійна угода чи інструкція.

Після завершення роботи з майстром, програма автоматично згенерує скрипт, який з’явиться у вікні редактора. Цей скрипт можна за потреби доповнювати та змінювати вручну, наприклад, додавати нові файли, завдання або змінювати параметри стиснення. Приклад скрипта включає розділи, які описують загальні налаштування, мови інтерфейсу, файли, що мають бути скопійовані, ярлики, які потрібно створити, а також додаткові завдання, наприклад, створення іконки на робочому столі.

Після завершення редагування скрипта потрібно натиснути кнопку компіляції. Якщо скрипт не містить помилок, Inno Setup створює виконуваний

файл інсталятора з розширенням.

Inno Setup також дозволяє реалізувати розширену функціональність. Програма автоматично генерує можливість видалення встановленого додатку, але ця частина може бути розширена, наприклад, вказівками на видалення тимчасових файлів чи порожніх директорій. Крім того, у скрипт можна додати змінення системного реєстру, зокрема збереження шляху до встановленої програми. Для більш складних випадків підтримується мова Pascal Script, що дозволяє реалізовувати додаткові дії, наприклад, відображення повідомлень після встановлення або перевірку введених користувачем даних.

3.4 Розміщення гри на спеціалізованому сервісі

Після завершення розробки гри важливим фінальним етапом є її публікація, тобто розміщення на відповідній онлайн-платформі, де кінцеві користувачі зможуть отримати доступ до продукту, ознайомитися з його описом, переглянути зображення, завантажити або придбати його. Серед великої кількості сучасних платформ, що надають подібні послуги, особливої уваги заслуговує сервіс itch.io, який був обраний для публікації розробленої гри.

Itch.io – це незалежна платформа для розповсюдження ігор, створена спеціально для інді-розробників, яка дозволяє публікувати проєкти практично без обмежень. На відміну від інших більш складних і зарегульованих платформ, таких як Steam, itch.io пропонує зручний, гнучкий і інтуїтивно зрозумілий процес завантаження ігор, що є особливо важливим для новачків або невеликих команд розробників.

Першим кроком стало створення профілю на сайті itch.io. Після реєстрації необхідно було заповнити базову інформацію про себе як автора, що формує довіру користувачів до проєкту.

Гра була зібрана у вигляді виконуваного файлу для операційної системи Windows. Крім того, були підготовлені додаткові ресурси: скріншоти, обкладинка гри, короткий опис.

Було створено окрему сторінку гри. Вказано назву, опис, теги, жанр гри, встановити ціну (або надати гру безкоштовно), а також обрати, чи буде гра доступна для браузера або лише для завантаження.

Зібрані файли гри були завантажені на сторінку проєкту у форматі .zip. Після завантаження можна вказати, для яких операційних систем призначено архів, а також надати інструкцію з запуску.

Особливу увагу було приділено дизайну сторінки гри: завантажено банери, підібрано кольорову схему, додано інформацію про автора та, за потреби, посилання на інші ресурси. Це підвищує довіру користувачів та формує перше враження.

Після завершення налаштувань гра була опублікована. Itch.io дозволяє переглянути сторінку () так, як її бачитимуть користувачі. Таким чином, перед публікацією можна переконатися, що все виглядає належним чином і працює коректно.

Висновки до розділу 3

У ході роботи над проєктом було здійснено комплексне налаштування фінального білду гри в середовищі Unity, що є важливим етапом підготовки продукту до розповсюдження та тестування. Цей процес включав вибір та конфігурування сцен, які будуть входити до складу білду, зокрема було додано сцени меню, основного ігрового процесу та екрану завершення, що відповідає типовій структурі ігор із чітким розмежуванням функціональних блоків. Обрана цільова платформа – Windows 64-біт, що забезпечує максимальну сумісність із десктопними системами. Для релізної версії було вимкнено опції, які використовуються для розробки і налагодження, що дозволило зменшити розмір інсталяційного пакету та підвищити продуктивність. Значну увагу приділено налаштуванням глобальних параметрів програми, зокрема, було встановлено унікальну іконку, що відповідає стилістиці гри, а також оптимізовано рівні якості

графіки з пріоритетом на максимальний рівень, що сприяє поліпшенню візуального сприйняття кінцевим користувачем.

Важливим етапом стало й збереження високої якості графічних ресурсів завдяки відсутності обмежень на імпорт текстур та використанню стандартних методів компресії, що збалансував розмір файлів і швидкодію. Ці налаштування свідчать про прагнення до професійного вигляду та стабільної роботи гри, що є ключовими факторами при випуску готового продукту на платформу Windows.

Після завершення конфігурування проекту було проведено тестування функціональності основних команд гри, що підтвердило коректність їх роботи. Головною метою цього тестування було знайти помилки, неточності або нестабільність, які могли б завадити грі працювати коректно чи зіпсувати враження гравця. Наступним кроком стало формування інсталяційного пакету за допомогою інструменту Inno Setup. Ця безкоштовна та потужна програма дозволяє створювати інсталятори для Windows з налаштованим інтерфейсом, підтримкою стиснення файлів, створенням ярликів і додаткових опцій, що робить розповсюдження програмного забезпечення зручним і професійним.

Процес створення інсталятора включав встановлення Inno Setup, підготовку файлів гри та запуск майстра скриптів, який допоміг визначити основні параметри – назву програми, версію, видавця, директорію встановлення, а також вибір файлів, що підлягають копіюванню. Важливою частиною стало налаштування створення ярликів у меню «Пуск» та на робочому столі, а також додавання супровідних документів, таких як ліцензійна угода та інструкції. Згенерований скрипт надавав можливість подальшого ручного редагування та розширення функціоналу інсталятора.

Після перевірки коректності скрипту була виконана компіляція, у результаті якої було отримано готовий інсталяційний файл, придатний для розповсюдження серед користувачів. Крім того, інструмент Inno Setup дозволяє реалізовувати додаткові можливості, такі як налаштування процесу видалення програми, редагування системного реєстру та використання скриптів на мові Pascal для розширеної кастомізації інсталяційного процесу.

Таким чином, проведена робота з налаштування білду гри та створення інсталятора є завершальним і дуже важливим кроком у розробці програмного продукту, що забезпечує його стабільність, привабливий вигляд, зручність встановлення та подальшого використання на платформі Windows. Це дозволяє розробнику створити професійний продукт, готовий до впровадження у реальне середовище користувачів.

Завершальним етапом проєкту була публікація гри на платформі, щоб представити її широкій аудиторії та дати користувачам змогу самостійно ознайомитися з продуктом. Для цього ми обрали itch.io – спеціалізовану онлайн-платформу, що ідеально підходить для незалежних розробників та інді-проєктів.

Процес публікації охоплював кілька кроків: реєстрацію акаунта розробника, створення сторінки гри, додавання опису, зображень, тегів та завантаження архіву з виконуваним файлом.

Платформа itch.io надала нам повний контроль над виглядом та змістом сторінки (рис. 3.12), а також можливість оновлювати файли в будь-який момент. Це виявилося надзвичайно зручним, особливо для проєктів, які можуть розвиватися й після первинної публікації.

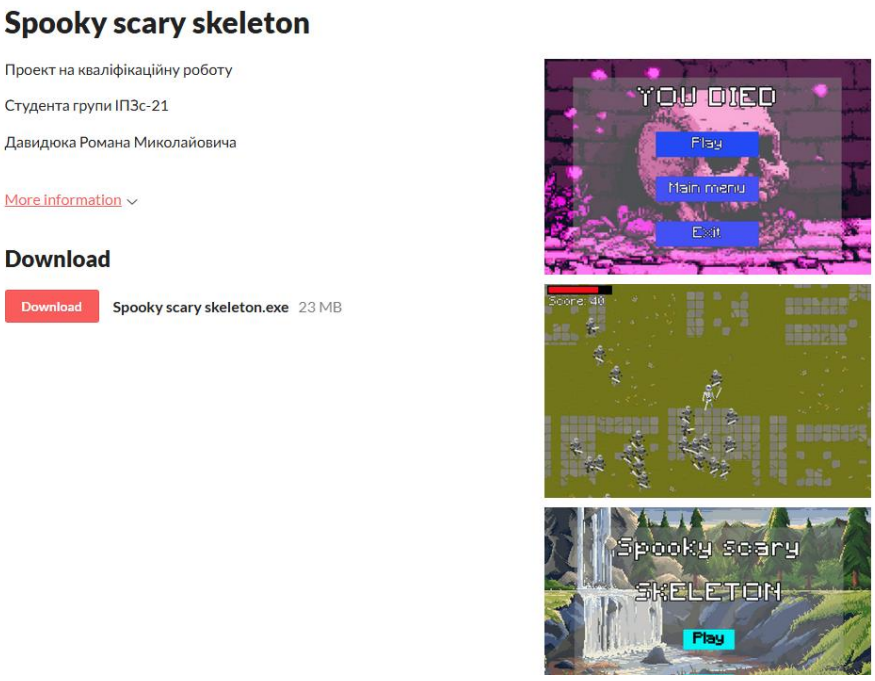

Рисунок 3.12 – Інтерфейс програми Inno Setup [16]

ВИСНОВКИ

У результаті виконання даної кваліфікаційної роботи було здійснено комплексний процес розробки комп’ютерної гри, починаючи з обґрунтування вибору жанру та формулювання вимог до проєкту і завершуючи створенням готового до розповсюдження виконуваного файлу.

Було детально досліджено актуальність жанру Survivor-like. Цей жанр набув значної популярності серед гравців завдяки своїй динамічності, високому рівню реграбельності та відчуттю постійного прогресу навіть у випадку невдачі. Аналіз сучасного ігрового ринку показав стійкий інтерес до ігор, що поєднують елементи виживання, випадкової генерації контенту та швидких ігрових сесій. Актуальність жанру також підкріплюється його здатністю залучати широку аудиторію як казуальних, так і більш досвідчених гравців, пропонуючи їм унікальний ігровий досвід, заснований на постійній адаптації та вдосконаленні навичок.

Було сформульовано як функціональні, так і нефункціональні вимоги до розроблюваної гри. Функціональні вимоги охоплювали реалізацію основних механік жанру Survivor-like: автоматичну атаку персонажа, генерацію ворогів хвилями, накопичення досвіду, систему покращень, босів, інтерфейс гравця та підрахунок очок. Нефункціональні вимоги стосувались продуктивності, адаптивності інтерфейсу до різних розширень екрана, можливості масштабування проєкту, а також забезпечення стабільної роботи за умови високого навантаження. Спираючись на аналіз існуючих ігор жанру та враховуючи потенційну цільову аудиторію, було спроєктовано основні елементи ігрового процесу, інтерфейсу користувача та взаємодії гравця з ігровим світом. Ці вимоги та проєкт базового функціоналу стали основою для подальшої розробки.

Для реалізації проєкту було здійснено ретельний вибір інструментів розробки гри. Після порівняльного аналізу кількох провідних ігрових рушіїв, включаючи Unreal Engine, Godot та власний розроблений Game Engine, було

обрано Unity. Цей вибір був обґрунтований його широкими можливостями, гнучкістю, великою екосистемою асетів та плагінів, а також активною спільнотою розробників. Unity надає зручний інструментарій для створення 2D та 3D ігор, потужну систему анімації, розвинені засоби для роботи зі звуком та графікою, а також спрощує процес розгортання гри на різних платформах.

Основною частиною роботи стала безпосередньо розробка гри в жанрі Survivor-like на обраному рушії Unity. В процесі розробки було реалізовано всі спроєктовані механіки, включаючи генерацію ігрових рівнів, систему появи та поведінки ворогів, збір та використання ресурсів, різноманітну зброю та бонуси, а також систему розвитку персонажа. Було створено візуальне оформлення гри, розроблено звуковий супровід та налаштовано інтерфейс користувача. Протягом усього процесу розробки проводилося тестування окремих компонентів та всієї гри в цілому для виявлення та усунення можливих помилок і недоліків.

Було здійснено компіляцію виконуваного файлу гри. Після оптимізації проєкту для досягнення належного рівня продуктивності було створено фінальний білд гри, готовий до запуску на комп’ютерах користувачів. Для забезпечення зручності встановлення та розповсюдження розробленої гри було також створено інсталяційний пакет за допомогою програми Inno Setup. Цей інсталятор спрощує процес встановлення гри, автоматично копіюючи необхідні файли та створюючи ярлики для зручного запуску.

Після реалізації ключових елементів ігрової логіки та інтеграції основних механік постала необхідність у проведенні повноцінного тестування розробленого продукту. Цей етап мав на меті виявлення можливих помилок, неточностей або нестабільностей, що могли б завадити коректній роботі гри або негативно вплинути на користувацький досвід. Здійснювалась перевірка окремих скриптів та модулів у межах середовища розробки Unity, де були відслідковані основні логічні помилки, порушення в обробці подій та некоректне поводження ігрових об’єктів. У результаті проведеного тестування гри було підтверджено відповідність реалізованих механік технічним вимогам та очікуванням користувача. Перевірено коректність роботи ключових елементів

геймплею: система руху персонажа, автоматична стрільба, механіка зіткнення з ворогами, система прокачування, інтерфейс користувача, а також логіка нарахування очок і генерації ворогів. Виявлені у процесі тестування помилки були класифіковані як логічні (некоректна поведінка босів, розрахунок очок), інтерфейсні (масштабування елементів на різних роздільностях), функціональні (стрільба під час паузи, залишкові об’єкти після знищення) та продуктивні (просідання FPS при великій кількості об’єктів на сцені). Після оптимізації, зокрема через впровадження object pooling, стабільність роботи гри було суттєво покращено.

Останнім етапом проєкту стало розміщення гри на публічній платформі з метою демонстрації виконаної роботи широкій аудиторії та надання можливості користувачам безпосередньо ознайомитися з ігровим продуктом. Для цього було обрано сервіс itch.io – спеціалізовану онлайн-платформу, орієнтовану на незалежних розробників та інді-проєкти. Процес публікації включав реєстрацію акаунта розробника, створення сторінки гри, додавання опису, зображень, тегів та завантаження архіву з виконуваним файлом. Платформа надала можливість повністю контролювати вигляд та зміст сторінки, а також оновлювати файли в будь-який момент, що виявилось особливо зручним для проєктів, які можуть розвиватись і після первинної публікації.

Таким чином, у межах даної кваліфікаційної роботи було повністю реалізовано цикл розробки гри в жанрі Survivor-like – від аналізу актуальності, проєктування і вибору рушія до створення ігрової логіки, графічного та звукового оформлення, компіляції та тестування. Проведено вичерпне налагодження гри з усуненням критичних помилок і оптимізацією продуктивності. Завершальним етапом стало розміщення гри на платформі itch.io, що забезпечило зручне поширення продукту та доступність для користувачів. Отриманий результат – повноцінна ігрова розробка, яка демонструє практичне застосування теоретичних знань і навичок у сфері геймдизайну та програмування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Exploring Popular Game Engines. A Comparative Overview. URL: https://www.argentics.io/game-engine-comparison (дата звернення: 22.03.2025).

2. Bramble R. The Seven Stages Of Game Development. URL: https://gamemaker.io/en/blog/stages-of-game-development (дата звернення: 27.03.2025).

3. Carson O. Comparing Popular Game Engines. PubNub. URL: https://www.pubnub.com/blog/comparing-popular-game-engines/ (дата звернення: 29.03.2025).

4. Guru G. Game Engines. A Comparative Analysis. Medium. URL: https://medium.com/@GlitchGuru/game-engines-a-comparative-analysis-ef9af01f125e (дата звернення: 05.04.2025).

5. Стаття Vampire Survivors. URL: https://cdn2.unrealengine.com/vampire-survivors-1280x800-5ff3b6e20fa0.jpg (дата звернення: 15.04.2025).

6. How to make a video game: a complete guide. URL: https://voice123.com/blog/video-games/how-to-make-a-video-game/ (дата звернення: 19.04.2025).

7. Vampire survivors level up. URL:https://oyster.ignimgs.com/mediawiki/apis.ign.com/vampire-survivors/d/df/Vampire_Survivors_level_up.png?width=640 (дата звернення: 22.04.2025).

8. Instructables. How to Make a Simple Game in Unity 3D. URL: https://www.instructables.com/How-to-make-a-simple-game-in-Unity-3D/ (дата звернення: 27.04.2025).

9. Khandelwal V. How to Create a Game In Unity. Complete Step-by-Step Guide. URL: https://www.simplilearn.com/tutorials/programming-tutorial/guide-to-learn-how-to-create-game-in-unity (дата звернення: 30.04.2025).

10. Learn to make a Game with Unity. URL: https://unitycodemonkey.com/kitchenchaoscourse.php (дата звернення: 02.05.2025).

11. Mandeville A. How to design a game. Medium. URL: https://medium.com/design-bootcamp/how-to-design-a-game-698c9fcd2015 (дата звернення: 09.05.2025).

12. Marek M. How to Build and Test Your Game in Unity. Medium. URL: https://medium.com/geekculture/how-to-build-and-test-your-game-in-unity-1eeab1b7937e (дата звернення: 12.05.2025).

13. Thompson W. The Vampire Survivors Effect: How Developers Utilize Gambling Psychology to Create Addictive Games. URL: https://hackernoon.com/the-vampire-survivors-effect-how-developers-utilize-gambling-psychology-to-create-addictive-games (дата звернення: 15.05.2025).

14. Unity game development. URL: https://unity.com/games (дата звернення: 30.05.2025).

15. Іконка програми. URL: https://media.istockphoto.com/id/1040956110/vector/pixel-skull-and-crossbones-icon-vectorillustration.jpg?s=612x612&w=0&k=20&c=2SqrmTbeOeOT7wq6uWc-X_QyBJ0VAIdEcRiuhUoPow= (дата звернення: 17.05.2025).

16. Spooky scary skeleton. URL: https://h4ppyendless.itch.io/spooky-scary-skeleton (дата звернення 20.05.2025).

ДОДАТКИ

Додаток А – Скрипт «PlayerMovement.cs»

```
using System.Collections;
using System.Collections.Generic;
using Unity.Mathematics;
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerMovement : MonoBehaviour
{
    [SerializeField]
    private float _speed;
    [Header(«Movement Bounds»)]
    [SerializeField] private Vector2 minBounds = new Vector2(-45f, -23f);
    [SerializeField] private Vector2 maxBounds = new Vector2(56f, 66f);
    private float horizontal;
    private bool isFacingRight = true;
    private Rigidbody2D _rigidbody;
    private Vector2 _movementInput;
    private Vector2 _SmoothedMovementInput;
    private Vector2 _movementInputSmoothVelocity;
    private SpriteRenderer _spriteRenderer;
    private Animator _animator;
    private void Awake()
    {
        _rigidbody = GetComponent<Rigidbody2D>();
        _animator = GetComponent<Animator>();
    }
    private void FixedUpdate()
    {
        SetPlayerVelocity();
        ClampPosition(); // <- додаємо обмеження тут
        Flip();
        SetAnimation();
    }
    private void SetAnimation()
    {
        bool isMoving = _movementInput != Vector2.zero;
        _animator.SetBool(«IsMoving», isMoving);
    }
    private void Update()
    {
        horizontal = Input.GetAxisRaw(«Horizontal»);
    }
    private void SetPlayerVelocity()
    {
        _SmoothedMovementInput = Vector2.SmoothDamp(
            _SmoothedMovementInput,
            _movementInput,
            ref _movementInputSmoothVelocity,
            0.1f);
```

```
        _rigidbody.velocity = _SmoothedMovementInput * _speed;
    }
    private void ClampPosition()
    {
        Vector3 clampedPosition = transform.position;
        clampedPosition.x = Mathf.Clamp(clampedPosition.x, minBounds.x,
maxBounds.x);
        clampedPosition.y = Mathf.Clamp(clampedPosition.y, minBounds.y,
maxBounds.y);
        transform.position = clampedPosition;
    }
    private void Flip()
    {
        if (isFacingRight && horizontal < 0f || !isFacingRight &&
horizontal > 0f)
        {
            isFacingRight = !isFacingRight;
            transform.Rotate(0f, 180f, 0f);
        }
    }
    private void OnMove(InputValue inputValue)
    {
        _movementInput = inputValue.Get<Vector2>();
    }
    public void IncreaseSpeed(float amount)
    {
        _speed += amount;
        Debug.Log($»Player speed increased to {_speed}»);
    }
}
```

Додаток Б – Скрипт «Bullet.cs»

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

public class PlayerShoot : MonoBehaviour
{
    [Header(«Shooting»)]
    [SerializeField] private GameObject _bulletPrefab;
    [SerializeField] private float _bulletSpeed = 10f;
    [SerializeField] private Transform _gunOffset;
    [SerializeField] private float _timeBetweenShots = 0.2f;
    [Header(«Boss»)]
    [SerializeField] private GameObject bossPrefab;
    [SerializeField] private Transform bossSpawnPoint;
    private bool _fireContinously;
    private float _lastFireTime;
    private int _fireLevel = 1;
    public int FireLevel => _fireLevel;
    void Update()
    {
        if (_fireContinously && Time.time - _lastFireTime >=
_timeBetweenShots)
        {
            FireBullet();
            _lastFireTime = Time.time;
        }
        // DEBUG спавн боса вручну
        if (Keyboard.current.bKey.wasPressedThisFrame)
        {
            SpawnBoss();
        }
    }
    private void FireBullet()
    {
        Vector3 mousePosition =
Camera.main.ScreenToWorldPoint(Mouse.current.position.ReadValue());
        mousePosition.z = 0f;
        Vector3 direction = (mousePosition -
transform.position).normalized;
        InstantiateBullet(direction);
        switch (_fireLevel)
        {
            case 2: StartCoroutine(DelayedSecondShot(direction)); break;
            case 3: FireParallelShots(direction); break;
            case 4: FireCircleShots(60); break;
            case 5: FireCircleShots(30); break;
            case 6: FireCircleShots(15); break;
            case 7:
                FireCircleShots(15);
```

```
                StartCoroutine(DelayedCircleShots(0.1f, 15));
                break;
        }
    }
    private void InstantiateBullet(Vector3 dir)
    {
        var bullet = Instantiate(_bulletPrefab, _gunOffset.position,
Quaternion.identity);
        bullet.GetComponent<Rigidbody2D>().velocity = dir * _bulletSpeed;
    }
    private IEnumerator DelayedSecondShot(Vector3 dir, float delay = 0.1f)
    {
        yield return new WaitForSeconds(delay);
        InstantiateBullet(dir);
    }
    private IEnumerator DelayedCircleShots(float delay, int angleStep)
    {
        yield return new WaitForSeconds(delay);
        FireCircleShots(angleStep);
    }
    private void FireParallelShots(Vector3 dir)
    {
        InstantiateBullet(Quaternion.Euler(0, 0, 15) * dir);
        InstantiateBullet(Quaternion.Euler(0, 0, -15) * dir);
    }
    private void FireCircleShots(int step)
    {
        for (int a = 0; a < 360; a += step)
        {
            float r = a * Mathf.Deg2Rad;
            InstantiateBullet(new Vector3(Mathf.Cos(r), Mathf.Sin(r),
0));
        }
    }
    public void UpgradeFireLevel(int newLevel)
    {
        _fireLevel = Mathf.Clamp(newLevel, 1, 7);
        Debug.Log($»[PlayerShoot] Fire level upgraded to {_fireLevel}»);
        // Кожні 5 рівнів — бос
        if (_fireLevel % 5 == 0)
        {
            SpawnBoss();
        }
    }
    private void SpawnBoss()
    {
        if (bossPrefab == null || bossSpawnPoint == null)
        {
            Debug.LogWarning(«[PlayerShoot] Boss prefab or spawn point not
assigned!»);
            return;
```

```
        }
        GameObject boss = Instantiate(bossPrefab, bossSpawnPoint.position,
Quaternion.identity);
        boss.transform.localScale *= 4; // Збільшуємо розмір боса
        if (boss.TryGetComponent<HeathController>(out var hc))
        {
            hc.IncreaseMaxHealth(90); // Збільшуємо здоров'я боса
            Debug.Log(«[PlayerShoot] Boss spawned with increased HP!»);
        }
        else
        {
            Debug.LogWarning(«[PlayerShoot]   Spawned   boss   has   no
HeathController!»);
        }
    }
    public void ReduceCooldown(float amount)
    {
        float minCd = 0.05f;
        _timeBetweenShots = Mathf.Max(minCd, _timeBetweenShots – amount);
        Debug.Log($»[PlayerShoot] Cooldown now {_timeBetweenShots:F2}s»);
    }

    private void OnFire(InputValue iv) => _fireContinously = iv.isPressed;
}
```

Додаток В – Скрипт «EnemyMovement.cs»

```
using System.Collections;
using System.Collections.Generic;
using Unity.Mathematics;
using UnityEngine;
using static UnityEditor.Searcher.SearcherWindow.Alignment;
public class Enemy : MonoBehaviour
{
    public void OnPlayerDied()
    {
        StopMovement();
    }
    private void Start()
    {
        GameObject player = GameObject.FindGameObjectWithTag(«Player»);
        if (player != null)
        {
            HeathController                playerHealth                =
player.GetComponent<HeathController>();
            if (playerHealth != null)
            {
                playerHealth.OnDied.AddListener(OnPlayerDied);
            }
        }
    }
    private void OnDestroy()
    {
        GameObject player = GameObject.FindGameObjectWithTag(«Player»);
        if (player != null)
        {
            HeathController                playerHealth                =
player.GetComponent<HeathController>();
            if (playerHealth != null)
            {
                playerHealth.OnDied.RemoveListener(OnPlayerDied);
            }
        }
    }
    private float horizontal;
    private bool isFacingRight = true;
    [SerializeField]
    private float _speed;
    [SerializeField]
    private float _rotationSpeed;
    private Rigidbody2D _rigidBody2D;
    private PlayerAwarenesOfController _playerAwarenesOfController;
    private Vector2 _targetDirection;
    private void Awake()
    {
        _rigidBody2D = GetComponent<Rigidbody2D>();
```

```
        _playerAwarenesOfController                                    =
GetComponent<PlayerAwarenesOfController>();
    }
    private void Update()
    {
        horizontal = Input.GetAxisRaw(«Horizontal»);
    }
    private void FixedUpdate()
    {
        UpdateTargetDirection();
        RotateTowardsTarget();
        SetVelocity();
    }
    private void UpdateTargetDirection()
    {
        if (_playerAwarenesOfController.AwareOfPlayer)
        {
            _targetDirection                                           =
_playerAwarenesOfController.DirectionToPlayer;
        }
        else
        {
            _targetDirection = Vector2.zero;
        }
    }
    private void RotateTowardsTarget()
    {
        if(_targetDirection == Vector2.zero)
        {
            return;
        }
        else
        {
            if   (_targetDirection.x   >   0   &&   !isFacingRight   ||
_targetDirection.x < 0 && isFacingRight)
            {
                isFacingRight = !isFacingRight;
                transform.Rotate(0f, 180f, 0f);
            }
        }
    }
    private void SetVelocity()
    {
        if (!_canMove || _targetDirection == Vector2.zero)
        {
            _rigidBody2D = GetComponent<Rigidbody2D>();
            _rigidBody2D.velocity = Vector2.zero;
        }
        else
        {
            _rigidBody2D.velocity = _targetDirection.normalized * _speed;
```

```
        }
    }
    private bool _canMove = true;
    public void StopMovement()
    {
        _canMove = false;
        _rigidBody2D.velocity = Vector2.zero;
    }
}
```

Додаток Г – Скрипт «BuffManager.cs»

```
using UnityEngine;
using UnityEngine.UI;

public class BuffManager : MonoBehaviour
{
    [Header(«UI»)]
    [SerializeField] private GameObject buffMenuUI;
    [SerializeField] private Button upgradeFireButton;
    [SerializeField] private Button reduceCooldownButton;
    [SerializeField] private Button increaseSpeedButton;
    [SerializeField] private Button buffHealthButton;

    [Header(«Refs»)]
    [SerializeField] private PlayerShoot playerShoot;
    [SerializeField] private PlayerMovement playerMovement;
    [SerializeField] private HeathController healthController;
    [SerializeField] private ScoreController scoreController;

    private void Start()
    {
        buffMenuUI.SetActive(false);

        upgradeFireButton.onClick.AddListener(OnUpgradeFire);
        reduceCooldownButton.onClick.AddListener(OnReduceCooldown);
        increaseSpeedButton.onClick.AddListener(OnIncreaseSpeed);
        buffHealthButton.onClick.AddListener(OnBuffHealth);

        scoreController.onThresholdReached.AddListener(ShowBuffMenu);
    }

    private void ShowBuffMenu()
    {
        Time.timeScale = 0f;
        buffMenuUI.SetActive(true);
    }

    private void CloseMenu()
    {
        buffMenuUI.SetActive(false);
        Time.timeScale = 1f;
    }

    private void OnUpgradeFire()
    {
        playerShoot.UpgradeFireLevel(playerShoot.FireLevel + 1);
        CloseMenu();
    }

    private void OnReduceCooldown()
```

```
    {
        playerShoot.ReduceCooldown(0.1f);
        CloseMenu();
    }

    private void OnIncreaseSpeed()
    {
        playerMovement.IncreaseSpeed(1f);
        CloseMenu();
    }

    private void OnBuffHealth()
    {
        healthController.IncreaseMaxHealth(10);
        CloseMenu();
    }
}
```