Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка мультиплатформенного адаптивного FPS шутера з використанням технології Unity

Виконав: студент IV курсу, групи СП-42
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Мірошниченко О.М
(підпис) (прізвище та ініціали)

Керівник Стоянов Ю.М
(підпис) (прізвище та ініціали)

Нормоконтроль Стоянов Ю.М
(підпис) (прізвище та ініціали)

Завідувач кафедри Петрик М. Р.
(підпис) (прізвище та ініціали)

Рецензент Небесний Р.М
(підпис) (прізвище та ініціали)

Тернопіль
2024

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет Комп'ютерно інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра Програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М. Р
(підпис) (прізвище та ініціали)
«__» _____ 2024 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Мірошниченко Олександр Миколайович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка мультиплатформенного адаптивного FPS шутера з використанням технології Unity

Керівник роботи к.т.н., ст. Викладач каф. ПІ Стоянов Ю. М
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» _____ 2024 року № _____

2. Термін подання студентом завершеної роботи

3. Вихідні дані до роботи Предметна область, технічне завдання, вимоги та специфікація, програмне рішення

4. Зміст роботи (перелік питань, які потрібно розробити): Вступ. Розділ 1. Аналіз предметної області та задачі. Розділ 2. Проєктування програмного рішення. Розділ 3. Тестування Програмного рішення. Розділ 4. Безпека життєдіяльності та основи охорони праці. Висновок. Список використаних джерел. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів): Діаграми, знімки екрану з проробленою роботою, ілюстративні зображення, інформативні зображення для доповнення тексту.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	к.т.н. доцент каф. МТ Лазарюк В.В		

7. Дата видачі завдання 14 лютого 2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи		
2.	Підбір джерел по темі кваліфікаційної роботи		
3.	Опрацювання джерел інформації по темі кваліфікаційної роботи		
4.	Розробка архітектури програмного рішення		
5.	Ознайомлення з новими технологіями		
6.	Розробка гри на основі опрацьованої інформації		
7.	Виконання звіту по проробленій роботі		
8.	Виконання завдання до підрозділу «Безпека життєдіяльності»		
9.	Виконання завдання до підрозділу «Основи охорони праці»		
10	Оформлення кваліфікаційної роботи		
11.	Нормоконтроль		
12.	Перевірка на плагіат		
13.	Попередній захист кваліфікаційної роботи		
14.	Захист кваліфікаційної роботи		

Студент _____ Мірошниченко О.М
(підпис) (прізвище та ініціали)

Керівник роботи _____ Стоянов Ю.М
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Розробка мультиплатформенного адаптивного FPS шутера з використанням технології Unity // Кваліфікаційна робота освітнього рівня «Бакалавр» // Мірошниченко Олександр Миколайович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-42 // Тернопіль, 2024 // С. – 76, рис. – 33, табл. – 1, кресл. – 0, додат. – 2, бібліогр. – 16.

Ключові слова: ігровий додаток, Unity, FPS, top-down, шутер, мультиплатформенний, адаптивний, C#, Zenject, Fusion, New Input System, GIT, JetBrains Rider.

Основною метою даної кваліфікаційної роботи є дослідження процесів аналізу, проєктування, розробки та тестування ігрового додатку, використовуючи передові технології та інструменти в сфері розробки ігор.

В першому розділу приділено увагу дослідженню предметної області та аналізу найпопулярніших рушіїв розробки.

Другий розділ присвячений опису проєктування ігрового додатку, архітектурного рішення, підготовки та налаштування середовища розробки.

Третій розділ описує тестування на різних платформах та надає варіанти перспектив розвитку ігрового додатку.

ANNOTATION

Development of a multi-platform adaptive FPS shooter using Unity technology // Qualification work for the educational level "Bachelor" / Miroshnichenko Oleksandr Mykolayovych // Ternopil National Technical University named after Ivan Puluj, Faculty of Computer Information Systems and Software Engineering, Department of Software Engineering, group SP-42 // Ternopil, 2024 // P. - 76, Fig. - 33, Table - 0, Drawings - 0, Supplement - 2, Bibliog - 16.

Keywords: game application, Unity, FPS, top-down, shooter, multiplatform, adaptive, C#, Zenject, Fusion, New Input System, GIT, JetBrains Rider.

The main purpose of this qualification work is to study the processes of analyzing, designing, developing and testing a game application using advanced technologies and tools in the field of game development.

The first chapter focuses on the study of the subject area and analysis of the most popular development engines.

The second section describes the design of the game application, the architectural solution, and the preparation and configuration of the development environment.

The third section describes testing on different platforms and provides options for the future development of the game application.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Геймдев – це галузь розробки комп'ютерних ігор, яка охоплює створення ігор для різних платформ, використовуючи передові технології і програмні інструменти.

Шутер – це жанр відеоігор, де гравець управляє персонажем або транспортним засобом, озброєним вогнепальною або іншою зброєю, з метою ураження ворогів.

Unity – це багатоплатформовий ігровий рушій, розроблений компанією Unity Technologies, який дозволяє створювати 2D та 3D ігри.

C# – це мова програмування з об'єктно-орієнтованою парадигмою, яка є основною мовою для розробки на платформі Unity.

Zenject – це фреймворк для інверсії управління (IoC) та залежностей (DI) у Unity, який допомагає організувати код.

Fusion – це технологія для реалізації багатокористувацьких ігор у Unity, яка забезпечує синхронізацію ігрового процесу між різними пристроями. New Input System – це нова система вводу в Unity, яка забезпечує більш гнучке і масштабоване управління пристроями вводу.

JetBrains Rider – це інтегроване середовище розробки (IDE), призначене для роботи з проектами на основі .NET і Unity. Git – це розподілена система керування версіями, яка дозволяє розробникам відслідковувати зміни в коді та співпрацювати над проектами.

GitHub – це веб-платформа для збереження та управління репозиторіями Git, яка дозволяє спільно працювати над проектами та забезпечує інструменти для керування версіями.

Unit тестування – це процес перевірки окремих частин коду (модулів) на коректність їх виконання. UML (Unified Modeling Language) – це мова моделювання для створення візуальної документації програмних систем.

Ассет – це будь-який ресурс, що використовується в грі, включаючи текстури, моделі, звуки та скрипти.

Префаб (Prefab) – це шаблон об'єкта в Unity, який може бути багаторазово використаний у грі.

Хост-клієнт (Host-Client) – це мережева архітектура, де один з гравців виступає хостом (сервером), а інші підключаються до нього як клієнти для спільної гри.

SDK (Software Development Kit) - це набір інструментів, бібліотек, документації, зразків коду та інших ресурсів, які допомагають розробникам створювати додатки для певної платформи або операційної системи.

ЗМІСТ

ВСТУП

Відеоігри стали важливим аспектом розважальної індустрії, що приносить значні прибутки та забезпечує мільйони людей по всьому світу засобами для відпочинку та взаємодії. Одним з популярних жанрів відеоігор є шутери, які залучають гравців своїм динамічним геймплеєм та захоплюючими сюжетами.

Метою цієї бакалаврської роботи є розробка мультиплатформенного адаптивного top-down шутера з використанням технології Unity. Крім того допоміжні інструменти для розробки допоможуть розробити потужну та функціональну платформу, що забезпечить зручність для гравців та полегшить управління ігровими механіками для розробників.

Планується використати інтерактивний дизайн та зручний інтерфейс, щоб забезпечити користувачам простоту та зручність взаємодії з грою. Крім того, розробка буде включати функціонал для керування персонажами, зброєю, рівнями, а також можливостями для багатокористувацької гри з використанням Fusion.

Передбачається, що даний проект сприятиме підвищенню якості ігрового процесу та залученню більшої кількості гравців. Крім того, розробка такого шутера може стати важливим кроком у кар'єрі розробника, демонструючи вміння використовувати сучасні інструменти та технології для створення складних ігор.

Очікується, що висновки та ідеї, отримані в результаті виконання цього дослідження, будуть цінним внеском у сферу розробки відеоігор та сприятимуть подальшому розвитку технологій для створення складних ігрових рішень.

РОЗДІЛ 1. РОЗГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛІЗ РУШІЇВ РОЗРОБКИ

1.1 Історія зародження відеоігор та шутерів

Зародження відеоігор бере свій початок наприкінці 1940-х та на початку 1950-х років. Ці ігри були специфічними для системи, що означало, що їх можна було грати лише на комп'ютері, для якого вони були створені, і вони представляли собою дуже прості симуляції, такі як Tic-Tac-Toe або NIM. Знадобилося десятиліття, щоб відеоігри стали доступними для гри на інших комп'ютерах.

Однією з таких ігор була "Spacewar!" – космічна бойова відеогра, розроблена для мейнфрейм-комп'ютера PDP-1 в MIT. У грі було два космічних кораблі, кожен з яких контролювався гравцем, і вони брали участь у бойових діях. Коли студенти MIT закінчили навчання, вони взяли з собою вихідний код для гри. "Spacewar!" стала однією з перших ігор, яка отримала широке розповсюдження.

У 1970-х роках розробники відеоігор почали усвідомлювати справжній потенціал відеоігор, зрозумівши, що вони можуть працювати на менших машинах і бути доступними за допомогою монет. Це усвідомлення призвело до створення аркадних відеоігор.

Першою дуже успішною аркадною грою був "Pong", випущений у листопаді 1972 року компанією Atari Inc. Ця гра виявилася надзвичайно прибутковою, приносячи кожній машині до 40 доларів на день у своєму піку та перевищуючи популярність будь-якої іншої аркадної гри того часу. Інші розробники почали копіювати цю гру, створюючи схожі ігри з м'ячем і

ракеткою, і застосовуючи загальні функції "Pong" до інших ігор, таких як баскетбол та хокей [1].

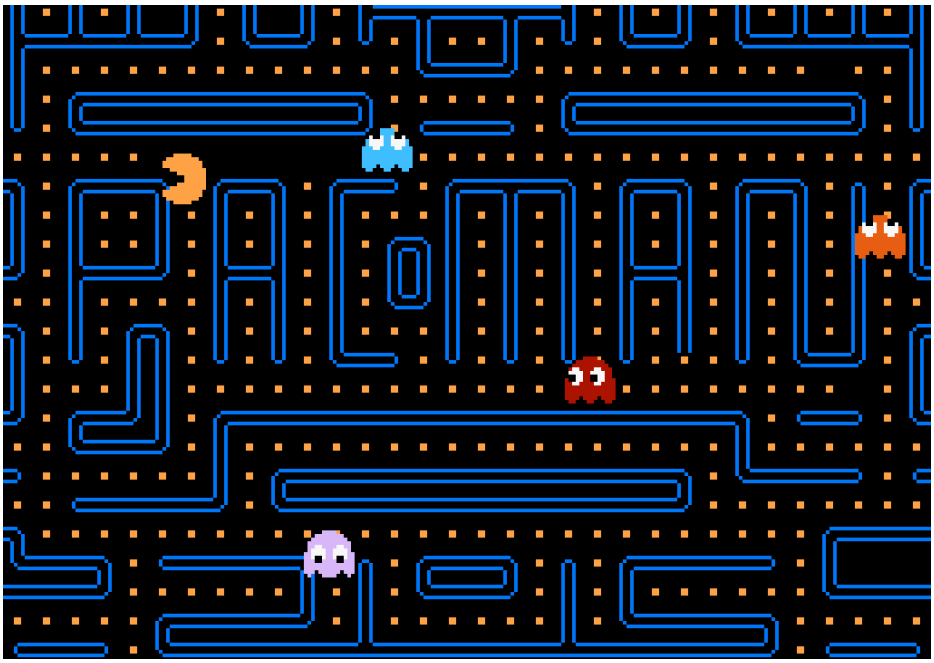

Рисунок 1.1 – Аркадна гра “Pong”

Після успіху "Pong" відбулося впровадження програмованих мікропроцесорів, що призвело до нової хвилі інновацій у розробці аркадних відеоігор. Розробники були вражені новою технологією, що дозволила створити такі ігри, як "Space Invaders", яка стала емблематичною назвою того, що пізніше буде відомо як "Золотий вік аркадних ігор".

Жанр шутерів з'явився на початку 1970-х років. Однією з перших ігор у цьому жанрі була "Space Invaders", випущена в 1978 році компанією Taito. Гра стала надзвичайно популярною і започаткувала хвилю інших шутерів. У "Space Invaders" гравці керували гарматою, яка стріляла по рядам наближаючихся інопланетян, що рухалися по екрану.

Популярність шутерів значно зросла у 1990-х роках з появою шутерів від першої особи (FPS). Однією з найвпливовіших ігор цього жанру була "Wolfenstein 3D" (1992), розроблена id Software.

Рисунок 1.2 – Гра "Wolfenstein 3D"

"Wolfenstein 3D" запропонувала гравцям новий досвід завдяки тривимірній графіці та інтенсивному геймплею. Наступним великим кроком у розвитку жанру став випуск гри "Doom" (1993), також від id Software. "Doom" стала культовою грою завдяки своїй графіці, мультиплеєру та модифікаціям, що дозволили спільноті створювати власні рівні та моди.

У 2000-х роках популярність шутерів продовжувала зростати, з'явилися такі знакові серії, як "Halo", "Call of Duty" та "Battlefield". Ці ігри підняли планку для всього жанру, пропонуючи гравцям високоякісну графіку, захоплюючі сюжети та багатокористувацькі режими. З розвитком онлайн-ігор шутери стали ще більш популярними, оскільки гравці з усього світу могли змагатися один з одним у реальному часі.

На сьогодні шутери залишаються одним з найпопулярніших жанрів у відеоіграх, залучаючи мільйони гравців і генеруючи мільярди доларів доходу щороку. Завдяки постійному розвитку технологій і творчості розробників, цей жанр продовжує еволюціонувати, пропонуючи нові враження та виклики для гравців.

Сучасні шутери часто вирізняються високою якістю графіки, реалістичними ефектами та складними сюжетними лініями. Вони пропонують різноманітні режими гри, включаючи кооперативні місії,

змагальні багатокористувацькі режими та розгалужені одиночні кампанії. Інновації в технологіях, такі як віртуальна реальність (VR) та розширена реальність (AR), також починають інтегруватися у жанр шутерів, надаючи гравцям ще більше занурення в ігровий процес.

Розробники продовжують експериментувати з новими механіками та стилями гри, що сприяє постійній еволюції жанру. Від інди-проєктів до великих блокбастерів, шутери залишаються важливою частиною ігрової індустрії, залучаючи нових гравців та утримуючи зацікавлення ветеранів жанру. Це забезпечує постійний розвиток та розширення можливостей для гравців у всьому світі.

1.2 Аналіз обраного рушія для розробки

Перед початком розробки гри необхідно визначитись з движком, бо це є однією з важливих аспектів, від неї буде залежати як саме буде будуватись гра, на якій мові програмування вона буде написана, підходи до реалізації, які допоміжні ресурси будуть використовуватись і т.д.

Для розробки я використав Unity, який надає низку переваг, які роблять його оптимальним вибором для реалізації даного проєкту.

Даний движок вирівається вперед своїми численними перевагами, а саме:

Мультиплатформенність: движок надає широкий спектр платформ, включаючи Windows, macOS, Linux, Android, iOS, консолі та WebGL. Це дозволяє розробникам легко адаптувати свої ігри до різних пристроїв і операційних систем, що є важливим для досягнення широкої аудиторії.

Інструменти розробки: Unity пропонує модульність і розширюваність, що дозволяє інтегрувати додаткові інструменти, такі як Zenject для управління залежностями, Fusion для мультиплеєрних можливостей.

Графічні можливості: Unity підтримує сучасні графічні технології, включаючи рендеринг в реальному часі, освітлення, тіні та постобробку, що дозволяє створювати як 2D, так і 3D графіку високої якості.

Фізичний рушій: вбудований фізичний рушій Unity забезпечує реалістичну фізику, що є важливим для створення інтерактивних та захоплюючих ігрових середовищ.

Ком'юніті та підтримка: Unity має велику та активну спільноту розробників, що дозволяє швидко знаходити відповіді на питання та отримувати допомогу. Регулярні оновлення рушія забезпечують підтримку нових технологій та вдосконалення існуючих функцій.

Навчальні ресурси: Unity пропонує велику кількість документації, туторіалів та онлайн-курсів, що робить його доступним для новачків та досвідчених розробників [2].

Рисунок 1.3 – Інтерфейс ігрового движка Unity

Серед інших рушіїв для розробки можу виділити: Unreal Engine [3], Godot [4] та CryEngine [5].

Unreal Engine, розроблений Epic Games, відомий своєю високоякісною графікою та потужними інструментами розробки. Серед основних переваг - потужний редактор матеріалів та шейдерів, підтримка C++ та Blueprint, що дозволяє швидко створювати прототипи та налагоджувати ігрову логіку. Цей рушій ідеально підходить для розробки AAA-ігор з реалістичною графікою. Однак його високий поріг входження, високі системні вимоги та менша кількість навчальних ресурсів, ніж у Unity, можуть стати перешкодою для початківців та невеликих студій [3].

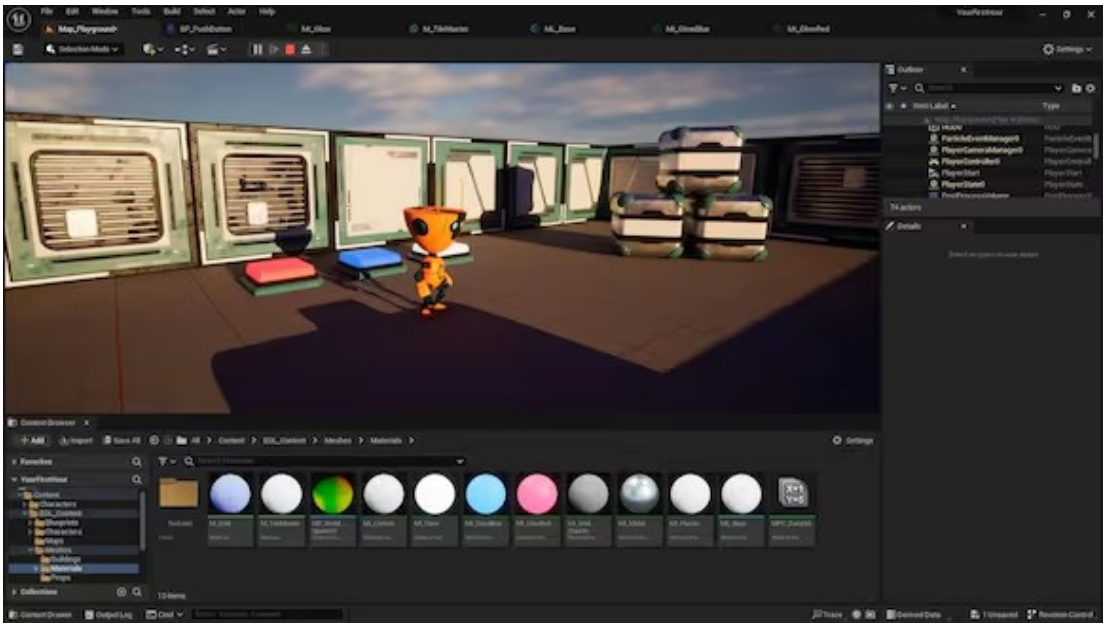

Рисунок 1.3 – Інтерфейс ігрового движка Unreal Engine

Godot - це безкоштовний рушій з відкритим вихідним кодом, що робить його доступним для всіх розробників. Він відомий своєю простотою використання та підтримкою як 2D, так і 3D графіки. Потужний вбудований редактор скриптів робить його ідеальним для інді-розробників та

початківців. Однак цей рушій менш потужний для великих проектів, має обмежену підтримку і менше підходить для розробки VR-ігор [4].

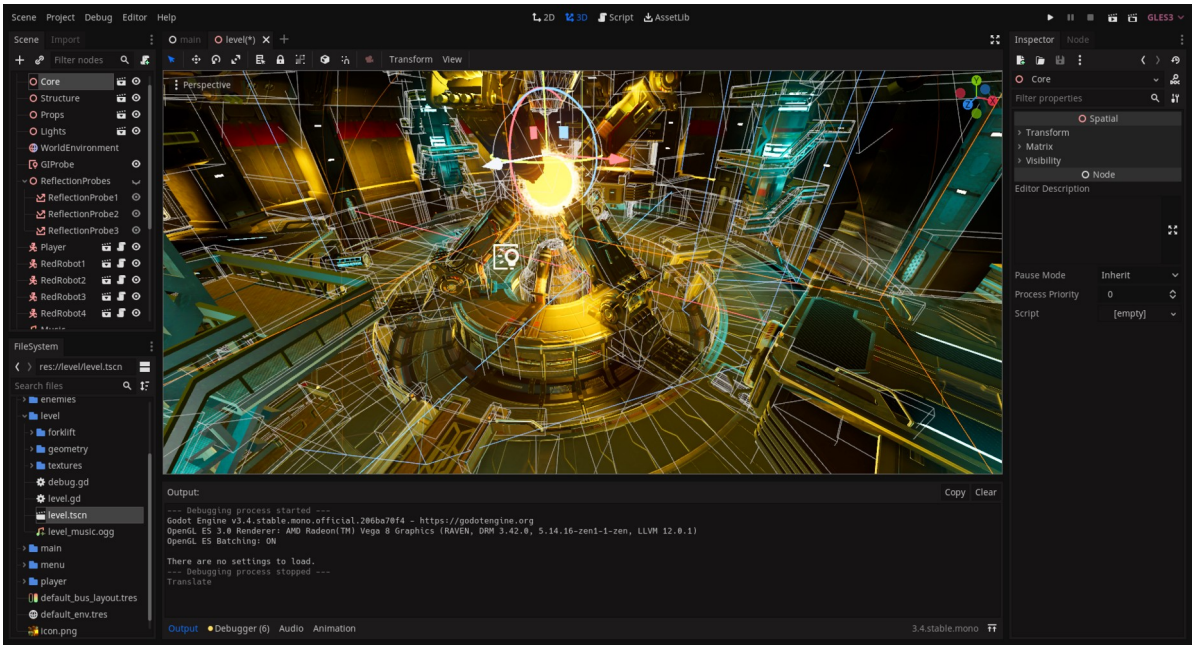

Рисунок 1.4 – Інтерфейс ігрового движка Godot

CryEngine славиться своєю високою якістю графіки та потужним фізичним рушієм. Він широко використовується для створення фотореалістичних ігор і підтримує VR. Однак, як і Unreal Engine, CryEngine має високий поріг входу, високі системні вимоги та меншу кількість навчальних ресурсів, що може ускладнити процес навчання і використання для нових розробників [5].

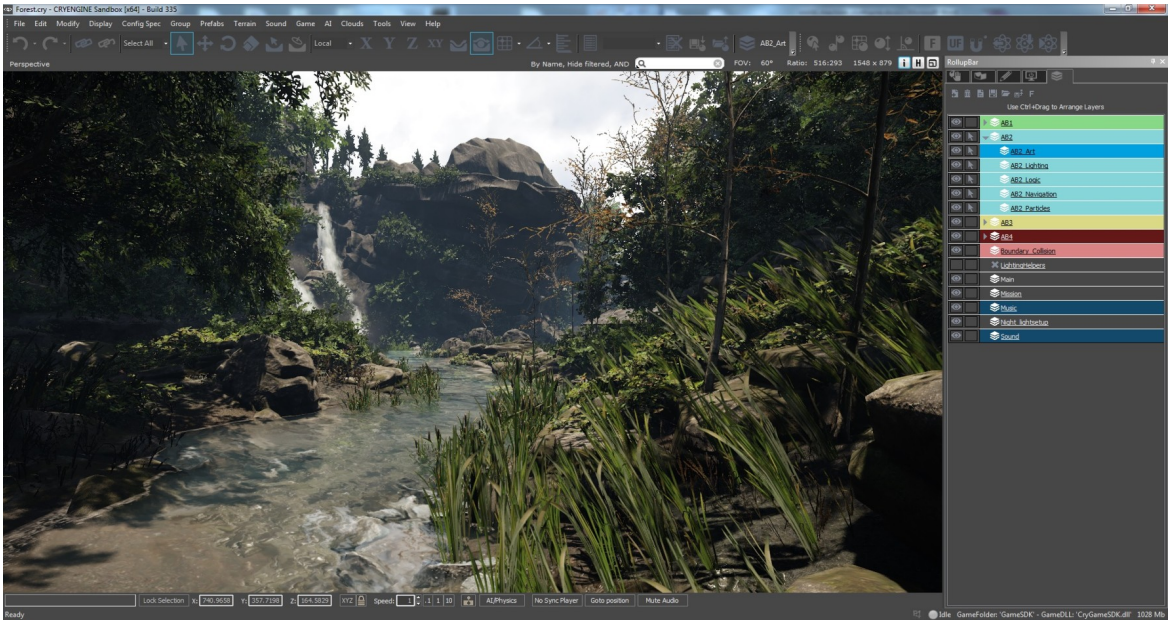

Рисунок 1.5 – Інтерфейс ігрового движка CryEngine

Таким чином, Unity є оптимальним вибором для розробки завдяки своїм широким можливостям, легкості використання та підтримці з боку спільноти. Unity дозволяє створювати високоякісні, динамічні та адаптивні ігри, що відповідають сучасним стандартам і вимогам гравців [2].

РОЗДІЛ 2. ПРОЄКТУВАННЯ ІГРОВОГО ДОДАТКУ

2.1 Аналіз жанру ігрового додатку

Коли було обрано загальну концепцію, а також сторону в яку буде рухатись проєкт, також потрібно визначитись з його жанром.

При виборі жанру для свого проєкту я вирішив обрати Shoot 'em up [6] замість FPS [7] з кількох важливих причин.

Shoot 'em up — це жанр гри, в якій головний герой бореться з великою кількістю ворогів, стріляючи в них, а також ухиляючись від їхнього вогню. Даний жанр являє собою хвилі ворогів, які можуть мати різні характеристики, паттерни поведінки і т.д, з якими гравець повинен боротись та вижити до кінця, зазвичай розглядаються з перспективи top-down або side-view [6].

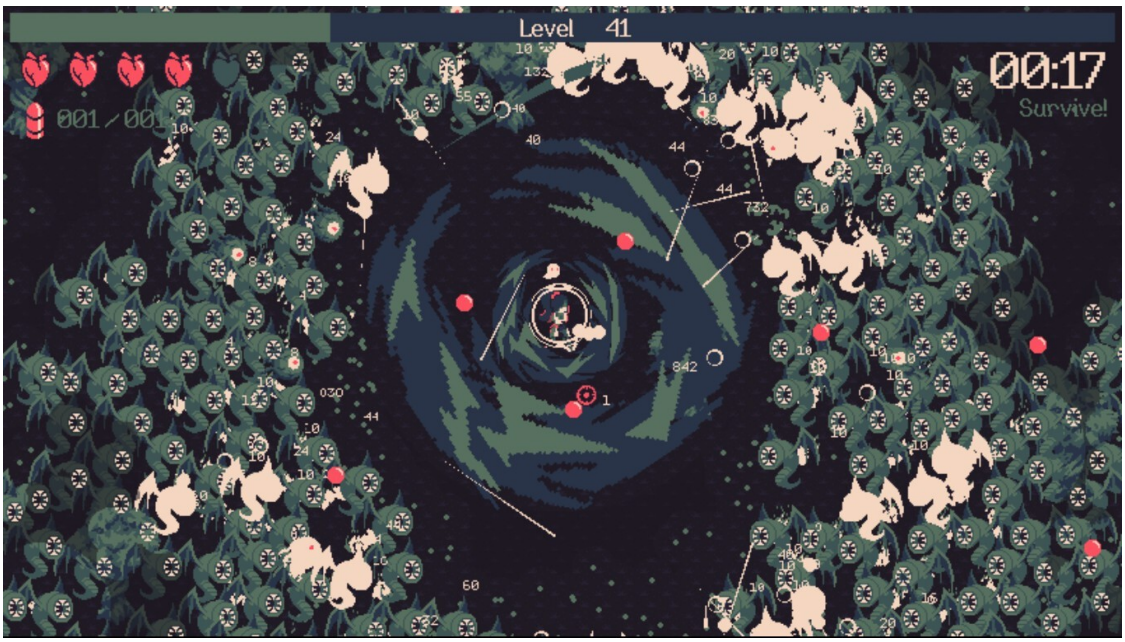

Рисунок 2.1 – Гра «20 minutes till dawn” в жанрі Shoot 'em up

Причини вибору іншого жанру:

- Має простіший геймплей та управління. Це означає менше складних механік і анімацій, що робить розробку швидшою та менш витратною. У FPS, як правило, потрібно реалізувати складніші фізичні взаємодії, тривимірну навігацію та детальніші моделі персонажів, що значно збільшує час розробки та ресурси.
- Легша адаптація для різних платформ, зокрема мобільних пристроїв. Вони мають простіше управління, яке підходить як для сенсорних екранів, так і для традиційних контролерів. FPS гри вимагають точного управління, що часто важко реалізувати на мобільних пристроях без додаткових периферійних пристроїв.
- Розробка апаратного забезпечення менш вимоглива порівняно з FPS, які потребують високої продуктивності для обробки складних тривимірних середовищ і фізики. Це дозволяє розробити гру, яка буде доступною для більш широкої аудиторії, включаючи користувачів зі старішими або менш потужними пристроями.
- FPS ігри часто вимагають високого рівня реалістичності, як у графіці, так і в механіках, що додає складності в розробці. Shoot 'em up жанр не вимагає такої реалістичності, що дозволяє зосередитися на динамічному та захоплюючому геймплеї без необхідності досягати реалістичних стандартів.

Обираючи жанр Shoot 'em up було зроблено врахування кількох важливих факторів: простота розробки та управління, мультиплатформенність, зниження вимог до апаратного забезпечення, візуальна привабливість, менші вимоги до реалізму та висока реіграбельність. Ці переваги дозволяють створити гру, яка буде доступною, захоплюючою та привабливою для широкої аудиторії. Крім того, використання Unity забезпечує ефективну реалізацію всіх необхідних

аспектів гри, що робить процес розробки більш ефективним та зручним. Таким чином, жанр Shoot 'em up дозволяє створити більш доступну та привабливу гру, яка зможе зацікавити різноманітну аудиторію гравців [6].

2.2 Проєктування програмного додатку

Процес проєктування ігрового додатку починається з формування завдання, яке визначає основні функціональні вимоги та механіки гри.

Основний ігровий процес включатиме хвилі ворогів, які будуть постійно спавнитися протягом гри. Під час хвиль будуть також з'являтися різні припаси, такі як аптечки, бомби та скриньки з патронами. Після завершення кожної хвилі всі вороги будуть знищені, але припаси продовжуватимуть з'являтися через певні проміжки часу. По закінченню всіх хвиль або після смерті всіх гравців, буде відображено список гравців із кількістю завданих збитків та вбитих ворогів. Гравець, який помер, стає спостерігачем і не може продовжувати гру.

Передбачено реалізувати три типи ворогів:

- Звичайний зомбі: повільний зомбі зі слабкою атакою.
- Прокачений зомбі: повільний зомбі з підвищеним здоров'ям та збільшеною силою атаки, але низькою частотою атак.
- Скелет: ворог, який атакує на відстані та має високу швидкість.

Також буде реалізовано три види зброї:

- Пістоль: зброя з малою дистанцією атаки та високою шкодою.
- Дробовик: зброя із середньою дистанцією атаки, низькою шкодою, що стріляє дробом.

- Автомат: зброя з середньою шкодою та великою дистанцією атаки.

Гра включатиме три види підбираних предметів:

- Аптечка: відновлює здоров'я гравця.
- Бомба: знищує ворогів у певному радіусі.
- Скриньки з патронами: поповнюють боєприпаси гравця.

Ігровий процес поділено на три хвилі:

- Перша хвиля триватиме одну хвилину з десяти секундною перервою, і міститиме звичайних зомбі та патрони як бонуси.
- Друга хвиля триватиме три хвилини з тридцяти секундною перервою, з ворогами у вигляді звичайних зомбі та скелетів, бонусами будуть патрони та аптечки.
- Третя хвиля триватиме п’ять хвилин з тридцяти секундною перервою, включатиме звичайних зомбі, скелетів та прокачених зомбі, а бонусами будуть патрони, аптечки та бомби.

Однією з важливих технічних складових гри буде механізм підключення кількох гравців до однієї ігрової сесії. В головному меню гравець матиме змогу вибору одного з трьох доступних персонажів, лобі гри буде складатися з двох гравців. Також буде розроблена механіка відображення таймера до кінця хвилі, і реалізована система зміни управління залежно від платформи. Після того як всі гравці зайшли в лобі починається гра, їм видається випадкова зброя, яка не повторюється між ними.

2.3 Підготовка проєкту до розробки

Після вибору рушія для розробки та розібраним детально проєктування наступним кроком буде підготування допоміжних інструментів, які будуть використовуватись на протязі всієї розробки ігрового продукту.

Так як було вирішено додати багатокористувацький режим мною було увалено обрати один з найпопулярніших для цього SDK, а саме Photon Fusion [8].

Photon Fusion - це висококласний SDK для неткоду з передачею станів, створений для професіоналів Unity, який надає гравцям найкращий досвід для будь-якого ігрового процесу з декількома варіантами мережевої топології.

Fusion підтримує дві принципово різні мережеві топології з однаковим API, а також однокористувацький режим без підключення до мережі [8].

Першим кроком при запуску Fusion є вибір між режимом Server/Host і Shared mode.

Рисунок 2.2 – Підручник для вибору режиму

Отож, коли ми розібрались, як поділяються режими і як для чого їх краще використовувати необхідно зрозуміти, як вони влаштовані, нижче буде наведено детальний опис цих режимів.

Server Mode — в даному режимі сервер має повну та виключну державну владу над усіма об'єктами, без винятків. Клієнти можуть змінювати мережеві об'єкти, лише надсилаючи свої дані на сервер (і чекаючи реакції сервера на ці дані) або запитуючи зміни за допомогою RPC. Серверний додаток збирається з проекту Unity і запускає повну headless Unity build. Ця збірка повинна бути розміщена на серверній машині або хмарному сервері. Photon не надає серверів для розміщення виділеного серверного додатку Fusion.

Host Mode(Host Client) — режим виступає одночасно і сервером, і клієнтом. Хост має локальний програвач, опитує вхідні дані для нього і інтерполює рендеринг відповідно до очікувань клієнта. Загалом, цей режим еквівалентний режиму сервера, але набагато дешевший у використанні, оскільки не потребує витрат на хостинг виділеного сервера. Однак за це доводиться платити довірою державного органу - іншими словами, недобросовісний хост може шахраювати. При роботі в режимі хостингу з-за брандмауера або маршрутизатора хмара Photon прозоро забезпечує UDP-пробивання або ретрансляцію пакетів, якщо це необхідно. Оскільки сесія належить хосту, вона буде втрачена, якщо хост відключиться. Fusion надає механізм міграції хостів, який дозволяє передавати повноваження новому клієнту в подібних ситуаціях, однак це не відбувається автоматично в режимі хоста (на відміну від Shared mode) і вимагає спеціальної обробки в клієнтському коді.

Shared Mode — даний режим має повноваження над мережевими об'єктами, які розподілені між усіма клієнтами. Зокрема, кожен клієнт

спочатку має державну владу над об'єктами, які він породжує, але може передати цю владу іншим клієнтам. За бажанням, клієнтам може бути дозволено забирати державні повноваження за власним бажанням. Симуляція завжди рухається вперед з однаковою швидкістю на всіх клієнтах. Сеанс мережі в режимі спільного доступу належить хмарі Photon і залишається активним, доки до нього підключений будь-який клієнт [8].

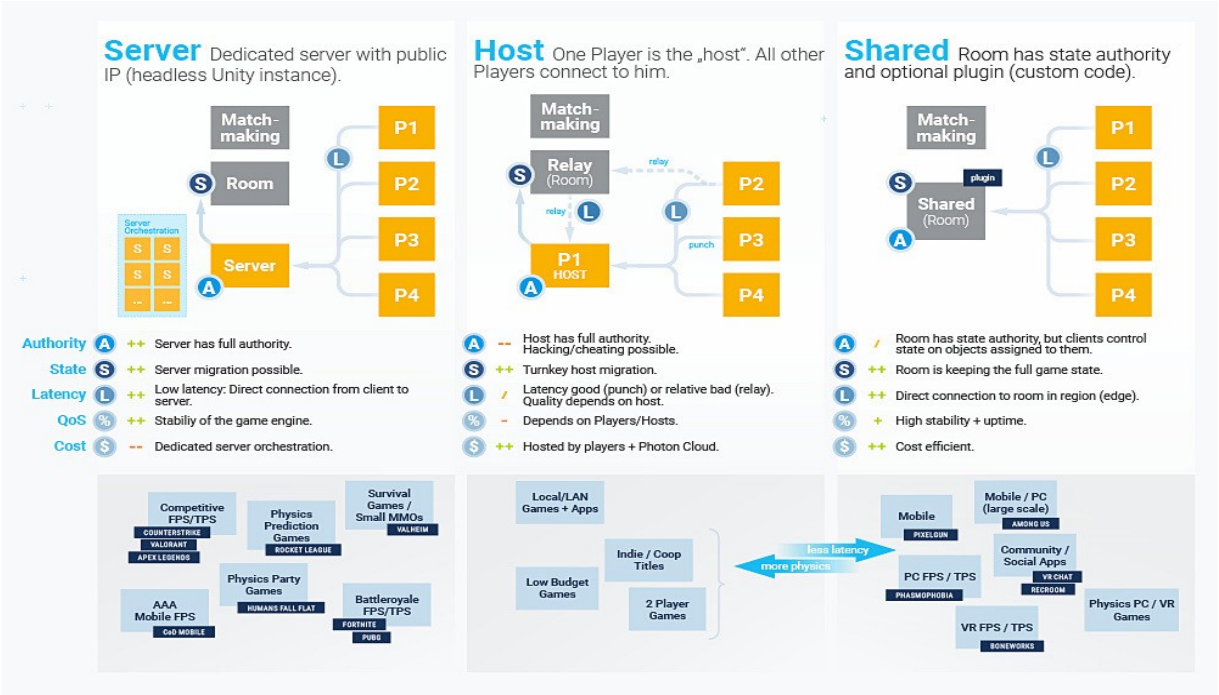

Рисунок 2.3 – Fusion Network Topologies

Отож, після аналізу всіх видів модів Photon Fusion мною було прийнято рішення використати Host mode, який дозволяє оптимізувати ресурси та забезпечити високу ефективність, що є ключовими факторами для успішної розробки та впровадження гри [8].

2.4 Налаштування проєкту

В першу чергу, після проєктування ігрового додатку та обраних SDK для роботи, слід вибрати версію на якій буде створюватись продукт, обрати необхідні модулі, створити проєкт і врешті налаштувати робоче середовище.

Спершу варто встановити головний лаунчер, через який починається основна робота — Unity Hub. Дана програма спрощує навігацію, завантаження модулів, керування проєктами та інсталяціями Unity [9].

Для розробки цієї кваліфікаційної роботи була обрана версія 2022.3.10f, яка на момент написання є найновішою та найстабільнішою версією.

При завантаженні версії потрібно вибрати необхідні модулі, для того щоб мати можливість запустити ігровий додаток на різних платформах, включаючи мобільні пристрої та персональні комп’ютери, що дозволить охопити, якомога більшу аудиторію і дасть безцінний досвід розробки кросплатформенних ігрових додатків.

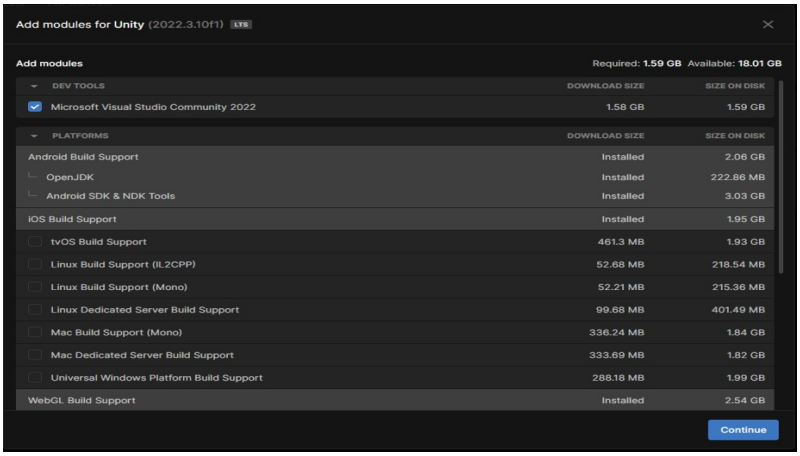

Рисунок 2.4 – Інсталяція необхідних модулів в Unity Hub

Коли всі необхідні модулі були інстальовані, і проєкт створено наступним кроком є інсталяція і налаштування допоміжних засобів розробки.

Для підключення і налаштування SDK Photon Fusion, потрібно завантажити сам SDK. Перш ніж імпортувати його потрібно налаштувати середовище розробки. Так як, деякі налаштування буде збережено в ScriptableObject, і щоб параметри залишались читабельними в будь-який час, потрібно в Asset Serialization встановити мод Force Text у меню Text Edit > Project Settings > Editor > Asset Serialization > Mode [8].

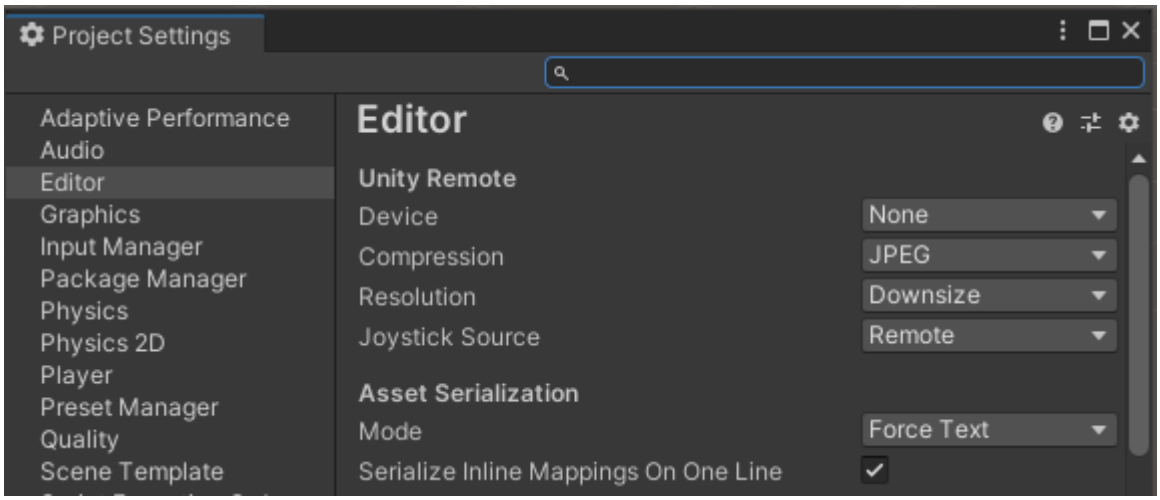

Рисунок 2.5 - Встановлення режим серіалізації на «Force Text»

Fusion IL Weaver генерує мережевий код низького рівня та вставляє його в Assembly-CSharp.dll, щоб не виникало конфліктів треба використати пакет Mono Cecil. Пакет можна встановити через Unity Package Manager.

Для цього можна перейти до Window > Package Manager > Click the + icon > Додати пакет з git URL і вставити дане посилання com.unity.nuget.mono-cecil@1.10.2.

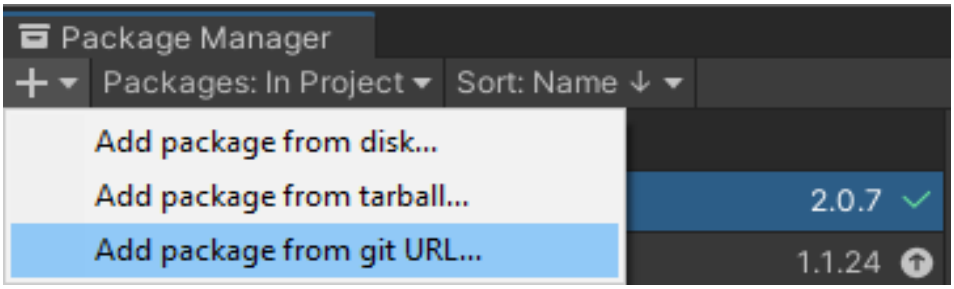

Рисунок 2.6- Додавання пакету Mono Cecil через Unity Package Manager

Після виконання всіх кроків проєкт готовий до імпортування SDK, і може бути імпортований за допомогою Assets > Import Package > Custom Package

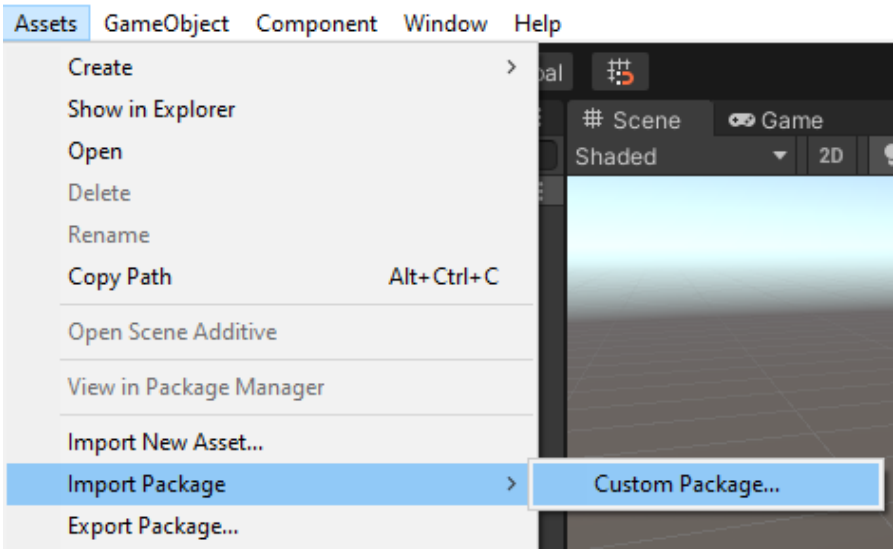

Рисунок 2.7 - Імпортування SDK Photon Fusion

Останнім же кроком для повноцінного налатування буде створення App ID, який використовується для визначення програми, підключення гравців за допомогою програми та пов’язує програму з відповідним типом сервера.

Після його створення буде доступний ID, який потрібно вставити в поле Fusion App Id у Fusion Hub [8].

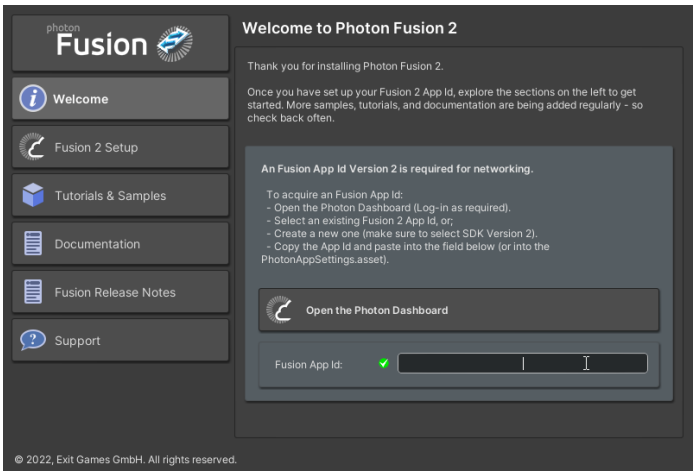

Рисунок 2.8 - Поле для ID у Fusion Hub

2.5 Проєктування архітектури додатку

Архітектура відіграє вирішальну роль у створенні програмного забезпечення, будучи фундаментом для всього процесу розробки. Важливо мати чітко визначену архітектуру, яка забезпечує стабільність, масштабованість та зручність підтримки додатку. У розробці ігрових додатків ці аспекти особливо важливі, оскільки включають різні механіки та компоненти, які повинні працювати злагоджено та ефективно.

Проєктування програмного додатку починається з визначення основних компонентів та їх взаємодій. Продукт, який розробляється включає кілька важливих елементів: лобі для гравців, вибір персонажів, хвилі ворогів, різні види зброї та підбираних предметів, справн ворогів,

систему здоров’я, таблицю лідерів, систему зіткнення, а також механізми підключення гравців до сесій.

Тепер, маючи загальне уявлення про архітектуру додатку, ми можемо детальніше розглянути основні класи проєкту, його атрибути та методи, які забезпечують повноцінну роботу додатку.

Розглянемо детально функціонал ігрового додатку починаючи з головного меню, яке зустрічає гравця при запуску гри.

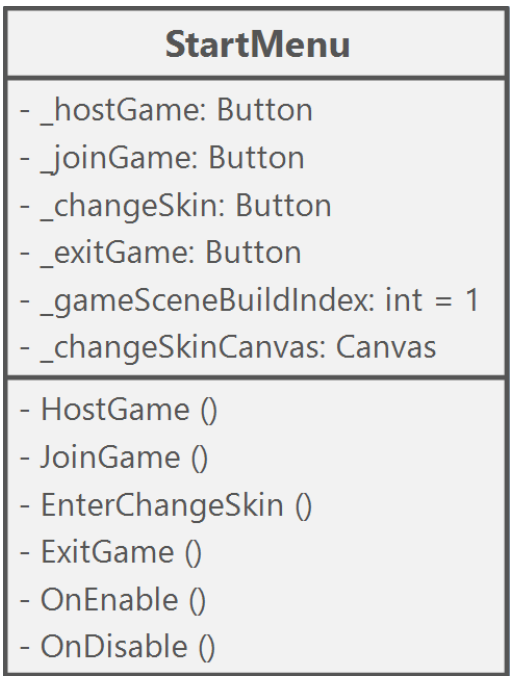

Рисунок 2.9 - Клас StartMenu

Клас StartMenu, який відображений на рисунку відповідає за функціонал головного меню. В нього входить всі кнопки, які розташовані на головному меню, а саме:

- Host game
- Join game
- Change Skin

- Exit game

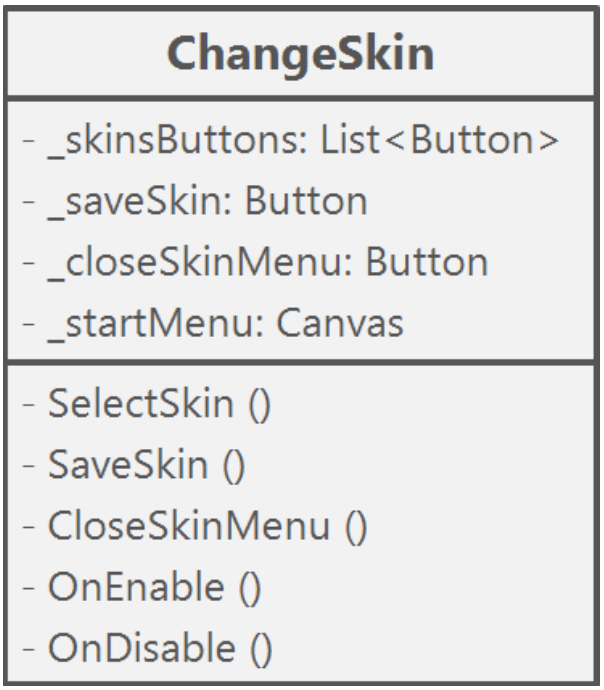

Рисунок 2.10 - Клас ChangeSkin

Як вказано на рисунку 2.10, клас ChangeSkin відповідає за вибір персонажа, та за збереження даних про вибраного персонажа.

Переглянувши всі компоненти головного меню, наступним кроком буде перехід до ігрової сцени, де відбувається сам процес гри.

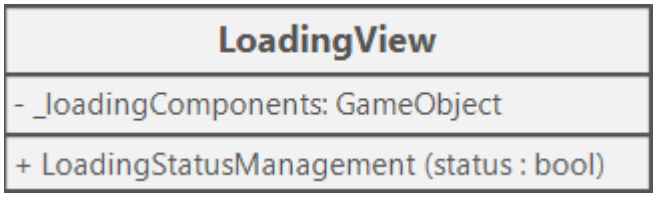

Рисунок 2.10 - Клас LoadingVeiw

При переході на нову сцену нас зустрічає анімована заставка Loading, за яку відповідає клас LoadingVeiw, яка зображена на рисунку 2.10.

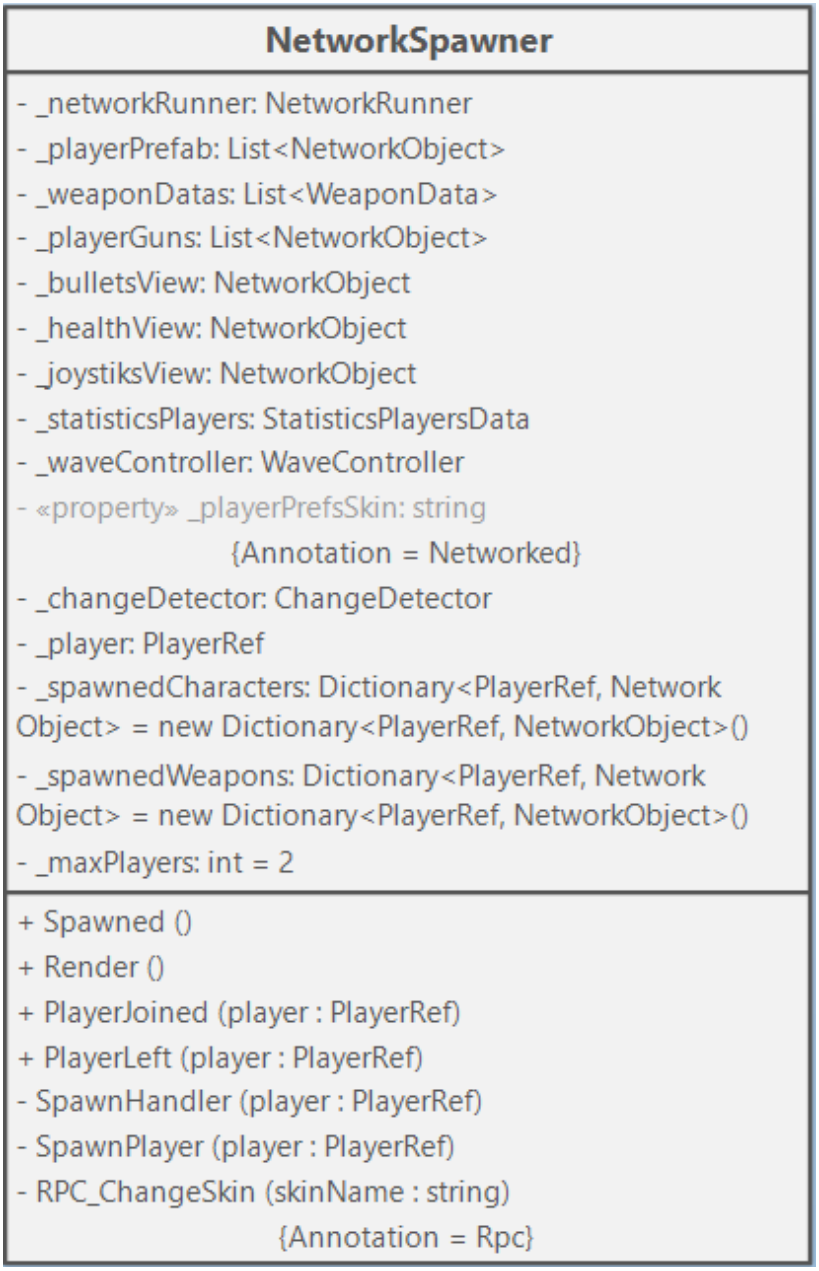

Рисунок 2.11 - Клас NetworkSpawner

Клас NetworkSpawner, який відображений на рисунку 2.11 відповідає за появу персонажа на ігровій сцені, виявлення нових гравців, які підключились до сесії і відключились від неї.

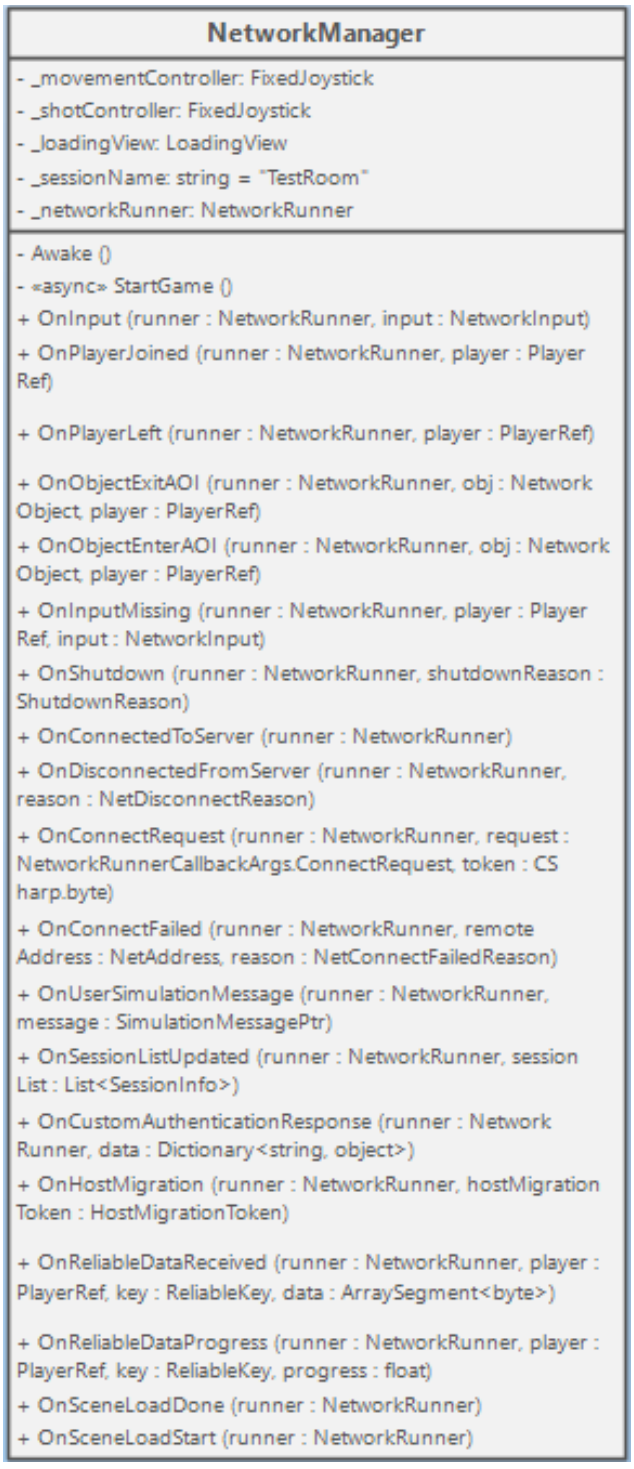

Рисунок 2.12 - Клас NetworkManager

Клас NetworkManager відповідає за створення кімнати і призначає ролі, а саме чи гравець є хостом сесії, чи клієнтом.

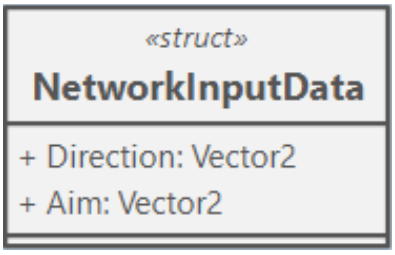

Рисунок 2.13 - Структура NetworkData

Немало важлива структура, яка відображена на рисунку 2.12, яка відповідає за збереження даних, за допомогою яких буде будуватись і вираховуватись рух персонажа і позиціонування прицілу.

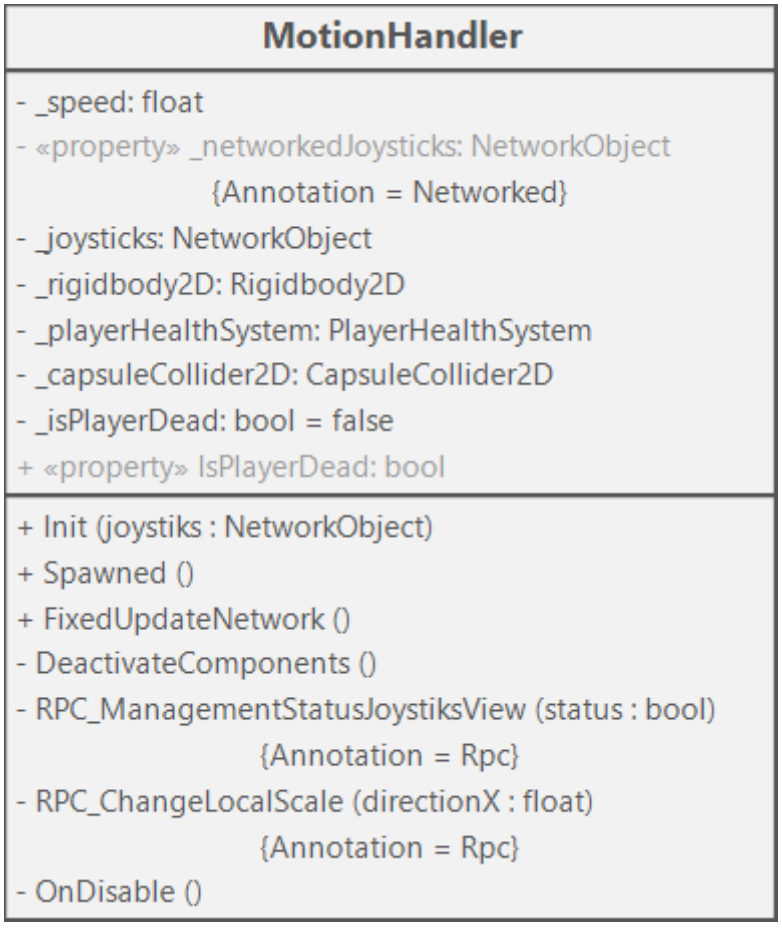

Рисунок 2.14 - Клас MotionHandler

Клас MotionHandler відповідальний за рух персонажа та відсилання даних на сервер для синхронізації даних між гравцями.

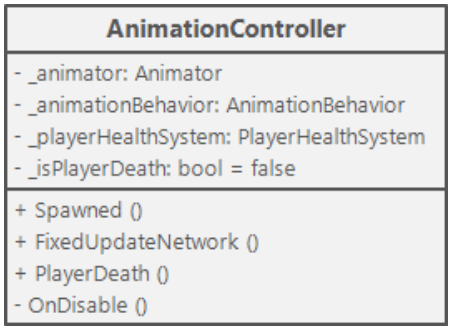

Рисунок 2.15 - Клас AnimationController

Компонент AnimationController призначений за переключення станів анімації персонажа. Персонаж включає три стани анімації, а саме:

- Idle – Стан очікування
- Run – Стан бігу
- Death – Стан смерті

Рисунок 2.16 - Клас WeaponController

WeaponController відповідальний за управління всіма аспектами роботи зброї в грі. Він включає в себе логіку для стрільби, перезарядки, зміни зброї та обробки взаємодій між гравцем і зброєю.

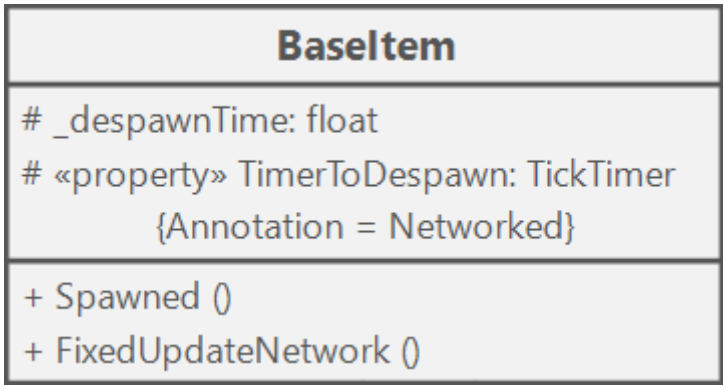

Рисунок 2.17 - Клас BaseItem

BaseItem - це абстрактний клас, який відповідає за представлення та обробку базових атрибутів і методів предметів, які гравець може використовувати під час гри. Клас визначає спільний інтерфейс і основний функціонал, який можуть успадковувати й розширювати інші класи для створення конкретних типів предметів, таких як зброя, аптечки, патрони та інші підбирані предмети.

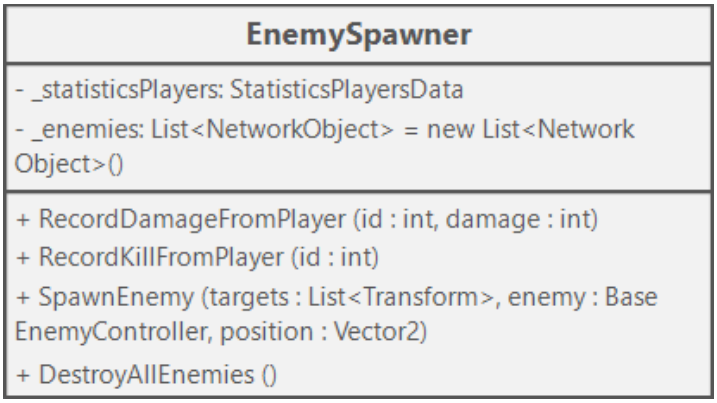

Рисунок 2.18 - Клас EnemySpawner

Клас EnemySpawner, який займається генерацією ворогів у визначених місцях на карті і відповідно видаленням всіх ворогів, коли хвиля підійшла до завершення.

Рисунок 2.19 - Клас WaveController

Клас WaveController відповідає за управління хвилями ворогів у грі. Він забезпечує запуск і завершення хвиль, створення ворогів і предметів у заданих місцях, а також керування станом гри в залежності від кількості живих гравців.

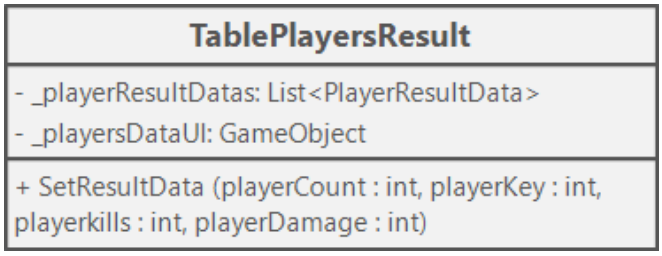

Рисунок 2.20 - Клас TablePlayersResult

Клас TablePlayersResult відповідає за відображення результатів гравців в інтерфейсі користувача після завершення ігрової сесії. Він приймає дані про результати гравців, ініціалізує та оновлює елементи інтерфейсу відповідно до отриманих даних.

Після детального розгляду основних класів ігрового додатку, ми готові перейти до побудови діаграми класів. Діаграма класів є важливим інструментом в процесі проектування програмного забезпечення, оскільки дозволяє візуалізувати структуру системи та взаємозв'язки між її компонентами.

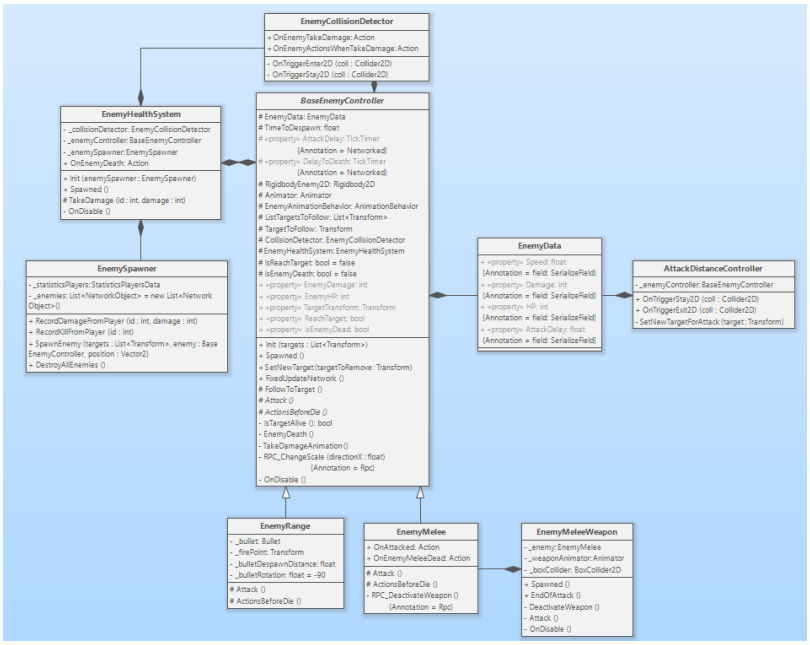

Рисунок 2.21 — Діаграма класів Enemy

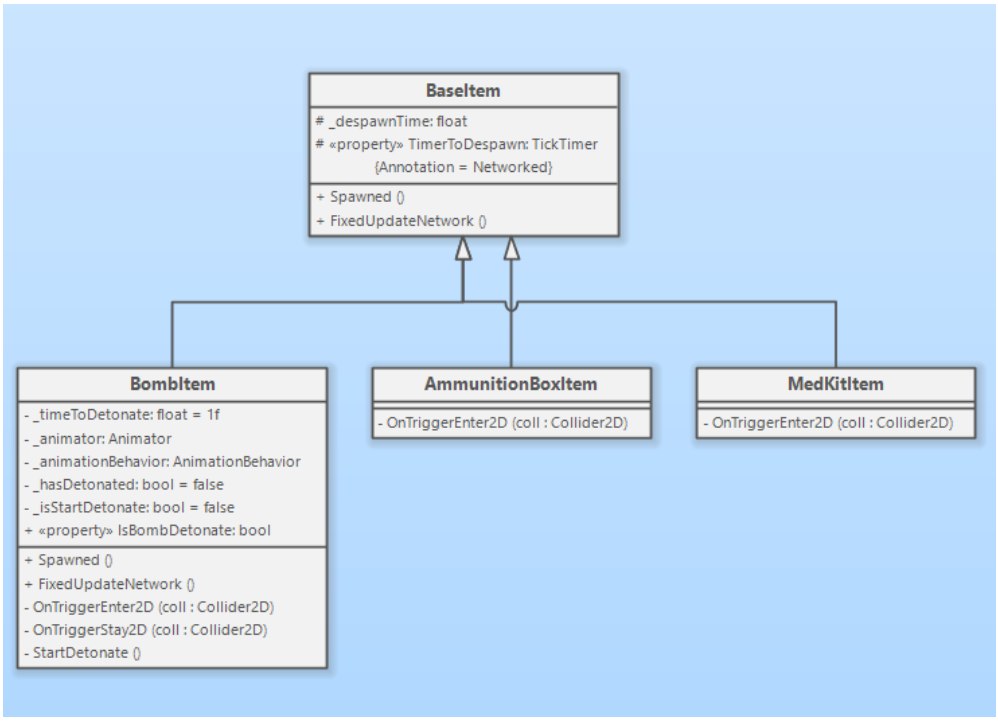

Рисунок 2.22 — Діаграма класів Item

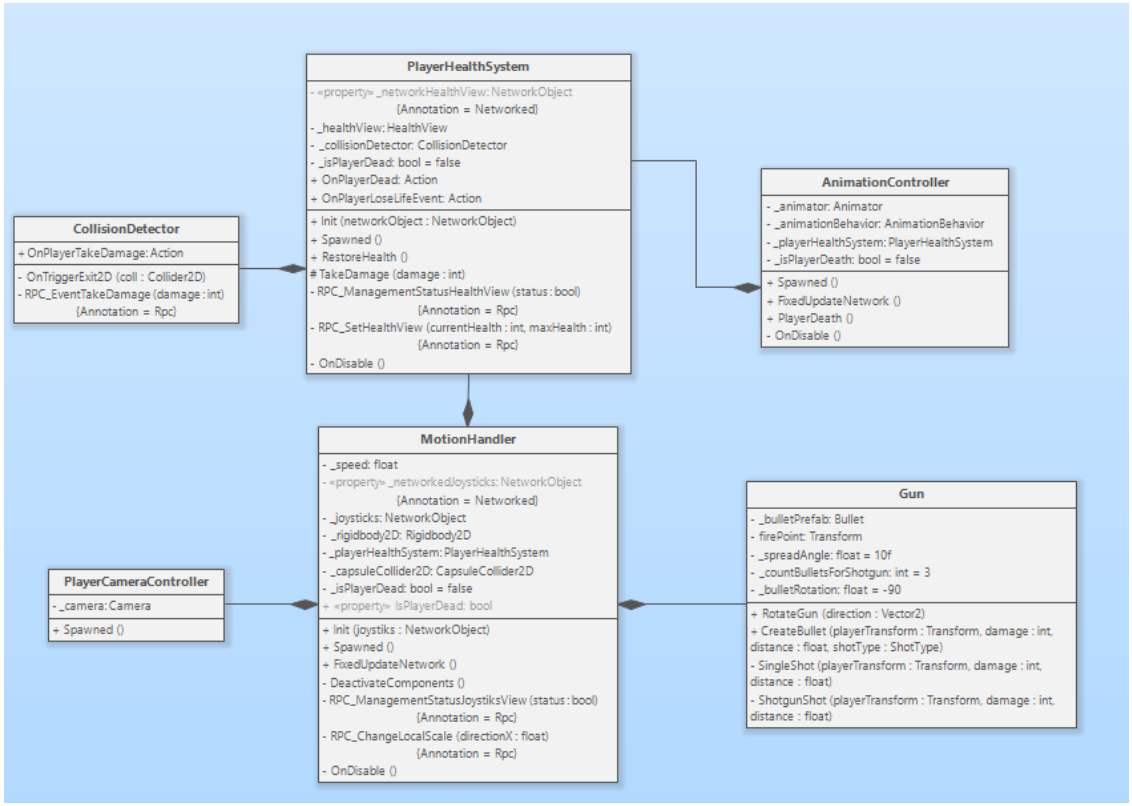

Рисунок 2.22 — Діаграма класів Player

На представлених діаграмах класів відображені взаємозалежності між основними класами ігрового додатку, що відіграють ключову роль у функціонуванні гри. Кожен клас виконує свої унікальні завдання, забезпечуючи реалізацію різноманітних механік та функцій, необхідних для повноцінної роботи додатку.

Детальний розгляд кожного класу та їх взаємодій підтверджує, що додаток включає всі необхідні елементи для створення повноцінного продукту. Класи добре інтегровані між собою, що дозволяє забезпечити злагоджену роботу гри. Це є важливим аспектом для подальшого розвитку та підтримки додатку, оскільки наявність чіткої архітектури значно полегшує процес внесення змін та розширення функціональності в майбутньому.

Таким чином, дані діаграми класів демонструють, що розроблений додаток має всі необхідні компоненти для створення якісного та стабільного ігрового продукту, що відповідає вимогам сучасного ринку.

РОЗДІЛ 3. ТЕСТУВАННЯ ІГРОВОГО ДОДАТКУ

3.1 Тестування основного функціоналу ігрового додатку

Тестування є критично важливим етапом у розробці програмного забезпечення, оскільки воно забезпечує виявлення та усунення помилок, підвищує якість продукту і гарантує його надійність та стабільність. Тестування включає в себе різні методи та підходи, що допомагають перевірити функціональність, продуктивність та зручність використання додатку.

Для ігрових додатків, зокрема, тестування має особливе значення. Гра є інтерактивною системою з численними компонентами, такими як графіка, механіки, звукові ефекти та мережеві функції. Всі ці компоненти повинні працювати злагоджено, щоб забезпечити захоплюючий та безперебійний ігровий досвід. Помилки або недоліки в одній з цих областей можуть значно знизити задоволеність користувачів та вплинути на репутацію гри.

Оскільки наша гра є багатокористувацькою і водночас є кросплатформенною, тестування потребує особливого підходу. Основна частина тестування буде проводитися шляхом створення білду гри та запуску його на різних пристроях і емуляторах. Це дозволить нам перевірити сумісність, продуктивність та стабільність гри в умовах, наближених до реальних.

Для початку можна розпочати тестування на персональному комп’ютері, щоб виявити, чи все працює злагоджено. Щоб протестувати гру на ПК, або мобільному пристрої достатньо скористатись запуском самої гри в Editor, налаштувавши деякі параметри у вкладці «Game», проте так як

наша гра не націлена на Single-player [10], а є багатокористувацькою, цього буде недостатньо.

Щоб вирішити це питання можна скористатись вбудованим інструментом Unity, а саме Build and Run. Цей інструмент дозволяє швидко створити білд гри та автоматично запустити його на локальному комп'ютері [2].

Процес виглядає наступним чином:

- Перейти до вкладки File > Build Settings
- У вікні Build Settings виберіть платформу PC
- Натисніть Build and Run
- Виберіть місце для збереження білду та дочекайтеся завершення процесу.

Отож, на разі важливо перевірити, чи весь важливий функціонал працює коректно, тому для початку почнемо з тестування вибору персонажа.

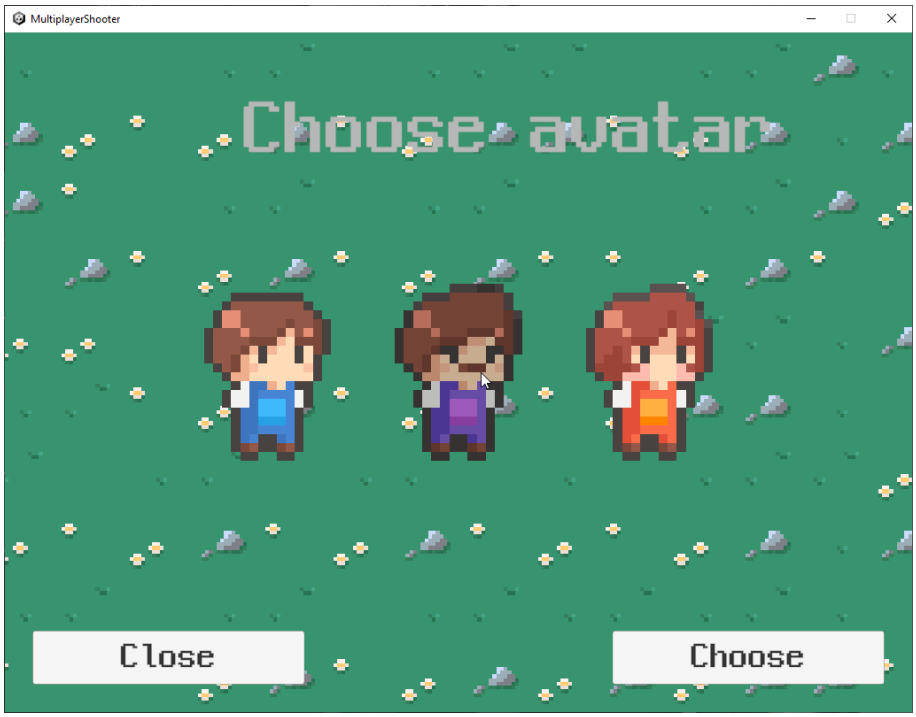

Рисунок 3.1 — Інтерфейс вибору персонажа

Як відображено на рисунку 3.1, було обрано другого по черзі персонажа серед трьох можливих, тепер протестуємо, як це відображається у сесії гри.

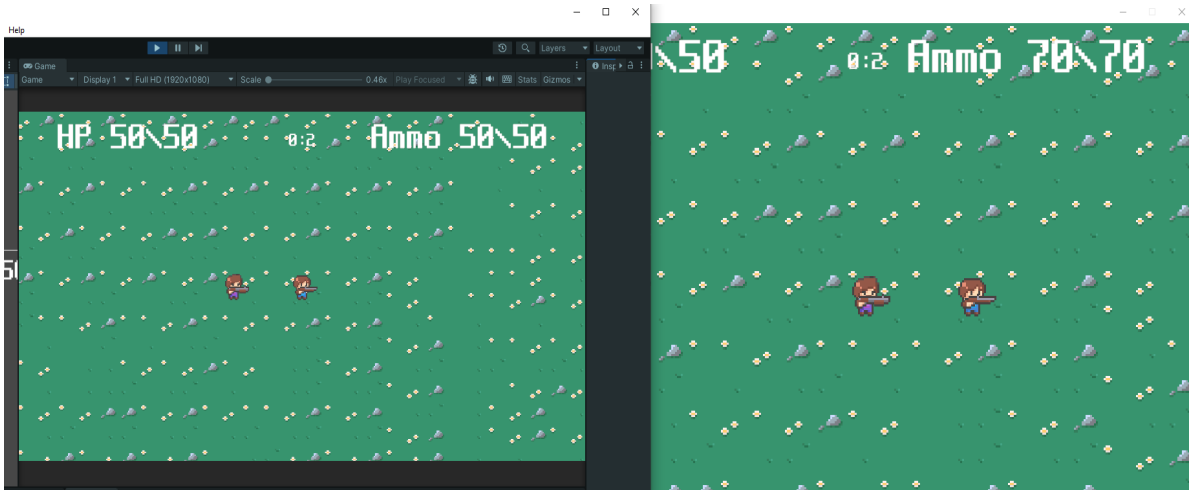

Рисунок 3.2 — Тестування ігрового додатку на ПК

З рисунку 3.2 ми можемо побачити, що вибраний персонаж з’явився у сесії і все даний момент працює коректно. Крім цього можна додати, що логіка зі зброєю відповідає вимогам, а саме у кожного гравця різна зброя.

Перед тим як перейти до тестування на мобільні пристрої, спочатку перевіримо на персональному комп’ютері, чи коректно працює система здоров’я і куль, цей аспект відіграє важливу роль в ігровому додатку, адже помилки в логіці даних систем є недопустимими, так як напряму впливає на процес гри.

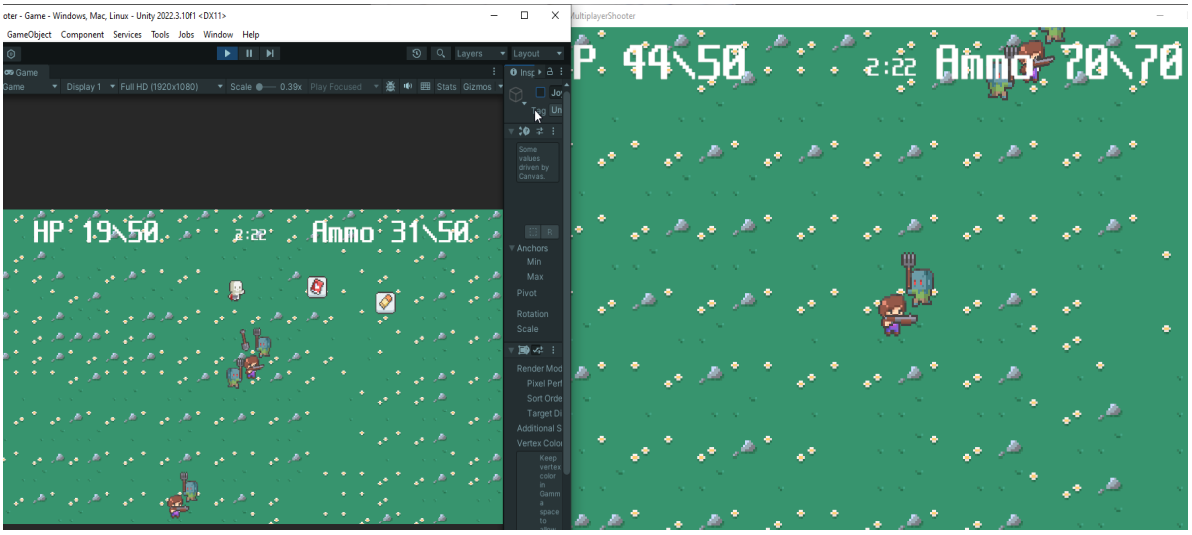

Рисунок 3.3 — Тестування системи здоров’я та куль

З огляду рисунка 3.3 можна зробити висновок, що логіка працює коректно, при витраті боєприпасів зменшується їхня кількість, при зіткненні з противником здоров’я зменшується, як було і задумано. Щоб впевнитись у логіці зіткнення куль з ворогами і відповідно підрахунку вбивств, і завданій їм шкоди варто оглянути таблицю лідерів див. на рисунок 3.4.

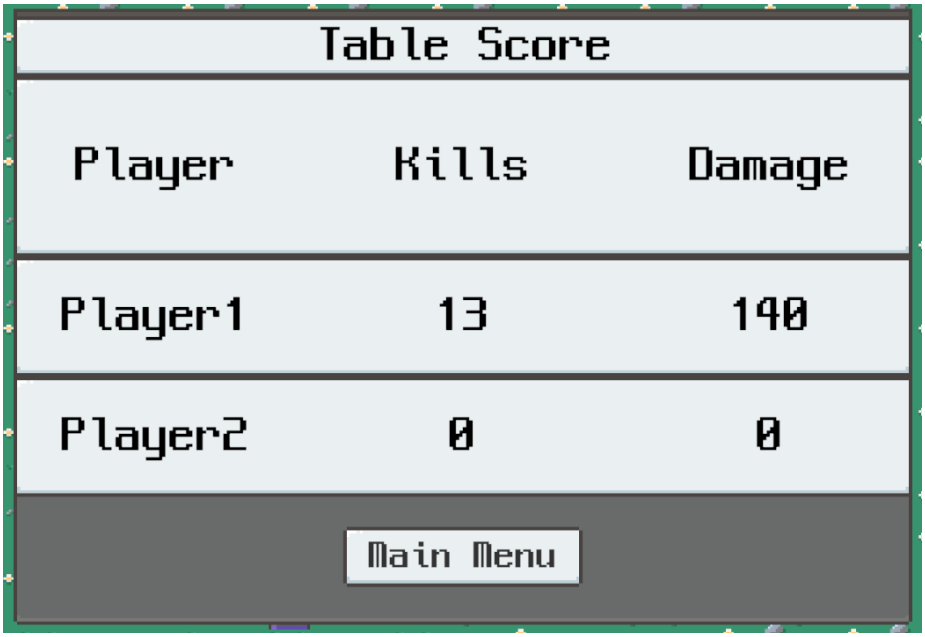

Рисунок 3.4 — Інтерфейс таблиці лідерів ігрового додатку

На рисунку 3.4 відображено систему лідерів сесії, з чого можемо зробити підсумок і затвердити, що підрахунок кількості вбитих ворогів та нанесеної шкоди коректно зберігається та відображається після закінчення гри.

Провівши тестування на ПК, наступним кроком буде перейти до мобільних пристроїв.

Тестування ігрового додатку на мобільних пристроях є більш складним процесом, оскільки включає перевірку роботи на різних моделях смартфонів. Для цього ми будемо використовувати інструменти, такі як Unity Remote та емулятори мобільних пристроїв.

Unity Remote - це інструмент, який дозволяє тестувати гру на мобільному пристрої без необхідності створення білду. За його допомогою можна швидко перевірити зміни в проєкті, запускаючи гру безпосередньо з Unity Editor на вашому смартфоні [11]. Процес виглядає наступним чином:

1. Встановлення додатку Unity Remote 5 з Google Play Store або Apple App Store на мобільний пристрій.
2. Підключіть мобільний пристрій до ПК за допомогою USB-кабелю
3. У проєкті Unity перейти до Edit > Project Settings > Editor
4. У розділі Device обрати підключений мобільний пристрій
5. Запустити додаток Unity Remote 5 на мобільному пристрої
6. В Unity Editor натиснути кнопку Play

Попіклувавшись про допоміжний інструмент для тестування на власному мобільному пристрої необхідно врахувати, що гра є багатокористувацькою і потрібно використати інструмент, який вирішить це питання. В цих цілях ми використаємо емулятор, на разі є багато безкоштовних додатків, які послужать для тестування, але в даному випадку ми використаємо NoxPlayer [12].

Також варто врахувати, що на мобільних пристроях можуть виникнути різного роду помилки, для того щоб виявити, що це за помилка і дізнатись звідки вона виникла ми використаємо Logcat [13].

Logcat - це інструмент для перегляду логів на пристроях Android. Дана утиліта дозволяє отримувати детальну інформацію про поведінку додатка під час його виконання, включаючи помилки, попередження, інформаційні повідомлення та налагоджувальні записи [13].

Врахувавши всі фактори ми можемо розпочати тестування на пристроях Android. В даному випадку демонстрація тестування буде проводитись в Editor ігрового рушія Unity та емуляторі NoxPlayer [12], щоб відобразити загальну картину продуктивності та функціональності гри на платформі Android.

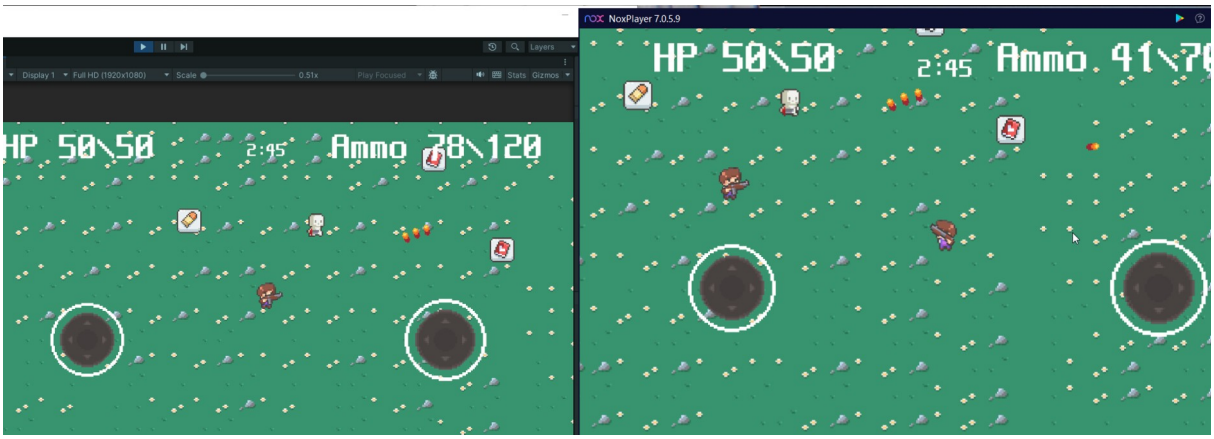

Рисунок 3.5 — Тестування ігрового додатку на платформі Android

Як можна помітити кулі на двох пристроях відображаються коректно, системи, які були протестовані на ПК працюють без ніяких проблем, на етапі тестування ніяких помилок виявлено не було.

Щоб остаточно впевнитись в коректності застосунку на мобільних пристроях, оглянемо таблицю лідерів. Це дозволить нам переконатися в

правильному відображенні та функціонуванні однієї з ключових функцій гри – рейтингової системи.

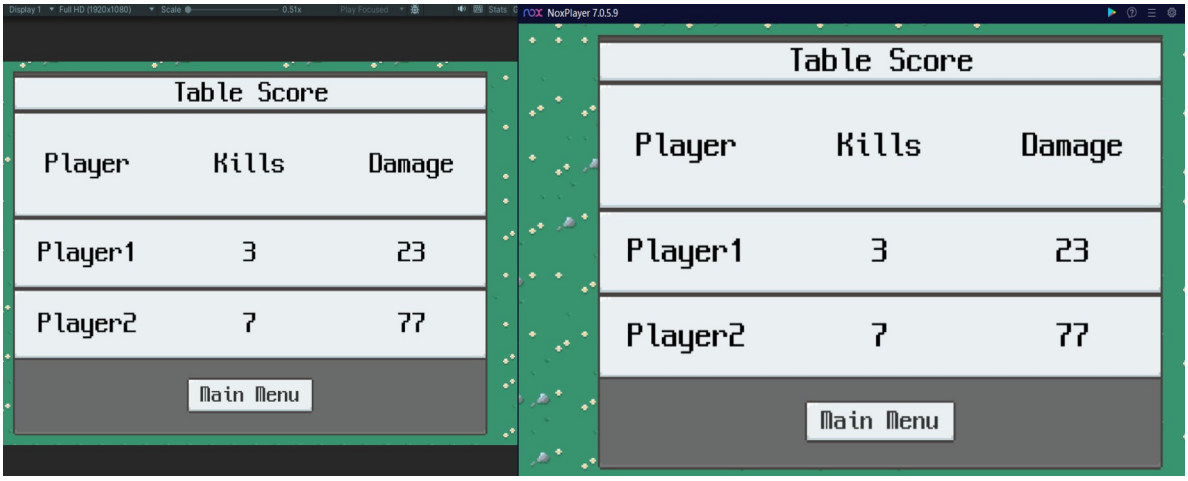

Рисунок 3.6 — Тестування ігрового додатку на платформі Android

Отож, базуючись на останньому продемонстрованому тесті можна зробити висновок, що на двох клієнтах все відображається вірно, дані про статистику гравців записуються і функціонал відповідає початковим вимогам.

3.2 Підсумки до третього розділу. Перспективи розвитку продукту

Детально розглянувши процес тестування ігрового додатку на різних платформах, в результаті чого отримали повну картину роботи гри в різних умовах. Тестування на різних пристроях всіх основних аспектів як: вибір персонажів, хвилі ворогів, різні види зброї та підбираних предметів, справн ворогів, систему здоров’я, таблицю лідерів, систему зіткнення, а також

механізми підключення гравців до сесій, дало зрозуміти, що функціонал додатку працює коректно і всі вимоги поставлені у проєктуванні були дотримані.

Проте незважаючи на це ігровий додаток повинен розвиватись, бо з плином часу без нового функціоналу аудиторії може набриднути грати в продукт, який залишився стояти на місці, особливо, якщо ми говоримо про багатокористувацьку гру, в якій є безліч способів себе вдосконалити та принести щось нове, задля запобігання цьому ми розглянемо можливі перспективи розвитку ігрового додатку. Для цього я наведу декілька способів, які допоможуть доповнити гру, а саме:

- Додавання нових ігрових режимів та завдань для збільшення інтересу користувачів.
- Збільшення пулу зброї, ворогів, карт, предметів, персонажів, щоб урізноманітнити стандартний режим гри.
- Впровадження додаткових соціальних функцій, таких як можливість створення та приєднання до кланів, організація турнірів.
- Розробка спеціальних версій гри для пристроїв з різними технічними характеристиками, щоб забезпечити максимально плавну гру на всіх платформах.
- Впровадження механізмів автоматичного налаштування графіки в залежності від потужності пристрою.
- Поліпшення механізмів захисту персональних даних користувачів.
- Впровадження нових моделей монетизації, таких як підписки, преміум-акаунти та рекламні інтеграції.
- Забезпечення балансування між монетизацією та задоволенням користувачів від гри.

Підсумовуючи даний пункт можна впевнено сказати, що до даного списку можна віднести ще купу різних покращень, які б сприяли поліпшенню ігровим процесом, але наведені вище покращення можуть стати стартом для залучення більшої кількості гравців.

РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Психологічні чинники небезпеки.

Під час надзвичайних ситуацій є критично важливим зберігати спокій, психологічний стан людини в таких моментах є ключовим аспектом для зменшення рівня небезпеки.

Для ефективних дій у складних і надзвичайних обставинах, таких як аварії та катастрофи, необхідно змінити психологічний підхід та рівень підготовки людини. Коли людина опиняється у нестандартній ситуації, яка вимагає негайних рішень, люди часто стають безпорадними перед обставинами. Вони неспроможні вирішувати навіть найпростіші, але важливі для їхнього життя та здоров'я (а також для оточуючих) проблеми [14].

Якщо під час надзвичайної ситуації людина зазнала психічної шкоди, то необхідно надати першу(ППД) або екстрену психологічну допомогу(ЕПД), нижче наведено детальний опис цих значень.

Екстрена психологічна допомога(ЕПД) - це невідкладна допомога, яка надається людині у кризовій ситуації або після виникнення події, що призвела до психологічного стресу. Це може бути надання емоційної підтримки, допомога у заспокоєнні, виявлення потреб та встановлення контакту з професійними службами, якщо це необхідно.

Перша психологічна допомога(ППД) - це перші дії, які людина може виконати для надання допомоги іншій людині, яка переживає емоційний або психологічний стрес. Це може включати в себе прослуховування,

вираження співчуття, допомогу у заспокоєнні та направлення на професійну психологічну допомогу, якщо це необхідно.

Кризова ситуація - це непередбачувана подія або стан, який викликає в людини значний психологічний стрес та неспроможність самостійно впоратися з ситуацією. Кризові ситуації можуть бути спричинені травмою, втратою, конфліктом, природними катастрофами, насильством тощо.

Тепер розглянемо різницю між екстреною та першою психологічною допомогою у вигляді таблиці:

Таблиця 4.1 — Різниця між ЕПД і ППД

Критерій	ЕПД	ППД
Час надання допомоги	Невідкладна, у момент виникнення кризової ситуації	Перші дії, найчастіше у момент після події, що спричинили стрес
Спрямованість	Орієнтована на стабілізацію та заспокоєння особи в кризовій ситуації	Орієнтована на виявлення потреб та підтримку в емоційному стані
Характер допомоги	Емоційна підтримка, спілкування, надання інформації про доступні ресурси	Емоційна підтримка, активне слухання, вираження співчуття, навігація в ресурсах
Професійна	Може здійснюватися	Може

участь	професійними психологами, соціальними працівниками та медичним персоналом	здійснюватися непрофесійними особами, колегами, друзями
Орієнтованість на дії	Направлена на дії, що сприяють заспокоєнню та стабілізації	Направлена на дії, що підтримують особу та допомагають управляти емоціями

4.2 Вимоги до профілактичних медичних оглядів для працівників ПК.

Програмісти, що працюють з ПК протягом тривалого часу, можуть мати проблеми зі здоров'ям, такі як серцево-судинні захворювання, головні болі, та болі в суглобах. Тому для працівників, які використовують комп'ютери, обов'язкові медичні огляди, встановлені в Державних санітарних правилах і нормах роботи з візуальними дисплейними терміналами електронно-обчислювальних машин (ДСанПіН 3.3.2.007-98) [15].

Працівники, що використовують персональні комп'ютери у своїй роботі, підлягають обов’язковим медичним оглядам. Водночас, згідно з роз'ясненням МОЗ від 20.01.2006 р. № 05.0101-18-58/21, обов'язкові медогляди поширюються на роботу з комп'ютерами на основі електронно-променевих трубок та не поширюються на роботу з рідкокристалічними

терміналами. Обов'язковість медичних оглядів для працівників, які працюють за електронно-обчислювальними машинами, передбачена розділом 6 Державних санітарних правил і норм роботи з візуальними дисплейними терміналами електронно-обчислювальних машин ДСанПіН 3.3.2.007-98, затверджених постановою Головного державного санітарного лікаря України від 10.12.98 р. № 7 (далі — ДСанПіН № 7).

Для працівників, що працюють з візуальними дисплейними терміналами електронно-обчислювальних машин, включаючи як колективно використовувані, так і персональні комп'ютери, медичні огляди є обов'язковими. Вони проводяться перед прийомом на роботу та періодично під час трудової діяльності згідно з вимогами Порядку проведення медичних оглядів певних категорій працівників, затвердженого Міністерством охорони здоров'я від 21.05.2007 р. № 246 [16].

Медичні огляди повинні проводитися раз на два роки і включати консультації з терапевтом, невропатологом та офтальмологом. За потреби можуть бути залучені лікарі інших спеціальностей. Метою цих оглядів є своєчасне виявлення захворювань, впливу загальних та професійних факторів праці, постійний контроль за станом здоров'я в умовах шкідливих та небезпечних виробничих факторів, а також розробка програм оздоровлення та реабілітації працівників, які належать до групи ризику.

Медичні огляди також спрямовані на оцінку придатності працівників. Вони включають перевірку гостроти зору, реакції, адаптації та стану ока, а також загальний стан організму. Остаточне рішення про придатність до роботи приймається індивідуально, враховуючи особливості працівника, умови праці та результати додаткових медичних оглядів.

Хронічні форми психічних захворювань, захворювання ендокринної системи, серйозні проблеми з бронхіальною системою, гіпертонічна

хвороба III стадії та інші захворювання є протипоказаннями для роботи з комп'ютером. Під час медичного огляду роботодавець повинен зберігати робоче місце та заробітну плату працівника. Після огляду роботодавець повідомляє працівника про можливість продовжувати або припинити роботу.

Періодичні медичні огляди повинні проводитися раз на два роки комісією у складі терапевта, невропатолога та офтальмолога. До складу комісії, що проводить попередні та періодичні медичні огляди, за наявності медичних показань, можуть залучати лікарів інших спеціальностей. Основними критеріями оцінки придатності до роботи з ВДТ ЕОМ і ПЕОМ є показники стану органів зору: гострота зору, показники рефракції, акомодації, стану бінокулярного апарату ока тощо. Також враховується загальний стан організму.

ВИСНОВКИ

В кінцевому результаті виконання даної кваліфікаційної роботи було здійснено дослідження та розробку адаптивного та мультиплатформенного ігрового додатку з використанням технології Unity.

Результатом цієї роботи стала функціональна гра, яка втілює концепцію жанру Shoot 'em up та використовує переваги Unity для забезпечення найкращого результату на виході, а також інтеграція багатокористувацького режиму за допомогою Photon Fusion.

Вибір жанру Shoot 'em up було обґрунтовано простотою розробки та меншими вимогами до апаратного забезпечення порівняно з FPS, що дозволило зосередитися на динамічному геймплеї та мультиплатформенності. Цей жанр також має довгу історію і популярність серед гравців, що забезпечило додатковий інтерес до проекту.

Unity було обрано за його переваги, такі як підтримка різних платформ, потужні інструменти для розробки, активна спільнота та наявність великої кількості навчальних ресурсів. Це забезпечило ефективний процес розробки та високу якість кінцевого продукту. Unity надає можливість легко інтегрувати анімації, звукові ефекти та інші елементи, що підвищують занурення гравця в ігровий процес.

Була створена гра з трьома типами ворогів та зброї, кількома типами підбираних предметів та трьома хвилями ворогів. Особливу увагу було приділено реалізації багатокористувацького режиму, де гравці можуть вибирати персонажів та співпрацювати для досягнення спільної мети. Проектування ігрового процесу включало створення детальних сценаріїв

поведінки ворогів, збалансованість ігрових механік та розробку інтуїтивного інтерфейсу користувача.

Використання Photon Fusion для реалізації багатокористувацького режиму дозволило забезпечити стабільну та ефективну мережеву взаємодію, що є критичним для сучасних ігор.

Таким чином, проведена робота демонструє можливості розробки високоякісних ігор за допомогою сучасних технологій та інструментів. Використання Unity та Photon Fusion дозволило створити ігровий додаток, який може бути основою для подальшого розвитку та вдосконалення. Цей проект підкреслює важливість поєднання творчості, технічних навичок та сучасних інструментів розробки для досягнення високих результатів у сфері геймдизайну та розробки ігор.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Brief history of video games and their design: From Pong to virtual reality [Електронний ресурс] – Режим доступу до ресурсу: https://www.redbull.com/us-en/history-of-video-games-design.

2. Unity [Електронний ресурс] – Режим доступу до ресурсу: https://unity.com/.

3. Unreal Engine [Електронний ресурс] – Режим доступу до ресурсу: https://www.unrealengine.com/en-US.

4. Godot [Електронний ресурс] – Режим доступу до ресурсу: https://godotengine.org/.

5. CRYENGINE [Електронний ресурс] – Режим доступу до ресурсу: https://www.cryengine.com/.

6. Shoot 'em up [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Shoot_%27em_up.

7. First-person shooter [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/First-person_shooter.

8. Photon Fusion [Електронний ресурс] – Режим доступу до ресурсу: https://www.photonengine.com/fusion.

9. Unity Hub [Електронний ресурс] – Режим доступу до ресурсу: https://unity.com/unity-hub.

10. Single-player video game [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Single-player_video_game.

11. Unity Remote [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Manual/UnityRemote5.html.

12. NoxPlayer [Електронний ресурс] – Режим доступу до ресурсу: https://www.bignox.com/.

13. Logcat [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Packages/com.unity.mobile.android-logcat@0.1/manual/index.html.

14. Безпека життєдіяльності, основи охорони праці [Електронний ресурс] – Режим доступу до ресурсу: https://dl.tntu.edu.ua/content.php?cid=299141.

15. Наказ N 246 [Електронний ресурс] – Режим доступу до ресурсу: https://zakon.rada.gov.ua/rada/show/v0007282-98#Text – Дата доступу: 08.05.2023.

16. МОЗ від 21.05.2007 р. № 246 [Електронний ресурс] – Режим доступу до ресурсу: http://search.ligazakon.ua/l_doc2.nsf/link1/RE14113.html – Дата доступу: 08.05.2023.

17. Репозиторій проєкту [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/Sl1ngyr/Multiplayer-Shooter.

ДОДАТКИ

Додаток А – Лістинг коду основних сутностей ігрового додатку

Лістинг 1 – код NetworkSpawner

```
using System.Collections.Generic;
using Fusion;
using Player;
using UnityEngine;
using Player.Weapon;
using Wave;

namespace Services.Network
{
    public class NetworkSpawner : NetworkBehaviour, IPlayerJoined,
IPlayerLeft
    {
        [SerializeField] private NetworkRunner _networkRunner;

        [SerializeField] private List<NetworkObject> _playerPrefab;
        [SerializeField] private List<WeaponData> _weaponDatas;
        [SerializeField] private List<NetworkObject> _playerGuns;

        [SerializeField] private NetworkObject _bulletsView;
        [SerializeField] private NetworkObject _healthView;
        [SerializeField] private NetworkObject _joystiksView;

        [SerializeField] private StatisticsPlayersData _statisticsPlayers;
        [SerializeField] private WaveController _waveController;

        [Networked] private string _playerPrefsSkin { get; set; }
        private ChangeDetector _changeDetector;
        private PlayerRef _player;

        private Dictionary<PlayerRef, NetworkObject> _spawnedCharacters =
```

```
new Dictionary<PlayerRef, NetworkObject>();
        private Dictionary<PlayerRef, NetworkObject> _spawnedWeapons = new
Dictionary<PlayerRef, NetworkObject>();

        private int _maxPlayers = 2;

        public override void Spawned()
        {
            _changeDetector =
GetChangeDetector(ChangeDetector.Source.SimulationState);


RPC_ChangeSkin(PlayerPrefs.GetString(Constants.PLAYER_PREFS_SKIN));
        }

        public override void Render()
        {
            foreach (var change in _changeDetector.DetectChanges(this))
            {
                switch (change)
                {
                    case nameof(_playerPrefsSkin):
                        if (_playerPrefsSkin == "") break;

                        SpawnHandler(_player);
                        _playerPrefsSkin = "";
                        break;
                }
            }
        }

        public void PlayerJoined(PlayerRef player)
        {
            _player = player;
        }
```

```
        public void PlayerLeft(PlayerRef player)
        {
            if (_spawnedCharacters.TryGetValue(player, out NetworkObject
playerObject))
            {
                _networkRunner.Despawn(playerObject);
                _spawnedCharacters.Remove(player);
            }

            if (_spawnedWeapons.TryGetValue(player, out NetworkObject
weaponObject))
            {
                _networkRunner.Despawn(weaponObject);
                _spawnedWeapons.Remove(player);
            }
        }

        private void SpawnHandler(PlayerRef player)
        {
            if (_networkRunner.IsServer)
            {
                SpawnPlayer(player);

                _statisticsPlayers.InitPlayers(player.PlayerId);

                _waveController.Init(player.PlayerId,
_spawnedCharacters[player].transform,
_networkRunner.SessionInfo.PlayerCount);
            }

            if (_networkRunner.SessionInfo.PlayerCount == _maxPlayers)
            {
                _waveController.StartWave();
            }
        }
```

```
        private void SpawnPlayer(PlayerRef player)
        {
            int weaponNumber = Random.Range(0, _weaponDatas.Count);

            NetworkObject gun = _playerGuns[weaponNumber];

            NetworkObject skinPlayer = null;

            foreach (var prefab in _playerPrefab)
            {
                if (prefab.gameObject.name == _playerPrefsSkin)
                {
                    skinPlayer = prefab;
                }
            }

            Vector3 spawnPosition = new Vector3((player.RawEncoded %
_networkRunner.Config.Simulation.PlayerCount) * 3, 1, 0);

            NetworkObject networkGunObject = _networkRunner.Spawn(gun,
spawnPosition, Quaternion.identity, player);

            NetworkObject networkPlayerObject =
_networkRunner.Spawn(skinPlayer, spawnPosition, Quaternion.identity,
player, ((runner, o) =>
            {

o.GetComponent<WeaponController>().Init(_weaponDatas[weaponNumber],
networkGunObject, _bulletsView);
                o.GetComponent<PlayerHealthSystem>().Init(_healthView);
                o.GetComponent<MotionHandler>().Init(_joystiksView);
            }));

            _playerGuns.RemoveAt(weaponNumber);
            _weaponDatas.RemoveAt(weaponNumber);
            _spawnedCharacters.Add(player, networkPlayerObject);
```

```
            _spawnedWeapons.Add(player, networkGunObject);

        }

        [Rpc]
        private void RPC_ChangeSkin(string skinName)
        {
            if (string.IsNullOrEmpty(skinName))
            {
                _playerPrefsSkin = Constants.SKIN_BY_DEFAULT;
            }
            else
            {
                _playerPrefsSkin = skinName;
            }

        }
    }
}
```

Лістинг 2 – код WaveController

```
using System.Collections.Generic;
using System.Linq;
using Enemy;
using Fusion;
using Items;
using Player;
using Services;
using UnityEngine;

namespace Wave
{
    public class WaveController : NetworkBehaviour
    {
        [SerializeField] private List<WaveData> _waveDatas;
        [SerializeField] private Vector2 _spawnPosition;
```

```
        [SerializeField] private TimerWaveController _timerWaveController;
        [SerializeField] private StatisticsPlayersData _statisticsPlayers;
        [SerializeField] private StatisticsPlayersController
_statisticsPlayersController;
        [SerializeField] private ItemSpawner _itemSpawner;

        [Networked] private TickTimer _spawnEnemy { get; set; }
        [Networked] private TickTimer _spawnItems { get; set; }

        [SerializeField] private EnemySpawner _enemySpawner;

        private int _currentWave = 0;
        private int _numberOfLivePlayers;
        private int _maxWave;
        private bool _isRunning = false;

        private Dictionary<int, Transform> _playerTransforms = new
Dictionary<int, Transform>();

        public bool IsWaveLaunch => _isRunning;

        public void Init(int id, Transform transformPlayer, int
numberOfPlayers)
        {
            _numberOfLivePlayers = numberOfPlayers;

            _playerTransforms.Add(id,transformPlayer);


transformPlayer.GetComponent<PlayerHealthSystem>().OnPlayerLoseLifeEvent +=
RemovePlayerTransform;
        }

        public override void Spawned()
        {
```

```
            _timerWaveController.EndWave += ChangeWave;
            _timerWaveController.Init(_waveDatas[_currentWave].Break,
_waveDatas[_currentWave].Duration);

            _maxWave = _waveDatas.Count;

            _spawnEnemy = TickTimer.CreateFromSeconds(Runner,
_waveDatas[_currentWave].DelayToSpawnEnemy);
            _spawnItems = TickTimer.CreateFromSeconds(Runner,
_waveDatas[_currentWave].DelayToSpawnItems);
        }

        public override void FixedUpdateNetwork()
        {
            if (_isRunning)
            {
                if (_spawnItems.ExpiredOrNotRunning(Runner))
                {
                    SpawnRandomItems();
                    _spawnItems = TickTimer.CreateFromSeconds(Runner,
_waveDatas[_currentWave].DelayToSpawnItems);
                }
            }

            if(!_timerWaveController.IsStartWave || !_isRunning) return;

            if (_spawnEnemy.ExpiredOrNotRunning(Runner))
            {
                SpawnRandomEnemy();
                _spawnEnemy = TickTimer.CreateFromSeconds(Runner,
_waveDatas[_currentWave].DelayToSpawnEnemy);
            }
        }

        public void StartWave()
        {
```

```
            _isRunning = true;
        }

        private void RemovePlayerTransform(int id)
        {
            if (_playerTransforms.TryGetValue(id, out Transform value))
            {

value.GetComponent<PlayerHealthSystem>().OnPlayerLoseLifeEvent -=
RemovePlayerTransform;

                _numberOfLivePlayers--;

                _playerTransforms.Remove(id);
            }

            if (_numberOfLivePlayers == 0)
            {
                DeactivateWave();
            }

        }

        private void SpawnRandomEnemy()
        {
            var randomEnemyNumber = Random.Range(0,
_waveDatas[_currentWave].Enemies.Count);

            List<Transform> _transforms =
_playerTransforms.Values.ToList();

            _enemySpawner.SpawnEnemy(_transforms,
_waveDatas[_currentWave].Enemies[randomEnemyNumber],
GetRandomPositionForSpawn());
        }
```

```
        private void SpawnRandomItems()
        {
            var randomItemNumber = Random.Range(0,
_waveDatas[_currentWave].Items.Count);


_itemSpawner.SpawnItems(_waveDatas[_currentWave].Items[randomItemNumber],
GetRandomPositionForSpawn());
        }

        private Vector2 GetRandomPositionForSpawn()
        {
            float positionX = Random.Range(0, _spawnPosition.x);
            float positionY = Random.Range(0, _spawnPosition.y);

            return new Vector2(positionX, positionY);
        }

        private void ChangeWave()
        {
            _currentWave++;

            if (_currentWave == _maxWave)
            {
                DeactivateWave();
                return;
            }

            _timerWaveController.Init(_waveDatas[_currentWave].Break,
_waveDatas[_currentWave].Duration);

            _enemySpawner.DestroyAllEnemies();

            _spawnEnemy = TickTimer.CreateFromSeconds(Runner,
_waveDatas[_currentWave].DelayToSpawnEnemy);
            _spawnItems = TickTimer.CreateFromSeconds(Runner,
```

```
_waveDatas[_currentWave].DelayToSpawnItems);
        }


        private void DeactivateWave()
        {
            if (_playerTransforms != null)
            {
                List<Transform> _players =
_playerTransforms.Values.ToList();

                foreach (var player in _players)
                {

player.GetComponent<PlayerHealthSystem>().OnPlayerDead?.Invoke();
                }
            }

            _isRunning = false;

            _enemySpawner.DestroyAllEnemies();
            _itemSpawner.DestroyAllItems();

            _timerWaveController.RPC_TimerStatusManagement(false);


_statisticsPlayersController.RPC_SetStatisticsPlayersDataToUI(_statisticsPl
ayers.GetPlayersKey(),
                _statisticsPlayers.GetPlayersKills(),
_statisticsPlayers.GetPlayersDamage());
        }

        private void OnDisable()
        {
            _timerWaveController.EndWave -= ChangeWave;
        }
```

```
    }
}
```

Лістинг 3 – код ChangeSkin

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

namespace UI
{
    public class ChangeSkin : MonoBehaviour
    {
        [SerializeField] private List<Button> _skinsButtons;

        [SerializeField] private Button _saveSkin;
        [SerializeField] private Button _closeSkinMenu;

        [SerializeField] private Canvas _startMenu;

        private void SelectSkin()
        {
            var button =
UnityEngine.EventSystems.EventSystem.current.currentSelectedGameObject;
            var skinName =
button.GetComponent<ButtonSkinsDescription>().SkinName;

            PlayerPrefs.SetString(Constants.PLAYER_PREFS_SKIN, skinName);

        }

        private void SaveSkin()
        {
            CloseSkinMenu();
```

```
        }

        private void CloseSkinMenu()
        {
            _startMenu.gameObject.SetActive(true);
            gameObject.SetActive(false);
        }

        private void OnEnable()
        {
            foreach (var button in _skinsButtons)
            {
                button.onClick.AddListener(SelectSkin);
            }

            _saveSkin.onClick.AddListener(SaveSkin);
            _closeSkinMenu.onClick.AddListener(CloseSkinMenu);
        }

        private void OnDisable()
        {
            foreach (var button in _skinsButtons)
            {
                button.onClick.RemoveListener(SelectSkin);
            }

            _saveSkin.onClick.RemoveListener(SaveSkin);
            _closeSkinMenu.onClick.RemoveListener(SaveSkin);
        }

    }
}
```

Лістинг 4– код BaseEnemyController

```
using System.Collections.Generic;
using Enemy.AnimationStates;
using Fusion;
using Player;
using Services;
using UnityEngine;

namespace Enemy
{
    public abstract class BaseEnemyController : NetworkBehaviour
    {
        [SerializeField] protected EnemyData EnemyData;
        [SerializeField] protected float TimeToDespawn;

        [Networked] protected TickTimer AttackDelay { get; set; }
        [Networked] protected TickTimer DelayToDeath { get; set; }

        protected Rigidbody2D RigidbodyEnemy2D;
        protected Animator Animator;
        protected AnimationBehavior EnemyAnimationBehavior;
        protected List<Transform> ListTargetsToFollow;
        protected Transform TargetToFollow;
        protected EnemyCollisionDetector CollisionDetector;
        protected EnemyHealthSystem EnemyHealthSystem;

        protected bool IsReachTarget = false;
        protected bool IsEnemyDeath = false;

        public int EnemyDamage => EnemyData.Damage;
        public int EnemyHP => EnemyData.HP;

        public Transform TargetTransform
        {
            get => TargetToFollow;
            set => TargetToFollow = value;
        }
```

```
        public bool ReachTarget
        {
            get => IsReachTarget;
            set => IsReachTarget = value;
        }

        public bool IsEnemyDead => IsEnemyDeath;

        public void Init(List<Transform> targets)
        {
            ListTargetsToFollow = targets;

            int randomTarget = Random.Range(0, targets.Count);
            TargetToFollow = ListTargetsToFollow[randomTarget];
        }

        public override void Spawned()
        {
            EnemyHealthSystem = GetComponent<EnemyHealthSystem>();
            CollisionDetector = GetComponent<EnemyCollisionDetector>();
            RigidbodyEnemy2D = GetComponent<Rigidbody2D>();
            Animator = GetComponent<Animator>();

            EnemyAnimationBehavior = new
AnimationBehaviorEnemyRun(Animator);
            EnemyAnimationBehavior.Enter();

            CollisionDetector.OnEnemyActionsWhenTakeDamage +=
TakeDamageAnimation;
            EnemyHealthSystem.OnEnemyDeath += ActionsBeforeDie;
        }

        public void SetNewTarget(Transform targetToRemove)
        {
            ListTargetsToFollow.Remove(targetToRemove);
```

```
            var randomTarget = Random.Range(0, ListTargetsToFollow.Count);

            TargetToFollow = ListTargetsToFollow[randomTarget];
        }

        public override void FixedUpdateNetwork()
        {
            if (IsTargetAlive())
            {
                SetNewTarget(TargetToFollow);
            }

            RPC_ChangeScale(TargetToFollow.transform.position.x);
            FollowToTarget();
        }

        protected void FollowToTarget()
        {
            if (IsEnemyDeath && DelayToDeath.ExpiredOrNotRunning(Runner))
            {
                EnemyDeath();
            }

            if(IsEnemyDeath) return;

            if (!IsReachTarget)
            {
                Vector3 direction = (TargetToFollow.transform.position -
gameObject.transform.position).normalized;
                Vector3 positionToMove = transform.position +
(EnemyData.Speed * Runner.DeltaTime * direction);

                RigidbodyEnemy2D.MovePosition(positionToMove);
            }
            else if (IsReachTarget && HasStateAuthority &&
```

```
AttackDelay.ExpiredOrNotRunning(Runner))
            {
                Attack();
                AttackDelay = TickTimer.CreateFromSeconds(Runner,
EnemyData.AttackDelay);
            }
        }

        protected abstract void Attack();
        protected abstract void ActionsBeforeDie();

        private bool IsTargetAlive()
        {
            return
TargetToFollow.GetComponent<MotionHandler>().IsPlayerDead;
        }

        private void EnemyDeath()
        {
            Runner.Despawn(Object);
        }

        private void TakeDamageAnimation()
        {
            if(IsEnemyDeath) return;

            EnemyAnimationBehavior.Exit();
            EnemyAnimationBehavior = new
AnimationBehaviorEnemyHit(Animator);
            EnemyAnimationBehavior.Enter();
        }

        [Rpc]
        private void RPC_ChangeScale(float directionX)
        {
            if (directionX > transform.position.x)
```

```
            {
                RigidbodyEnemy2D.transform.localScale = new Vector2(1, 1);
            }
            else if (directionX < transform.position.x)
            {
                RigidbodyEnemy2D.transform.localScale = new Vector2(-1, 1);
            }
        }

        private void OnDisable()
        {
            CollisionDetector.OnEnemyActionsWhenTakeDamage -=
TakeDamageAnimation;
            EnemyHealthSystem.OnEnemyDeath -= ActionsBeforeDie;
        }

    }
}
```

Додаток Б – Диск