МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ
ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ФІНАНСІВ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА

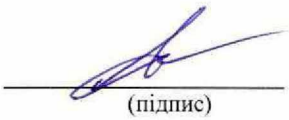
на тему: Розробка кросплатформеного ігрового застосунку у середовищі Unity

Здобувача вищої освіти першого (бакалаврського) рівня,
IV курсу, групи КБ-21-1, спеціальності 122 «Комп'ютерні науки»,
освітня програма «Інформаційні управляючі системи і технології (за галузями)»


(підпис)

Морозова Олександра Володимировича
(прізвище, ім'я, по батькові)

Науковий керівник


(підпис)

Омельчук Антон Анатолійович, к.т.н.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

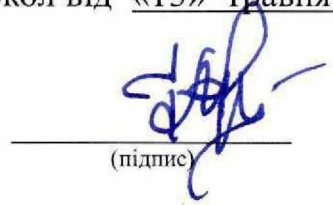
Керівник освітньо-професійної програми


(підпис)

Омельчук Антон Анатолійович, к.т.н.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту за рішенням кафедри комп'ютерних та інформаційних технологій і систем протокол від «13» травня 2025 № 18

Завідувач кафедри


(підпис)

Філоненко Михайло Миколайович, к.ф.-м.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Ірпінь – 2025

Реєстрація №41 23.05.2025 _____
(дата) (підпис відповідальної особи)

Результати перевірки до захисту 23.05.2025 _____
(до захисту, на доопрацювання) (дата) (підпис)

Результат захисту 5 (відмінно) 04.06.2025 _____
(оцінка) (дата) (підпис)

Голова комісії: _____ В.М. Франчук
(підпис)

Члени комісії:

_____ М.М. Філоненко
(підпис)

_____ С.М. Грищенко
(підпис)

_____ А.А. Омельчук
(підпис)

БЛАНК ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Об'єкти контролю	Діапазон оцінки в балах	Фактична кількість балів	Примітки	Дата, П.І.Б.: керівника, голови комісії членів комісії,секретаря, (підписи)
ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ КОМІСІЄЮ ПО ЗАХИСТУ				
1. Зміст випускної роботи (якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	**від 0 до 30**	46		Омельчук А.А. Франчук В.М. Філоненко М.М. Грищенко С.М. Омельчук А.А. Суховецька Л.П.
2. Дотримання вимог щодо оформлення	**від 0 до 10**	9		
3. Вчасне виконання роботи на всіх етапах у встановлені кафедрою терміни	**від 0 до 5**	4		
4. Рецензент	**від 0 до 5**	5		
РАЗОМ БАЛІВ	**від 0 до 50**	46		

Оцінювання кваліфікаційної роботи науковим керівником - 47 балів.

ПЕРЕВЕДЕННЯ ОЦІНКИ ЗА 100-БАЛЬНОЮ ШКАЛОЮ ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалоюЄКТС	Оцінка за національною шкалою	Дата	П.І.Б.: голови, членів комісії, секретаря, (підпис)
93	A	5 (відмінно)	04.06.2025	Франчук В.М. Філоненко М.М. Грищенко С.М. Омельчук А.А. Суховецька Л.П.

ПРАВИЛА ПЕРЕВЕДЕННЯ ОЦІНОК ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою
90-100	A	5 (відмінно)
80-89	B	4 (добре)
70-79	C	
60-69	D	3 (задовільно)
50-59	E	
35-49	FX	2 (незадовільно)

Факультет фінансів та цифрових технологій
Кафедра комп'ютерних та інформаційних технологій і систем
Галузь знань: 12 «Інформаційні технології»
Спеціальність: 122 «Комп'ютерні науки»
Освітня програма: «Інформаційні управляючі системи і технології (за галузями)»

Затверджую
Завідувач кафедри Філоненко М.М.
«14» листопада 2024

Завдання на кваліфікаційну роботу
здобувачу вищої освіти
Морозову Олександру Володимировичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи Розробка кросплатформеного ігрового застосунку у середовищі Unity

Затверджена наказом ректора від «13» листопада 2024 № 1717

2. Строк здачі закінченої роботи «08» травня 2025
3. Мета роботи: розробка кросплатформеного ігрового застосунку в середовищі Unity з акцентом на оптимізацію продуктивності

Об'єкт дослідження: процес розробки кросплатформеного ігрового застосунку

Предмет дослідження технології розробки кросплатформеного ігрового застосунку у середовищі Unity

Орієнтовна база дослідження: аналіз літератури, ринку світу, існуючих рішень та вимог користувачів й розробників.

4. Вихідні дані до роботи Інформаційне середовище ігрової розробки, технології Unity, документацію та матеріали з розробки 2D ігор, системи збереження даних та оптимізації продуктивності.
5. Перелік графічного матеріалу Методичні рекомендації кафедри комп'ютерних та інформаційнних технологій і систем, Adobe Photoshop, Plant UML, Unity Engine EULA, Unity Engine API, Visual Studio.
6. Зміст кваліфікаційної роботи (перелік питань за кожним розділом)

Вступ. Теоретико-методологічні основи розробки кросплатформених ігрових застосунків. Аналіз технологій розробки кросплатформених ігрових застосунків. Визначення особливості розробки 2D застосунку

Дослідження і проектування ігрового застосунку у середовищі Unity.

Розробка й оптимізація кросплатформеного ігрового застосунку. Реалізація й оптимізація ключових елементів додатку. Тестування та впровадження застосунку. Список використаних джерел.

7. Календарний план виконання роботи

№ п/п	Назва етапів кваліфікаційноїроботи	Строк виконання етапів роботи	
		За планом	Фактично
1	2	3	4
1	Вибір теми, підбір літературних джерел, формування структури кваліфікаційної роботи	03.11.2024	29.10.2024
2	**І розділ** (розкрити сутність і стан дослідження наукової проблематики роботи, після чого навести її загальну характеристику; огляд літератури за темою і вибір конкретних напрямів досліджень)	27.12.2024	27.12.2024
3	**ІІ розділ** (аналіз експериментальної частини про методики досліджень)	21.02.2025	19.02.2025
4	**ІІІ розділ** (проведені теоретичні і (або) експериментальні дослідження та проектні рішення)	20.04.2025	20.04.2025
5	**Висновки**	01.05.2025	01.05.2025
6	Доопрацювання роботи з урахуванням зауважень наукового керівника	10.05.2025	10.05.2025
7	Отримання відгуку керівника	27.05.2025	27.05.2025
8	Отримання зовнішньої рецензії	29.05.2025	29.05.2025

8. Дата видачі завдання «14» листопада 2024

9. Керівник кваліфікаційної роботи _____ (підпис) Омельчук А (прізвище, ініціали

10. Керівник освітньо-професійної програми _____ (підпис) Омельчук А (прізвище, ініціали

11. Завдання до виконання отримав здобувач вищої освіти _____ (підпис) Морозов О. (прізвище, ініціали

АНОТАЦІЯ

Мета роботи є розробка кросплатформеного ігрового застосунку в середовищі Unity з акцентом на оптимізацію продуктивності.

Завдання роботи є:

1. Аналіз технологій розробки кросплатформених ігрових застосунків;
2. Визначення особливості розробки 2D застосунку;
3. Створення моделей й архітектури програми;
4. Реалізація й оптимізація ключових елементів додатку;

Предметом дослідження є технології розробки кросплатформеного ігрового застосунку у середовищі Unity

Базою дослідження є аналіз літератури, ринку світу, існуючих рішень та вимог користувачів й розробників.

За результатами дослідження розроблено ігровий кросплатформений застосунок у середовищі Unity.

Одержані результати можуть бути використані для створення ігрового застосунку в освітню, економічну сферу подання інформації користувачам.

Кваліфікаційна робота складається зі вступу (3 сторінки), трьох розділів (40 сторінок), 16 зображень, висновків (2 сторінки), списку використаних джерел (41 джерел), та трьох додатків (12 сторінок).

Ключові слова: кросплатформений застосунок, проєкт, розробка, додаток, ігровий застосунок, програма, програмний продукт.

Результати дослідження представлені на II Всеукраїнської науково-практичної конференції молодих вчених «Математичні методи та цифрові технології в економіці».

Рік виконання кваліфікаційної роботи: 2025.

Рік захисту роботи: 2025.

ANNOTATION

The aim of this work is the development of a cross-platform gaming application in the Unity environment with a focus on performance optimization.

The objectives of the work are:

1. Analysis of technologies for developing cross-platform gaming applications;
2. Identification of the specific features of 2D application development;
3. Creation of models and application architecture;
4. Implementation and optimization of key application elements.

The subject of the research is the technologies for developing a cross-platform gaming application in the Unity environment.

The research is based on the analysis of literature, the global market, existing solutions, and the requirements of users and developers.

As a result of the study, a cross-platform gaming application was developed in the Unity environment.

The obtained results can be used to create a gaming application in the educational and economic spheres for presenting information to users.

The qualification work consists of an introduction (3 pages), three chapters (40 pages), 16 images, conclusions (2 pages), a list of references (41 sources), and three appendices (12 pages).

Keywords: cross-platform application, project, development, application, gaming application, program, software product.

The research results were presented at the II All-Ukrainian Scientific and Practical Conference of Young Scientists "Mathematical Methods and Digital Technologies in Economics."

Year of qualification work completion: 2025.

Year of defense: 2025.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 3
ВСТУП 5
РОЗДІЛ 1. ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ОСНОВИ РОЗРОБКИ КРОСПЛАТФОРМЕНИХ ІГРОВИХ ЗАСТОСУНКІВ 8
1.1. Аналіз технологій розробки кросплатформених ігрових застосунків ... 8
1.2. Особливості розробки 2D застосунків у середовищі Unity 13
1.3. Обґрунтування вибору технологій для реалізації кросплатформеного застосунку Warbound 17
РОЗДІЛ 2. ДОСЛІДЖЕННЯ І ПРОЕКТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ У СЕРЕДОВИЩІ UNITY 22
2.1. Створення моделей й архітектури ігрового застосунку Warbound 22
2.2. Моделі даних та система збереження інформації 27
2.3. Механіки покрокових боїв 33
2.4. Структуризація даних для кросплатформеної реалізації 35
РОЗДІЛ 3. РОЗРОБКА Й ОПТИМІЗАЦІЯ КРОСПЛАТФОРМЕНОГО ІГРОВОГО ЗАСТОСУНКУ 37
3.1. Реалізація ключових елементів ігрового застосунку 37
3.2. Оптимізація продуктивності для різних платформ 42
3.3. Тестування та впровадження застосунку 44
ВИСНОВКИ 47
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 49
ДОДАТКИ 53
Додаток А 53
Додаток Б 59
Додаток В 64

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

2D – двовимірний (two-dimensional)

3D – тривимірний (three-dimensional)

AI – штучний інтелект (Artificial Intelligence)

API – програмний інтерфейс застосунку (Application Programming Interface)

App Store – цифровий магазин застосунків Apple

AWS – Amazon Web Services

C# – C Sharp (мова програмування)

C++ – C Plus Plus (мова програмування)

CI/CD – Continuous Integration/Continuous Deployment (безперервна інтеграція/безперервне розгортання)

CPU – центральний процесор (Central Processing Unit)

DirectX – бібліотека для роботи з мультимедіа

ETC2 – Ericsson Texture Compression 2 (формат стиснення текстур)

FPS – кадрів за секунду (Frames Per Second)

GDScript – мова програмування рушія Godot

GML – GameMaker Language (мова програмування)

Google Play – цифровий магазин застосунків Google

GPU – графічний процесор (Graphics Processing Unit)

HTML5 – HyperText Markup Language 5 (мова розмітки гіпертексту)

iCloud – хмарний сервіс Apple

IDE – інтегроване середовище розробки (Integrated Development Environment)

iOS – операційна система Apple

JSON – JavaScript Object Notation (формат обміну даними)

Linux – операційна система

LOD – Level of Detail (рівень деталізації)

LTS – Long Term Support (довгострокова підтримка)

macOS – операційна система Apple

Metal – графічний API Apple

MMORPG – масова багатокористувацька онлайн-рольова гра (Massively Multiplayer Online Role-Playing Game)

OpenGL ES – Open Graphics Library for Embedded Systems

ПК – персональний комп'ютер

PBR – Physically Based Rendering (фізично коректний рендеринг)

PSD – Photoshop Document (формат файлів Adobe Photoshop)

PVRTC – PowerVR Texture Compression (формат стиснення текстур)

REST – Representational State Transfer (архітектурний стиль)

SDK – Software Development Kit (набір засобів розробки)

SOLID – принципи проєктування програмного забезпечення (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion)

Steam – платформа розповсюдження цифрового контенту

UI – користувацький інтерфейс (User Interface)

Unity – платформа для розробки інтерактивних застосунків

URP – Universal Render Pipeline (універсальний конвеєр рендерингу)

UX – користувацький досвід (User Experience)

VR – віртуальна реальність (Virtual Reality)

WebGL – Web Graphics Library (бібліотека для веб-графіки)

Windows – операційна система Microsoft

XML – eXtensible Markup Language (розширювана мова розмітки)

ВСТУП

У сучасному цифровому світі дедалі складніше сприймати інформацію через текст або історії, написані в наукових чи художніх книгах. Надмірна кількість зображень, текстів та монотонних звуків створює інформаційне перенасичення, що ускладнює концентрацію та розуміння матеріалу. Усе більше людей віддають перевагу коротким відео, однак цей формат також поступово втрачає свою новизну та привабливість, адже довгі відеоролики та стріми, які колись здавалися інноваційними, поступово перестають утримувати увагу глядачів. Люди все частіше шукають глибше занурення в інформацію, хочуть не лише споживати контент, але втрачають свою концентрацію на важливих деталей чи моментах матеріалу.

Одним із найцікавіших напрямків у сучасному цифровому світі є розробка ігрових застосунків. Вони не лише формують розважальний сектор, а й виступають важливим рушієм технологічного бізнесу. Індустрія відеоігор стимулює розвиток інновацій у програмуванні, комп’ютерній графіці та покращенні користувацького досвіду, що впливає на широкий спектр технологічних сфер, від штучного інтелекту до віртуальної реальності.

Актуальність розробки програмного забезпечення для одного пристрою більше не відповідає сучасним вимогам цифрового середовища. Користувачі очікують безперешкодний доступ до застосунків із різних пристроїв, Кросплатформені рішення розширюють можливості використання та оптимізують витрати на розробку, зменшуючи необхідність підтримки окремих версій. Високий попит на такі технології підтверджується тим, що понад 70% мобільних програм створюються на Unity[69], а ринок мобільних застосунків демонструє стабільне зростання з використанням кросплатформеним рішенням. Також інтеграція хмарних технологій та онлайн-сервісів дозволяє користувачам синхронізувати дані між пристроями, що робить кросплатформені підходи не просто перевагою, а необхідністю[11]. Ігнорування цього тренду може призвести до втрати конкурентоспроможності та обмеження потенційного розвитку програмного забезпечення.

Дослідники та розробники здебільшого зосереджують увагу на оптимізації продуктивності[16], системах збереження даних[39, 43], особливостях реалізації кросплатформених функцій[33] та розробці багатокористувацьких механік[21].

Unity – це один із провідних рушіїв для розробки кросплатформених застосунків, що забезпечує підтримку мобільних, настільних, консольних та VR-платформ. Завдяки широкому функціоналу та гнучкості ця технологія активно використовується в ігровій індустрії та поза її межами, сприяючи впровадженню інтерактивних рішень у різних сферах. Unity дозволяє оптимізувати процес розробки, забезпечити стабільну продуктивність на різних пристроях та інтегрувати сучасні підходи до збереження даних та взаємодії з користувачами. Вибір цього рушія має бути спрощеним до мобільних, комп'ютерних платформ, оптимізуючи витрати на розробку та підтримку.

Розробка застосунку з покроковою тактичною системою у стилі фентезі, яка ставить перед собою завдання забезпечити стратегічну глибину між гравцями. Вплив таких проєктів, як Slay the Spire, Albion Online, Balatro, Hearthstone, формує концепцію командних боїв, що потребує продуманої механіки та технологічної точності. Кросплатформеність у цьому жанрі забезпечує доступність для ширшої аудиторії та інтеграцію між пристроями, дозволяючи користувачам взаємодіяти без обмежень між собою.

Метою кваліфікаційної роботи є розробка кросплатформеного ігрового застосунку в середовищі Unity з акцентом на оптимізацію продуктивності, інтеграцію з базами даних та реалізацію механік, що підвищують якість багатокористувацькою тактичною системою.

Для досягнення мети необхідно вирішити такі завдання:

1. Проаналізувати сучасні технології розробки кросплатформених ігрових застосунків та обґрунтувати вибір Unity як основного ігрового рушія.
2. Дослідити особливості розробки 2D ігор у середовищі Unity та системи збереження даних користувачів.
3. Розробити архітектуру ігрового застосунку, визначити його основні механіки та функціонал.

4. Реалізувати систему покрокових боїв та взаємодії між гравцями у реальному часі.
5. Інтегрувати систему збереження даних для забезпечення доступу з різних пристроїв.
6. Оптимізувати продуктивність застосунку для забезпечення плавної роботи на різних платформах.
7. Протестувати функціональність застосунку на різних платформах.

Об'єктом дослідження є процес розробки кросплатформеного ігрового застосунку.

Предметом дослідження є технології розробки кросплатформеного ігрового застосунку у середовищі Unity.

Методи дослідження охоплюють теоретичний аналіз літературних джерел і документації, емпіричне тестування програмного забезпечення з оцінкою продуктивності та порівняльний аналіз підходів до реалізації кросплатформених функцій для вибору оптимального рішення.

База досліджень включає інформаційне середовище ігрової розробки, технології Unity, документацію та матеріали з розробки 2D ігор, системи збереження даних та оптимізації продуктивності.

Науково-практична значимість дослідження полягає у створенні кросплатформеного застосунку, що демонструє сучасні методи розробки інтерактивного контенту. Отримані результати можуть бути використані для вдосконалення механік та оптимізації процесу розробки. Створена модель слугує основою для подальшого розвитку та впровадження нових функцій.

Результати дослідження представлені на II Всеукраїнської науково-практичної конференції молодих вчених «Математичні методи та цифрові технології в економіці»[10].

РОЗДІЛ 1. ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ОСНОВИ РОЗРОБКИ КРОСПЛАТФОРМЕНИХ ІГРОВИХ ЗАСТОСУНКІВ

1.1. Аналіз технологій розробки кросплатформених ігрових застосунків

Сучасний світ стрімко рухається шляхом цифрової інтеграції, де користувачі вже не прив'язані до єдиного пристрою. Щодня мільйони людей змінюють свої смартфони на ноутбуки і навпаки, очікуючи, що їхній досвід взаємодії з програмами залишиться незмінним. Однак розробка окремих версій застосунків для кожної платформи стала непрактичною — вона вимагає значних ресурсів, подвоює зусилля команд та ускладнює підтримку й визнає незручність користувачів. Кросплатформена розробка пропонує вирішення цих проблем, дозволяючи створювати програмні продукти, що працюють однаково ефективно на різних пристроях. Попри очевидні переваги, реалізація таких застосунків ставить перед розробниками специфічні виклики, пов'язані з продуктивністю, користувацьким досвідом та обмеженнями різних операційних систем.

Користувачі взаємодіють з цифровим контентом через різноманітні інтерфейси – зазвичай від персональних комп'ютерів та ноутбуків до планшетів, смартфонів та спеціалізованих пристроїв, таких як ігрові консолі та пристрої віртуальної реальності. Як свідчать дослідження українських науковців з Національного технічного університету України Грищенка й Потапової – «впровадження кросплатформених підходів до розробки програмного забезпечення дозволяє знизити витрати на розробку та підтримку на 40-60% порівняно з традиційним підходом створення окремих програмних продуктів для кожної цільової платформи»[4].

Стрімкий розвиток інформаційного суспільства та цифрових технологій сформував нову парадигму розробки програмного забезпечення, в межах якої кросплатформеність стала критичною необхідністю для успішної конкуренції на ринку інтерактивних продуктів. Кросплатформені програмні продукти функціонують на різних обчислювальних платформах з ідентичною

функціональністю та користувацьким досвідом. Технологічний простір кросплатформеної розробки характеризується різноманіттям підходів, інструментів та середовищ, що дозволяють реалізувати цю концепцію з різним ступенем ефективності та з різними інвестиційними витратами[2].

Кросплатформені програми змінили користувацький досвід, дозволивши користувачам отримувати доступ до програм з численних пристроїв та операційних систем. Методологічною основою кросплатформеної розробки виступає принцип "розробки єдиного коду для різних платформ". За результатами опитування агентства соціальних мереж OST Agency, витрати на розробку бізнес-додатків за 2024 рік, середня вартість розробки одного готового нативного додатка (iOS або Android) коливається від 50 000 до 300 000 доларів США. Створення двох нативних додатків окремо може збільшити витрати на 30-50% та понад 42% компаній зараз вибирають кросплатформені фреймворки за 2025 рік через їх обсяг у ринку (рис.1.1.)[38].

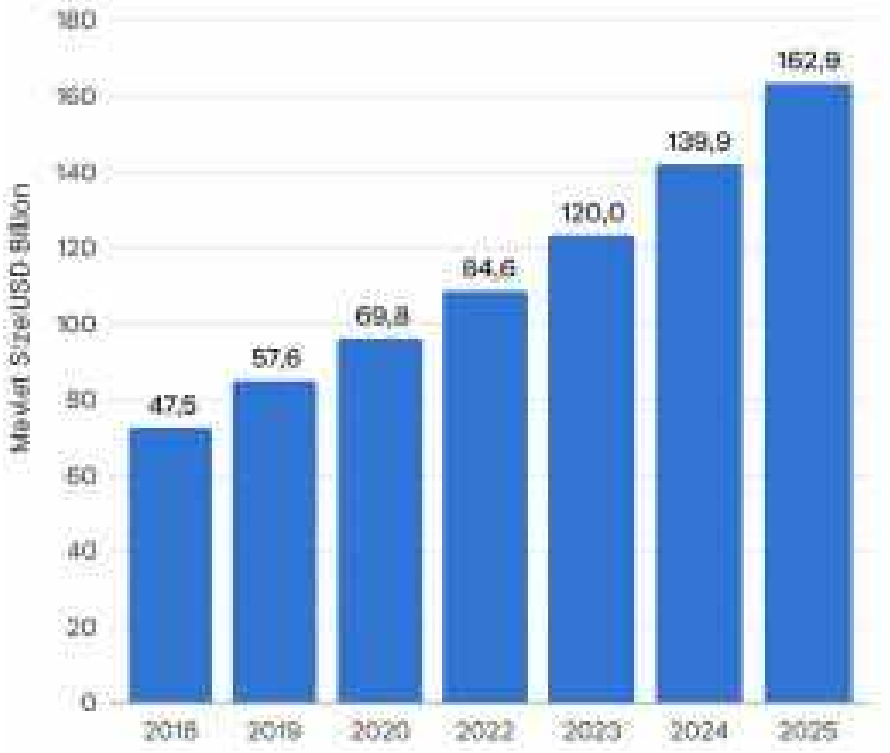


Рис. 1.1. Обсяг інвестицій на розробку крос-платформених додатків за 2018 – 2025 роки.

Занурення у світ сучасної розробки кросплатформених програм вимагає володіння потужним арсеналом. Для створення кросплатформених програм розробники використовують різні інструментальні засоби. Інтегроване середовище розробки (IDE) – це програма, яка дозволяє писати код, перевіряти його на помилки та запускати. Популярні IDE включають Visual Studio, Android Studio та Xcode[16]. Системи контролю версій, наприклад Git, зберігають історію змін у коді та допомагають кільком розробникам працювати над одним проектом без конфліктів[39]. Хмарні сервіси збереження даних, такі як Firebase, дозволяють зберігати інформацію користувачів онлайн, щоб вони могли продовжити використання програми з різних пристроїв[9].

Сучасний ринок пропонує розробникам декілька платформ для створення кросплатформених додатків. Серед найвідоміших в світі є: Unity, Unreal Engine, Godot та інші. Unity підтримує понад 25 операційних систем і найчастіше використовується для мобільних програм[21]. Unreal Engine відомий високоякісною графікою та використовується для створення візуально вражаючих програм[18]. Godot є безкоштовною альтернативою з відкритим кодом, що робить його доступним для початківців та невеликих команд[19].

Три основні ігрові рушії — Unity, Unreal Engine та Godot — відіграють провідну роль у галузі кросплатформеної розробки. Unity, створений у 2004 році компанією Unity Technologies, завоював лідерські позиції з понад 70% ринку мобільних застосунків[21]. Сьогодні даний сервіс вважається одним із найперспективніших інструментів для створення продуктів у цій галузі. Інтегровані редактори графіки та інтерфейсів полегшують створення привабливих і зрозумілих продуктів, які можуть конкурувати на ринку цифрових технологій[1]. Платформа використовує C# як основну мову програмування, забезпечуючи оптимальний баланс між простотою вивчення та функціональною потужністю, розвинена екосистема Unity з Asset Store істотно прискорює процес розробки, скорочуючи його майже вдвічі згідно з дослідженнями[11, 28]. Інтуїтивно зрозуміле середовище з можливостями візуального проектування робить Unity доступним навіть для початківців[16].

Unreal Engine, розроблений Epic Games у 1998 році, спочатку функціонував як внутрішній інструмент компанії. Платформа базується на мові C++ і пропонує систему візуального програмування Blueprints[15]. Основними перевагами цього рушія є фотореалістична графіка та розширені можливості візуалізації з підтримкою складних ефектів. Проте Unreal Engine має обмежений функціонал для 2D-розробки порівняно з Unity[12] та висуває значні вимоги до апаратного забезпечення, що може ускладнювати розробку для пристроїв середньої потужності[4].

Godot Engine, представлений у 2014 році, виділяється серед конкурентів як проект з відкритим кодом під ліцензією MIT, що підтримується спільнотою під керівництвом Software Freedom Conservancy[19]. Платформа пропонує власну мову GDScript з синтаксисом, подібним до Python, а також підтримку C#, C++ та візуального програмування[33]. Компактність рушія (менше 70 МБ) забезпечує ефективну роботу навіть на пристроях з обмеженими ресурсами. Godot має потужні інструменти для 2D-розробки, які багато розробників вважають більш зручними, ніж аналогічні рішення в Unity[19]. Однак менш розвинена екосистема та спільнота обмежують доступність готових ресурсів, плагінів та навчальних матеріалів.

Для систематизації та порівняльного аналізу розглянутих технологічних рішень було розроблено порівняльну таблицю ключових характеристик (таблиця 1.1).

Таблиця 1.1.

Аналіз технологічних рішень для кросплатформеної розробки

Характеристика	**Unity**	**Unreal Engine**	**Godot Engine**
Підтримка 2D-розробки	Висока	Середня	Висока
Підтримка мобільних платформ	Висока	Середня	Середня

Продовження табл. 1.1.

Характеристика	**Unity**	**Unreal Engine**	**Godot Engine**
Інструменти для багатокористувацької розробки	Розвинені	Розвинені	Базові
Крива навчання	Середня	Висока	Низька
Мова програмування	C#	C++	GDScript/C#
Підтримка хмарних сервісів	Розвинена	Базова	Обмежена
Розмір спільноти	Висока	Висока	Середня
Оптимізація для мобільних пристроїв	Висока	Середня	Висока
Ліцензійні умови	Комерційна/Безкоштовна	Роялті/Безкоштовна	Безкоштовна

Джерела:[4, 12, 43].

Важливою складовою інструментарію кросплатформеної розробки є системи збереження та синхронізації даних. Розробники мають вибір між кількома підходами до управління інформацією користувачів.

Локальне збереження з хмарною синхронізацією забезпечує доступ до даних навіть при відсутності інтернет-з'єднання, одночасно підтримуючи оновлення між різними пристроями[3]. Розробники можуть використовувати формати JSON, XML або бінарні файли, залежно від вимог проєкту.

Спеціалізовані хмарні інструменти, як Unity Cloud Save та Firebase, надають готові рішення для управління даними. Ці сервіси забезпечують надійність, масштабованість та безпеку як і розробникам, так і користувачам [34]. Їхнє використання спрощує створення програм, оскільки вони містять вбудовані системи автентифікації та синхронізації в даних середовищах.

Для проєктів з нестандартними вимогами розробляють власні серверні рішення, хоча це потребує більших ресурсів на створення та підтримку.

Оптимізація продуктивності також є ключовим інструментом розробки. Для 2D програм важливо оптимізувати графічні ресурси через атласи спрайтів,

рівні деталізації, форматування текстур та об'єктні пули[12]. У багатокористувацьких проєктах критичним є налаштування мережевої взаємодії шляхом мінімізації трафіку, вибору ефективних протоколів та застосування методів прогнозування[9].

Інструменти для створення інтерфейсу, такі як Unity UI, дозволяють розробляти адаптивні інтерфейси для різних пристроїв та розмірів екранів[23].

Аналіз технологій показує, що кожен інструмент має свою нішу: Unity підходить для більшості 2D проєктів з широкою підтримкою платформ, Unreal Engine - для проєктів з високими вимогами до графіки, а Godot - для невеликих проєктів з обмеженими ресурсами.

1.2. Особливості розробки 2D застосунків у середовищі Unity

Unity є одним із провідних середовищ для розробки двовимірних застосунків завдяки спрощеному спектру спеціалізованих інструментів та технологій. У цьому підрозділі будуть ключові особливості розробки 2D застосунків у середовищі Unity, порівнюючи з іншими компонентами та інструментами, а також методи оптимізації двовимірних проєктів.

Розробка двовимірних застосунків у Unity характеризується специфічними підходами до побудови графічного контенту та розрахунку фізичних взаємодій, попри те, що базується на одній фундаментальній архітектурі рушія Unity. Основною відмінністю є використання 2D спрайтів, які представляють собою плоскі зображення, замість тривимірних моделей[22] Крім того, у двовимірних застосунках застосовується окремий набір фізичних компонентів, спеціально розроблених для роботи з об'єктами, що існують у площині. Система фізики Unity для 2D, включаючи 2D колайдери та інші відповідні компоненти, оптимізована для ефективної обробки зіткнень та інших фізичних взаємодій без необхідності врахування третього виміру. Використання цих специфічних інструментів сприяє оптимізації продуктивності та спрощує логіку обчислень у порівнянні з 3D розробкою[22].

У версіях Unity, починаючи з 2022, відбулися значні покращення інструментів двовимірної розробки. Згідно з офіційною документацією, "для 2D проєктів зміни зосереджені на прискоренні базового програмного забезпечення, імпорту, анімації та фізики. Переглянуто систему атласів спрайтів. Додано підтримку файлів формату PSD та управління шарами до 2D PSD Імпортера, а також впроваджено тріангуляцію Делоне для 2D фізики"[40], що підвищили ефективність розробки двовимірних застосунків та їх продуктивність на різних платформах. Він пропонує широкий набір спеціалізованих інструментів для розробки двовимірних застосунків. Розглянемо ключові компоненти та їх функціональність:

Sprite Editor є центральним інструментом для роботи з двовимірною графікою в Unity. Він дозволяє розділяти спрайт-аркуші на окремі спрайти, налаштовувати їх параметри та створювати спрайт-анімації[5]. Це особливо важливо для оптимізації використання пам'яті та підвищення продуктивності рендерингу.

Tilemap – система для конструювання двовимірних рівнів за допомогою тайлів. Вона включає Tilemap Renderer для відображення тайлів, Tilemap Collider 2D для додавання колізій та Rule Tiles для автоматичного вибору правильного тайлу на основі його оточення[22]. Ця система дозволяє швидко розробляти та редагувати складні двовимірні рівні без необхідності ручного розміщення кожного елемента.

2D Physics є фізичним рушієм, оптимізованим для двовимірних застосунків. Він включає Rigidbody 2D для додавання фізичних властивостей до об'єктів, різні типи колайдерів (Box Collider 2D, Circle Collider 2D, Polygon Collider 2D тощо) для визначення фізичної форми об'єктів та систему 2D Joints для з'єднання об'єктів[12]. Вона забезпечує реалістичну фізичну взаємодію між об'єктами у двовимірному просторі.

2D Animation надає інструменти для створення та відтворення двовимірних анімацій. Вона включає Animation Window для створення та редагування анімацій, Animator Controller для управління та переходу між

анімаціями та 2D IK (Inverse Kinematics) для анімування персонажів з використанням інверсної кінематики[22]. Ці інструменти дозволяють створювати складні та плавні анімації для двовимірних персонажів та об'єктів.

Sprite Atlas є інструментом для об'єднання спрайтів у атласи, що дозволяє оптимізувати використання пам'яті та підвищити продуктивність рендерингу[22]. Використання спрайт-атласів є критично важливим для мобільних застосунків, де ефективне використання ресурсів є ключовим фактором продуктивності.

2D Lighting є системою освітлення для двовимірних застосунків, яка дозволяє створювати динамічне освітлення та тіні[22]. Вона включає 2D Light для створення джерел світла, 2D Shadow Caster для створення тіней та підтримку нормальних мап для двовимірних спрайтів, що дозволяє досягти ефекту об'ємності на пласких об'єктах. Для структурування двовимірних застосунків Unity використовує кілька ключових концепцій:

GameObject є базовим об'єктом у сцені, який може містити різні компоненти, такі як Transform, SpriteRenderer, Collider2D, Rigidbody2D та інші[41]. Ця модульна система дозволяє гнучко конфігурувати об'єкти відповідно до їх функціональності.

Component є модулями, які додаються до GameObject для надання йому певних функцій. Наприклад, SpriteRenderer відповідає за відображення спрайту, Collider2D — за колізії, а Rigidbody2D — за фізичну поведінку об'єкта[41]. Компонентна архітектура дозволяє ефективно повторно використовувати функціональність та створювати складні об'єкти з простих компонентів.

Prefab є шаблоном GameObject, який можна використовувати для створення багатьох однакових об'єктів. Prefabs особливо корисні для створення ворогів, предметів та інших повторюваних елементів застосунку[8]. Система префабів дозволяє централізовано управляти властивостями об'єктів, що значно спрощує процес розробки та підтримки.

Scene є контейнером для ігрових об'єктів та середовища. Застосунок може містити кілька сцен, які представляють різні рівні, меню або інші частини

програми[41]. Система сцен дозволяє структурувати застосунок та ефективно управляти його ресурсами.

Оптимізація є ключовим аспектом розробки двовимірних застосунків, особливо для кросплатформених проєктів, які повинні ефективно працювати на різних пристроях. Основні методи оптимізації включають:

Батчинг (Batching) — об'єднання кількох об'єктів в один для зменшення кількості викликів рендерингу. Unity підтримує статичний та динамічний батчинг, але для двовимірних застосунків особливо важливим є динамічний батчинг, який дозволяє об'єднувати спрайти, що використовують однакові матеріали[22].

Оптимізація графіки, фізики та продуктивності є ключовими аспектами ефективної розробки кросплатформених застосунків. Стиснення текстур, використання спрайт-атласів та правильний вибір формату зменшують розмір застосунку та споживання пам’яті, що особливо важливо для мобільних платформ. Удосконалення фізичного рушія через Composite Collider 2D та використання простих колайдерів знижує навантаження на ЦП, покращуючи загальну продуктивність[22]. Для ефективного управління ресурсами застосовують об’єктні пулі, асинхронне завантаження та оптимізацію скриптів, що дозволяє мінімізувати витрати системних ресурсів[16].

Для мобільних застосунків особливо важливо використовувати легкі шейдери, налаштовувати якість графіки відповідно до можливостей пристрою та ефективно управляти пам'яттю[27]. Ці методи дозволяють забезпечити оптимальну продуктивність навіть на пристроях з обмеженими ресурсами.

Для розробки кросплатформених двовимірних багатокористувацьких застосунків Unity надає ряд спеціалізованих рішень:

Unity Services, включаючи Unity Multiplayer Service, надають інструменти для створення багатокористувацьких застосунків з можливістю масштабування відповідно до кількості користувачів[36]. Ці сервіси забезпечують надійну інфраструктуру для мережевої взаємодії без необхідності розробки власних серверних рішень.

Cloud Save є сервісом для синхронізації прогресу користувачів між різними пристроями, що є критично важливим для кросплатформених застосунків[11, 43]. Цей сервіс забезпечує надійне збереження даних у хмарі та їх доступність на всіх підтримуваних платформах.

Інтеграція з сторонніми мережевими рішеннями, такими як Photon, Mirror або Fishnet, розширює можливості Unity для створення складних багатокористувацьких застосунків[29]. Важливо забезпечити інтуїтивний інтерфейс, плавну анімацію та відгучливий контроль, що підвищує залученість користувачів.

Розробка двовимірних застосунків у Unity вимагає розуміння як технічних аспектів платформи, так і принципів проєктування користувацького досвіду. Використання спеціалізованих інструментів та методів оптимізації дозволяє створювати високоякісні кросплатформені застосунки, які забезпечують оптимальну продуктивність на різних пристроях.

1.3. Обґрунтування вибору технологій для реалізації кросплатформеного застосунку Warbound

На основі проведеного аналізу технологій розробки кросплатформених програмних рішень та особливостей створення двовимірних застосунків необхідно обґрунтувати вибір технологічного стеку для реалізації майбутнього програмного кросплатформеного продукту. Було названо Warbound, поєднавши «War» (переклад з англійського «війна») та «bound» (пов’язаний). Він представляє собою двовимірний багатокористувацький рольовий застосунок у фентезі стилі з покроковою системою взаємодії, де гравець поринається у вигаданий світ у тропічному містечку. Даний проєкт натхненний такими програмними продуктами як Slay the Spire, Albion Online, Mobile Legends та інші кооперативні застосунки, але в більш спрощений формат, в якого буде не суттєво залежати швидкодія пристрою від її рік виробництва. Програма повинна функціонувати на різних платформах, включаючи персональні комп'ютери

(Windows), мобільні пристрої (Android, iOS) та забезпечувати коректну синхронізацію даних користувачів між ними з максимально-допустимими оптимізацією процесу. Проект представляє собою тактичну покрокову гру з елементами RPG, де гравці керують персонажами та їхніми роботами у битвах на полі з клітинками. Гра складається з двох основних частин: створення і кастомізація персонажів та тактичні битви між командами гравців і/або ШІ-опонентів.

За даними фірми досліджень ринку Cognitive Market Research, у 2025 році глобальний ринок тактичних RPG оцінюється у 4,43 мільярда доларів, а до 2033 року прогнозується його зростання до 10,19 мільярда доларів із середньорічним темпом приросту 10,98% (таблиця 1.2.)[17]. Це свідчить про високий попит на тактичні стратегічні ігри, особливо серед гравців віком 18-35 років, які становлять основну аудиторію жанру.

Порівняно з конкурентами, такими як XCOM, Fire Emblem, Into the Breach та Advance Wars, проект пропонує глибоку кастомізацію персонажів, що є важливим фактором залучення гравців. Крім того, мультиплеєрний аспект та інтеграція бойових роботів додають нові стратегічні можливості, яких бракує у класичних представниках жанру.

Ринок тактичних RPG активно розвивається завдяки зростанню популярності мобільних платформ та кросплатформеності, що дозволяє залучати ширшу аудиторію. За даними DataIntelo, ринок RPG загалом оцінювався у 23,2 мільярда доларів у 2023 році і прогнозується його зростання до 53,8 мільярда доларів до 2032 року (таблиця 1.2.)[19]. Це підтверджує тенденцію до розширення жанру та підвищення інтересу до стратегічних ігор із глибокими механіками розвитку персонажів.

Таблиця 1.2.

Динаміка продажів на світовому ринку тактичних рольових ігор за регіонами (2021-2033)

Континент	2021 ($ млн)	2025 ($ млн)	2033 ($ млн)	Середньорічний темп приросту (%)
Світовий ринок	2920.41	4430.2	10194.9	10.98
Північна Америка	820.636	1220.52	2719.99	10.54
Європа	791.432	1209	2837.23	11.25
Азіатсько-Тихоокеанський регіон	922.851	1468.61	3633.45	11.99
Південна Америка	154.782	209.105	382.307	7.83
Близький Схід	119.737	167.018	326.235	8.73
Африка	110.976	155.943	295.651	8.32

Джерела:[48, 49].

2D RPG з мультиплеєром займають стратегічно важливу нішу на глобальному ігровому ринку, особливо в сегменті тактичних рольових ігор. За прогнозами Cognitive Market Research, глобальний ринок тактичних RPG, оцінений у 4,43 мільярда доларів у 2025 році, досягне 10,19 мільярда доларів до 2033 року із середньорічним темпом приросту 10,98%[26]. Особливо помітне зростання спостерігається в Азіатсько-Тихоокеанському регіоні з прогнозованим темпом приросту 11,99% (рис 1.2.). Рольові ігри становлять 23% світових ігрових уподобань, демонструючи стійкість у конкуренції з шутерами та іншими масовими жанрами, особливо на консольних платформах (рис.1.3.).

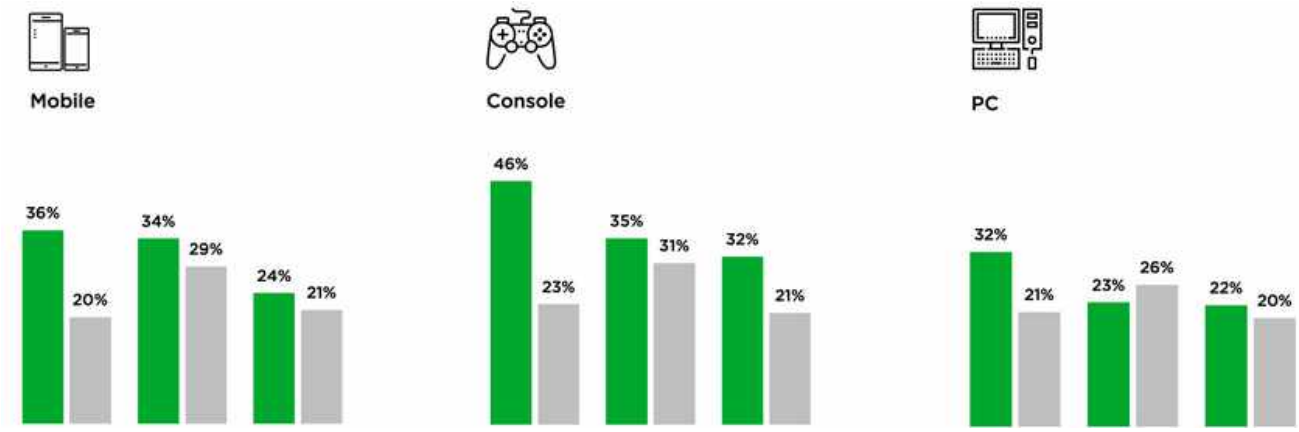


Рис 1.2. Статистика популярності жанру RPG у світовому ринку на популярних пристроях, де сірий колір – це відсоток гравців світу, а зелений – в Японії[26].

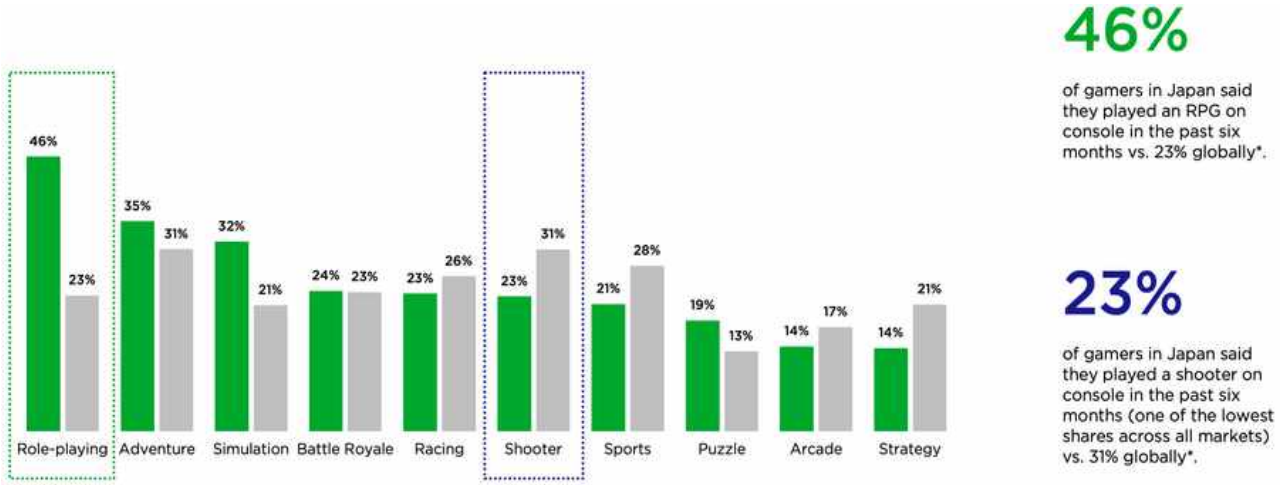


Рис. 1.3. Статистика популярності усіх жанрів у світовому ринку у консольних пристроях, де сірий колір – це відсоток гравців світу, а зелений – в Японії[26].

Unity пропонує потужні інструменти для 2D графіки, включаючи Sprite Editor, Tilemap та системи анімації[5]. Суттєвою перевагою є інтегрована підтримка багатокористувацьких функцій через Unity Multiplayer Service та можливість інтеграції з іншими мережевими рішеннями[14]. Хмарні сервіси Unity Cloud Save забезпечують надійне збереження прогресу користувачів та синхронізацію між пристроями[17, 26].

Використання C# як основної мови програмування надає доступ до сучасних парадигм програмування, включаючи об'єктно-орієнтоване

проектування та асинхронне програмування[27]. Важливим фактором була також наявність розвиненої спільноти розробників та документації.

Для багатокористувацької взаємодії використовуються внутрішні інструменти Unity Services, що забезпечує оптимальну сумісність з іншими компонентами. Unity Services функціонують ідентично на всіх підтримуваних платформах, забезпечуючи консистентний користувацький досвід. Важливою перевагою є масштабованість, яка дозволяє адаптувати серверну інфраструктуру відповідно до зростання кількості користувачів[9, 36, 43].

Для збереження даних використовується комбінований підхід з локальним кешуванням та синхронізацією через Unity Cloud Save, що забезпечує надійність даних та функціонування при нестабільному з'єднанні[31].

Оптимізація продуктивності включає:

- Систему рівнів деталізації (LOD) для графічних елементів[24];
- Асинхронне завантаження ресурсів;
- Використання спрайт-атласів;
- Реалізацію об'єктних пулів.

Таким чином, обраний технологічний стек (Unity, C#, Unity Services) та обраний жанр розробки забезпечує оптимальне вирішення ключових завдань проєкту: кросплатформеність, багатокористувацьку взаємодію, збереження даних, високу продуктивність та її успіх на різних пристроях, який зможе задовольнити потреби різних категорій користувачів й буде створений зручний та простий в експлуатації кросплатформений застосунок.

РОЗДІЛ 2. ДОСЛІДЖЕННЯ І ПРОЕКТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ У СЕРЕДОВИЩІ UNITY

2.1. Створення моделей й архітектури ігрового застосунку Warbound

Перед початком розробки будь-якого ігрового застосунку критично важливим є вибір правильних інструментів для створення графічних елементів – "ассетів" (від англ. assets - ресурси, активи). У сучасній індустрії розробки ігор, де візуальна складова відіграє ключову роль у залученні користувачів, якість графічних ресурсів безпосередньо впливає на успіх проєкту[1]. Для застосунку Warbound було обрано комплексний підхід до створення візуальних елементів, що поєднує професійні інструменти з оптимізованим робочим процесом.

Adobe Photoshop став основним інструментом для створення 2D-ассетів завдяки своїй універсальності та глибокій інтеграції з Unity. Згідно з офіційною документацією Unity, інтеграція з Photoshop через пакет PSD Importer дозволяє безпосередньо імпортувати файли формату PSD, зберігаючи всю структуру шарів та позиціонування елементів[42]. Ця функціональність є особливо цінною при розробці складних анімаційних секвенцій та багатошарових UI-елементів, оскільки дозволяє митцям працювати в звичному середовищі без необхідності додаткової конвертації файлів.

Функціональність Photoshop виходить далеко за межі базового редагування. Програма дозволяє дизайнеру працювати з деталізованими текстурами та нормал-картами, завдяки яким об'єкти в Unity виглядають більш реалістично, навіть коли використовуються в 2D-іграх (рис. 2.1.). Потужні інструменти для корекції зображень, роботи з масками та різноманітні режими накладання допомагають створювати візуальні елементи, які бездоганно інтегруються у середовище Unity. Найважливішою перевагою такої інтеграції є можливість оновлювати графіку без необхідності відтворювати весь процес композиції заново – достатньо внести правки в PSD-файл та повторно імпортувати його, що значно економить час у процесі розробки.

За останні роки сучасні можливості Adobe Photoshop значно розширились, перетворюючи його на потужний інструмент для розробки професійної 2D-графіки, ігрових ассетів. Цей прогрес став можливим завдяки постійному вдосконаленню функціоналу розробниками Adobe, а також активній підтримці з боку креативної спільноти. Особливо значний крок вперед відбувся з 2020 року, коли інтеграція Photoshop з хмарними сервісами Creative Cloud та ігровими рушіями суттєво прискорила робочий процес для розробників ігор[42].

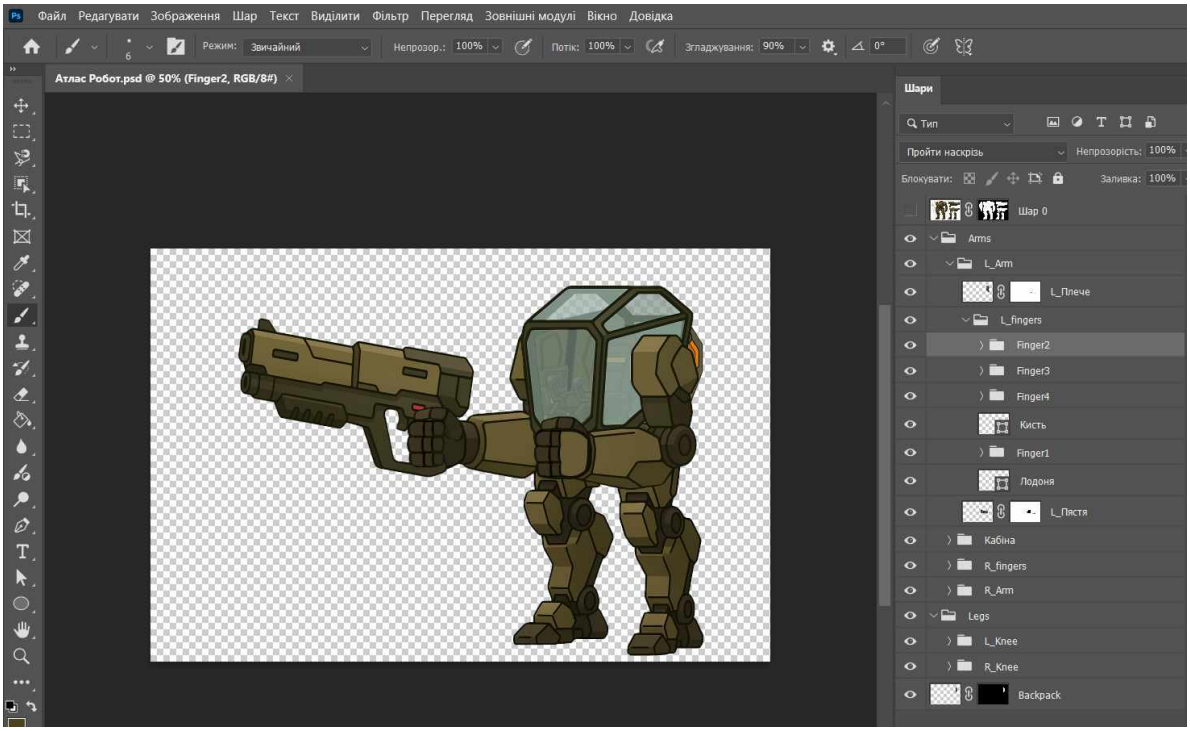


Рисунок 2.1 – Процес створення текстури в Adobe Photoshop 2025.

У Warbound усі графічні елементи, включаючи спрайти персонажів, інтерфейс користувача, фонові текстури та візуальні ефекти, були розроблені саме в Adobe Photoshop, оскільки забезпечив повний контроль над художнім стилем гри та дозволив уникнути обмежень, пов'язаних із використанням готових ассетів (рис 2.1.), де представлені ключові графічні елементи, створені в процесі розробки.

Ключові переваги Photoshop для розробки ігор:

- Розширені інструменти малювання та ретуші для унікального візуального стилю
- Підтримка роботи з векторною та растровою графікою
- Можливість створення складних багатошарових композицій
- Широкий вибір форматів експорту з підтримкою прозорості
- Інтеграція з іншими інструментами Adobe

Після підготовки графічних матеріалів для розробки, необхідно визначити архітектурну концепцію програмного застосунку. Архітектура є фундаментальним аспектом, що формує структуру, організацію компонентів та механізми взаємодії між елементами системи.

Для наочного відображення структури компонентів використовуються UML-діаграми класів (рис. 2.2), які демонструють взаємозв'язки між компонентами системи.

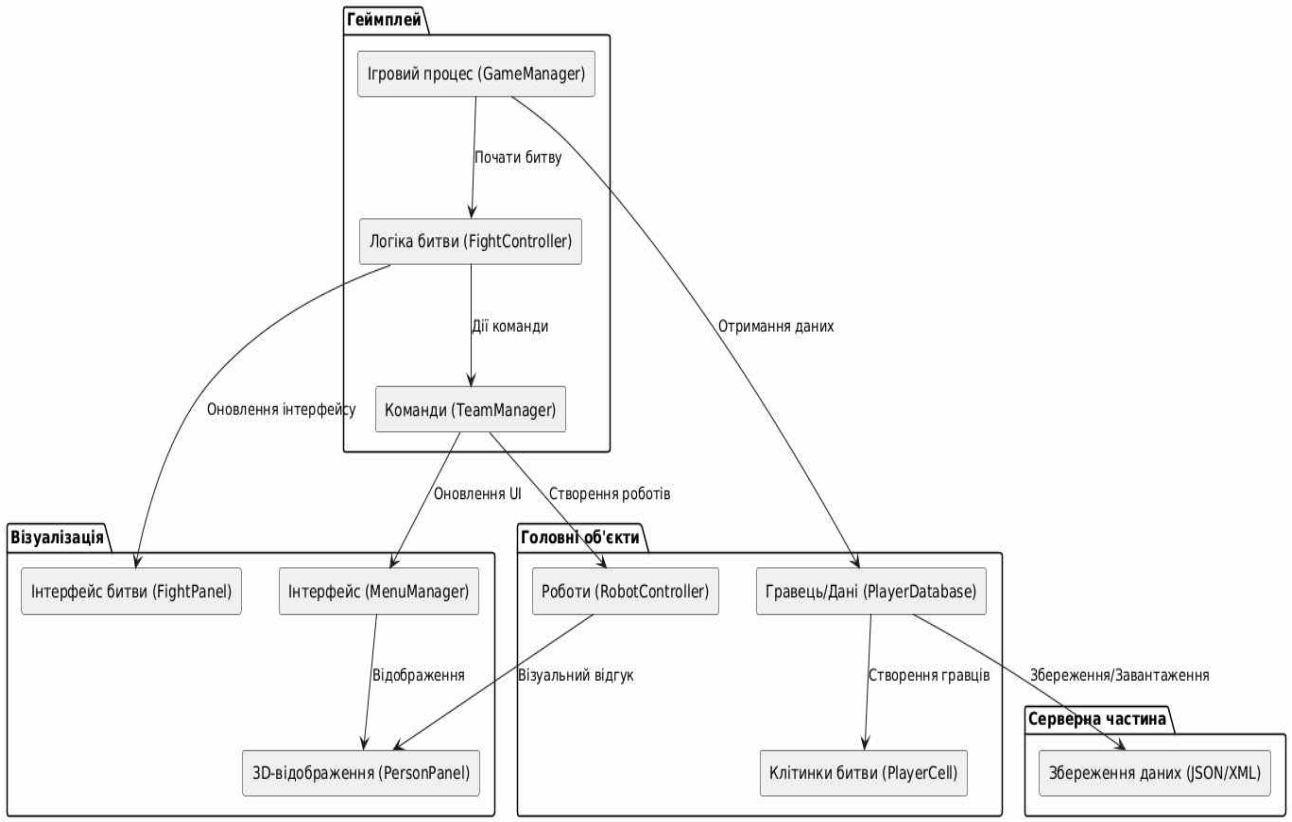


Рисунок 2.2. UML-діаграма архітектури застосунку Warbound

На наведеній вище схемі представлена архітектура розробленого ігрового застосунку, що складається з чотирьох основних блоків, які взаємодіють між собою через визначені інтерфейси:

Блок "Геймплей" є центральним компонентом архітектури, що керує основними ігровими процесами. Він має в собі три головні й ключові об’єкти програми:

1. GameManager – головний контролер, що координує всі системи гри, ініціює бої та керує переходами між режимами.
2. FightController – реалізує логіку бойової системи, включаючи ходи гравців, атаки та визначення результатів.
3. TeamManager – відповідає за формування команд, розподіл гравців та координацію їхніх дій.

Даний блок діє як мозковий центр гри, надсилаючи команди до інших блоків та обробляючи вхідні дані. Наприклад, коли GameManager ініціює бій, він передає команду "Почати битву" до FightController, який потім взаємодіє з TeamManager для координації дій команд.

Блок "Візуалізація" забезпечує все візуальне представлення гри та перетворює їх на візуальні об’єкти й отримує дані від блоку "Геймплей". Наприклад, коли відбувається оновлення стану команд у TeamManager, ця інформація передається до MenuManager для оновлення UI та до PersonPanel для відображення актуального стану персонажів.

- FightPanel – відображає поле бою, сітку клітинок та керування боєм користувача.
- MenuManager – керує загальною навігацією між різними панелями інтерфейсу на сцені.
- PersonPanel – забезпечує 2D-візуалізацію персонажів та їх роботів.

Блок "Головні об'єкти" забезпечує функціональність конкретних ігрових об'єктів. TeamManager взаємодіє з RobotController для створення роботів, а

PlayerDatabase надає дані про гравців до GameManager та створює PlayerCell об'єкти для розміщення на полі бою. Має в собі такі класи:

- RobotController – керує поведінкою персонажа, їх атаками, захистом та візуальними ефектами.
- PlayerDatabase – зберігає та керує всіма даними про гравців, їх характеристики та прогрес.
- PlayerCell – візуалізує клітинки на полі бою, які розміщуються в персонажі.

Блок "Серверна частина" взаємодіє виключно з PlayerDatabase, забезпечуючи збереження та завантаження прогресу гри. Це дозволяє гравцям продовжувати гру з того місця, де вони зупинилися, маючи свою систему збереження даних (JSON/XML), який записує їх у файли та завантажує при потребі.

Представлена архітектурна модель відповідає багаторівневому шаблону проєктування (multilayer architecture pattern), який забезпечує чітке розділення відповідальності між компонентами системи та їх логічну організацію[13]. Розглянемо детальніше кожен із архітектурних шарів застосунку.

Архітектурна модель відповідає багаторівневому шаблону проєктування, забезпечуючи розділення відповідальності між компонентами:

Presentation Layer (Шар представлення) реалізований через компоненти Unity UI з використанням Canvas Scaler для адаптації до різних пристроїв.

Business Layer (Бізнес-шар) містить основну логіку гри, використовуючи патерни State Machine для управління станами бою, Observer для комунікації між компонентами та Repository для абстрагування доступу до даних.

Data Layer (Шар даних) відповідає за зберігання та управління даними через комбінацію локального кешування та хмарного збереження.

Network Layer (Мережевий шар) забезпечує багатокористувацьку функціональність з ефективною синхронізацією стану гри.

2.2. Моделі даних та система збереження інформації

Ефективна організація даних є критичним фактором успіху будь-якого багатокористувацького застосунку. Для Warbound було розроблено комплексну систему моделей даних, яка забезпечує гнучкість, продуктивність та легкість розширення. Згідно з сучасними дослідженнями, правильно спроєктовані моделі даних можуть підвищити продуктивність застосунку на 40-60% порівняно з неоптимізованими рішеннями[3]. Моделі даних у Warbound спроєктовані з використанням принципів SOLID для забезпечення гнучкості та розширюваності[13].

SOLID – це набір принципів об'єктно-орієнтованого програмування, які допомагають створювати гнучкий, підтримуваний і масштабований код[13].

Абревіатура розшифровується так:

S – Single Responsibility Principle (Принцип єдиної відповідальності): Клас має виконувати лише одну задачу і не містити зайвої логіки.

O – Open/Closed Principle (Принцип відкритості/закритості): Код має бути відкритим для розширення, але закритим для модифікації.

L – Liskov Substitution Principle (Принцип підстановки Лісков): Підкласи повинні коректно замінювати батьківські класи без змін у логіці.

I – Interface Segregation Principle (Принцип розділення інтерфейсів): Краще мати кілька вузьких інтерфейсів, ніж один великий, який вимагає непотрібних методів.

D – Dependency Inversion Principle (Принцип інверсії залежностей): Класи повинні залежати від абстракцій, а не від конкретних реалізацій.

Один із прикладів варто зазначти центральну модель Player:

[System.Serializable] // атрибут, який дозволяє об'єкту бути використаний у IDE- редакторі

public class Player // даний клас

{ // позначення обмеженя блока коду.

public string nickname; // Поля даних цього класу

```
public int level;
public int experience;
public int health;
public int attack;
public int defense;
public Dictionary<string, int> customization;
public List<Item> inventory; // Список. В даному випадку інвентаря
public List<Achievement> achievements;
public Player(string name) // Конструктор та методи класу
{
    nickname = name;
    level = 1;
    experience = 0;
    health = 100; // Ініціалізація полів
} }
```

Атрибут «[System.Serializable]» дозволяє Unity автоматично серіалізувати об'єкти цього класу для збереження. Dictionary використовується для зберігання налаштувань кастомізації, де ключ - назва частини тіла, а значення - індекс обраного варіанту[30].

Ядро системи та управління гравцями організоване через клас GameManager, який координує стан гри (GameState), управляє поточним гравцем (currentPlayer) та його даними (currentCharacterData), а також взаємодіє з FightController для проведення боїв (рис. 2.3.). Дані гравців зберігаються та завантажуються через PlayerDatabase, де реалізовано методи створення акаунтів (CreateAccount), пошуку (GetPlayer) та збереження (SaveDatabase). Для створення та редагування персонажів використовується PlayerController, який відповідає за введення нікнейму, генерацію аватарів (CreateCharacter) та збереження їх у базу даних (SaveCharacterToDatabase). Передача даних між сценами забезпечується класом DataScene, що зберігає обраного гравця (selectedPlayer) та його характеристики (characterData).

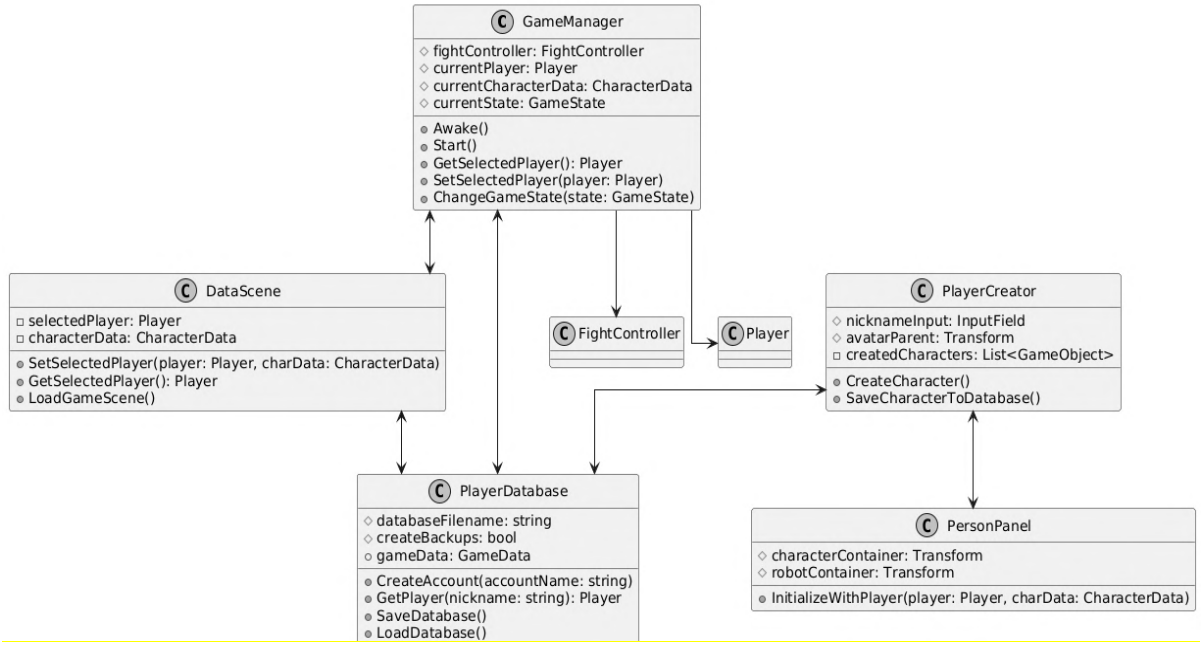


Рис 2.3. Ядро системи та управління гравцями.

Центральною моделлю даних є клас Player, який представляє персонажа гравця та інкапсулює всі його характеристики, здібності та прогрес (рис. 2.4), клас Player має взаємозв'язки з GameManager, DataScene та PlayerDatabase через різні компоненти системи (рис. 2.3.).



Рис. 2.4. UML-структура класу Player.

Ця модель включає базову інформацію (ім'я, рівень, досвід), бойові характеристики (здоров'я, сила, захист, швидкість), налаштування кастомізації

(словник з індексами обраних елементів), інвентар предметів та обладнання, прогрес та досягнення, а також історію боїв та статистику. Кожен компонент моделі спроєктований з урахуванням можливості серіалізації для збереження та передачі через мережу.

Система збереження даних реалізує багаторівневий підхід, що поєднує локальне кешування з хмарною синхронізацією (рис. 2.5.). Такий підхід забезпечує швидкий доступ до даних навіть при відсутності мережевого з'єднання, автоматичну синхронізацію між пристроями користувача та захист від втрати даних через резервне копіювання[15].

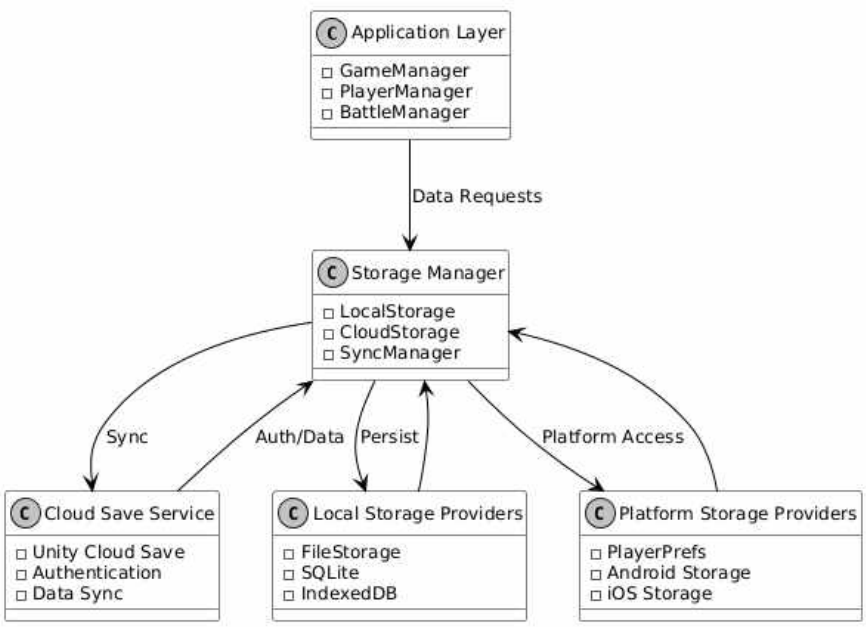


Рис. 2.5. Архітектура системи збереження даних

Storage Manager – центральний компонент системи, який координує взаємодію між застосунком, локальним сховищем та хмарним сервісом. Він реалізує патерн Facade, надаючи простий та уніфікований інтерфейс для збереження та завантаження даних.

Local Storage Provider – компонент, відповідальний за локальне збереження даних. Він абстрагує конкретні механізми збереження даних, такі як файлова

система, SQLite або IndexedDB, надаючи єдиний інтерфейс для роботи з локальним сховищем.

Platform Storage Provider - компонент, який забезпечує доступ до специфічних для платформи механізмів збереження даних, таких як PlayerPrefs в Unity, AndroidStorage на Android або iOSStorage на iOS.

Cloud Save Service - компонент, який забезпечує взаємодію з хмарним сервісом Unity Cloud Save для синхронізації даних між різними пристроями:

```
public class CloudSaveManager : MonoBehaviour // взаємодія зі сценою Unity
{ private async Task SaveToCloud(string key, object data) { try {
    var jsonData = JsonUtility.ToJson(data);
    await CloudSaveService.Instance.Data.SaveAsync(
      new Dictionary<string, object> { { key, jsonData } } );  }
  catch (Exception e)
  { Debug.LogError($"Failed to save to cloud: {e.Message}"); } } }
```

Система автоматично вирішує конфлікти при синхронізації, використовуючи timestamp (мітку часу) останньої модифікації. При конфлікті зберігається версія з новішою міткою часу.

Sync Manager - компонент, відповідальний за синхронізацію даних між локальним сховищем та хмарним сервісом. Він реалізує стратегії вирішення конфліктів та оптимізації мережевого трафіку.

Система збереження даних реалізує наступні ключові функції:

Локальне збереження використовує бінарну серіалізацію для оптимізації розміру файлів та швидкості доступу. Дані зберігаються в спеціальній директорії застосунку з використанням шифрування AES-256 для захисту від несанкціонованого доступу[10]. Система автоматично створює резервні копії при кожному збереженні, зберігаючи останні 5 версій для можливості відновлення[10].

Хмарна синхронізація реалізована через Unity Cloud Save - сервіс, який забезпечує надійне збереження даних у хмарі з автоматичною синхронізацією між пристроями[11]. Інтеграція з цим сервісом дозволяє користувачам

безперешкодно переходити між різними платформами, зберігаючи весь прогрес. Система використовує диференційну синхронізацію, передаючи лише змінені фрагменти даних для оптимізації мережевого трафіку.

Для забезпечення цілісності даних реалізовано систему транзакцій, яка гарантує атомарність операцій збереження. У випадку збою під час збереження, система автоматично відкочується до попереднього стабільного стану. Також впроваджено механізм вирішення конфліктів для ситуацій, коли дані були змінені на різних пристроях одночасно[3].

Міграція даних між версіями застосунку забезпечується через систему версіонування схем. Кожна модель даних містить інформацію про версію, а при завантаженні даних старішої версії автоматично виконується міграція до поточного формату. Це дозволяє безболісно оновлювати застосунок без втрати користувацьких даних[39].

Забезпечення безпеки даних включає в собі:

- Шифрування даних - дані шифруються як під час зберігання, так і під час передачі, що забезпечує їх конфіденційність та цілісність.
- Аутентифікація - доступ до даних надається лише автентифікованим користувачам, що запобігає несанкціонованому доступу.
- Авторизація - кожен користувач має доступ лише до своїх даних, що забезпечується на рівні хмарного сервісу.
- Захист від підробки - дані підписуються цифровим підписом, що дозволяє виявити спроби їх модифікації.

Для інтеграції з Unity Cloud Save використовується офіційний API, який надає методи для збереження та завантаження даних у хмарі. Інтеграція реалізована з використанням патерну Adapter, який адаптує інтерфейс Unity Cloud Save до інтерфейсу IStorageProvider, що використовується в застосунку, яке представлено на рисунку 2.6.

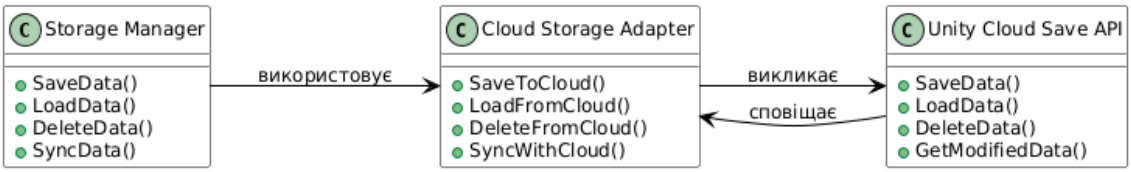


Рис. 2.6. Схема взаємодії з Unity Cloud Save

Storage Manager викликає методи Cloud Storage Adapter, який, у свою чергу, взаємодіє з Unity Cloud Save API для збереження та завантаження даних у хмарі. API (Application Programming Interface) – це набір правил, механізмів і інструментів, які дозволяють програмам взаємодіяти одна з одною. API визначає, як одна програма може запитувати та отримувати дані або викликати функції іншої програми без необхідності розбиратися у внутрішній логіці її роботи. Такий підхід забезпечує гнучкість та можливість заміни хмарного сервісу без модифікації основної логіки застосунку.

2.3. Механіки покрокових боїв

Покрокові бої є серцем геймплею Warbound, поєднуючи стратегічну глибину з доступністю для широкої аудиторії.

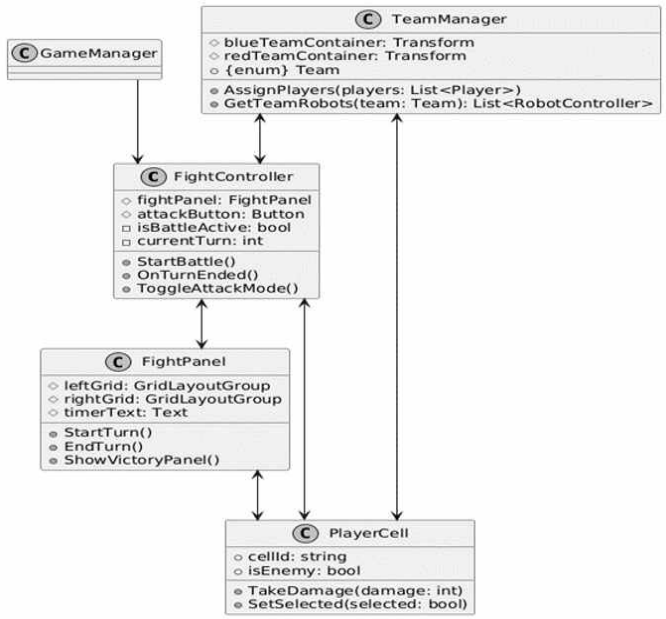


Рис. 2.7. Структура покрокового бою в застосунку Warbound

Дизайн бойової системи натхненний успішними проєктами жанру, але адаптований для багатокористувацької взаємодії та кросплатформеного використання[25]. Її загальна структура знаходиться на рисунку 2.7.

FightController - центральний клас, що керує боєм; BattleAction - клас для представлення дій у бою; PlayerCell - представляє клітинку на полі бою. Усі скрипти знаходиться у додатку А.

Основою бойової системи є покрокова модель з одночасним плануванням дій. На початку кожного раунду всі гравці одночасно вибирають дії для своїх персонажів протягом обмеженого часу (30 секунд). Після закінчення фази планування всі дії виконуються в порядку, визначеному швидкістю персонажів. Така система створює напруження та вимагає передбачення дій супротивника[33].

Поле бою представлене двома сітками розміром 3×5 клітинок для кожної команди. Така конфігурація забезпечує достатній простір для тактичного маневрування, зберігаючи компактність для мобільних екранів. Кожна клітинка може містити одного персонажа або робота, а позиціонування впливає на доступність цілей для різних типів атак[32].

Система дій включає базові атаки (звичайну кнопку атаки та ультимативну кнопку здібності), спеціальні здібності з різними ефектами, переміщення між клітинками, використання предметів та взаємодію з роботами. Кожна дія має свою вартість у очках дії, що обмежує кількість можливих дій за хід і змушує гравців приймати складні тактичні рішення.

Кожен персонаж може мати власного робота, які мають спеціалізовані ролі: атакуючі (високий урон, низький захист), захисні (поглинання шкоди, баф союзників), підтримки (лікування, зняття дебафів) та контролю (блокування кнопок або ходів). Синергія між персонажем та його роботами є ключовим фактором перемоги.

Система ефектів забезпечує різноманітність тактичних можливостей. Ефекти можуть бути миттєвими (пряма шкода, лікування), тривалими (отрута, регенерація), накопичувальними (підсилення з кожним ходом) та умовними

(активація при певних діях). Взаємодія різних ефектів створює емерджентний геймплей з безліччю можливих комбінацій[28].

2.4. Структуризація даних для кросплатформеної реалізації

Ефективність кросплатформеного застосунку значною мірою залежить від правильного вибору алгоритмів та структур даних. Для Warbound було проведено детальний аналіз вимог кожної підсистеми та підібрано оптимальні рішення, що забезпечують баланс між продуктивністю та використанням ресурсів[35].

Для представлення ігрового поля використовується двовимірний масив з додатковими структурами для швидкого пошуку. Основний масив Cell[,] забезпечує O(1) доступ за координатами, що критично важливо для частих операцій перевірки стану клітинок. Додатково підтримується Dictionary<int, Cell> для швидкого пошуку клітинок за ідентифікаторами персонажів, що дозволяє ефективно знаходити позицію конкретного юніта[7].

Система управління діями використовує Priority Queue на основі бінарної купи для черги виконання дій. Пріоритет визначається швидкістю персонажа та типом дії, забезпечуючи правильний порядок виконання. Операції вставки та вилучення виконуються за O(log n), що забезпечує ефективність навіть при великій кількості одночасних дій[12].

Для оптимізації пошуку шляхів на полі бою використовується алгоритм A* з евристикою Манхеттенської відстані. Алгоритм адаптований для гексагональної сітки та враховує перешкоди у вигляді інших персонажів. Для покращення продуктивності використовується кешування розрахованих шляхів та ієрархічний пошук для великих відстаней[23].

Система ефектів реалізована через композитний патерн з використанням об'єктного пулу:

```
public ObjectPool(T prefab, int initialSize, Transform parent = null)   {
    this.prefab = prefab; this.parent = parent;
   for (int i = 0; i < initialSize; i++) {
```

```
// Створення нового об'єкта на основі префабу
var obj = GameObject.Instantiate(prefab, parent);
// Деактивація об'єкта, щоб він не був видимим або активним відразу
obj.gameObject.SetActive(false);
// Додавання об'єкта до пулу (черги для повторного використання)
pool.Enqueue(obj); } }
```

Кожен ефект є незалежним об'єктом з власною логікою, що дозволяє легко додавати нові типи ефектів. Об'єктний пул зменшує навантаження на збирач сміття, що важливо для мобільних платформ з обмеженими ресурсами[16].

Управління пам'яттю реалізовано через багаторівневу систему кешування. Часто використовувані ресурси (спрайти персонажів, звукові ефекти) зберігаються в оперативній пам'яті. Менш критичні ресурси завантажуються асинхронно за потребою. Система автоматично адаптується до доступної пам'яті пристрою, вивантажуючи некритичні ресурси при необхідності[37]. Як приклад такої системи є метод:

```
public TValue Get(TKey key, Func<TValue> factory) // отримання кешу
{ var value = factory();
var size = EstimateSize(value);
    // Якщо додавання нового значення перевищує ліміт пам’яті - звільняємо місце
while (currentMemory + size > maxMemory && cache.Count > 0) {
  RemoveLeastValuable(); // Видаляємо зайвий кешу  }
cache[key] = new CacheEntry // Додаємо нове значення до кешу      {
  Value = value,  // Збережене значення
  Size = size,  // Розмір значення
  LastAccess = DateTime.Now,  // Час додавання до кешу
  AccessCount = 1  // Початкове значення лічильника доступу      };
currentMemory += size; // Оновлюємо поточне використання пам’яті
return value; } }
```

РОЗДІЛ 3. РОЗРОБКА Й ОПТИМІЗАЦІЯ КРОСПЛАТФОРМЕНОГО ІГРОВОГО ЗАСТОСУНКУ

3.1. Реалізація ключових елементів ігрового застосунку

Реалізація ключових елементів застосунку Warbound передбачає імплементацію функціональних компонентів, створення інтерфейсу користувача та забезпечення стабільної взаємодії між ними. Розробка здійснювалася відповідно до методології Agile з використанням інкрементального підходу до додавання функціональності та отримувати зворотній зв’язок[20]. Такий підхід дозволив на ранніх етапах отримати працюючий прототип з мінімальною функціональністю та поступово розширювати його можливості, що відповідає сучасним тенденціям у розробці ігрового програмного забезпечення.

Процес розробки застосунку Warbound структурно можна розділити на наступні основні етапи: створення базової архітектури проекту та налаштування середовища розробки, розробка системи створення та управління персонажами, імплементація механік покрокових боїв та бойової системи, реалізація системи збереження даних та багатокористувацької взаємодії, інтеграція користувацького інтерфейсу та візуальних ефектів, а також оптимізація для різних цільових платформ. Кожен етап передбачав розробку відповідних компонентів та їх поступову інтеграцію в загальну структуру застосунку.

Процес розробки застосунку Warbound структурно можна розділити на наступні основні етапи:

1. Створення базової структури проєкту та налаштування середовища розробки;
2. Розробка системи створення та налаштування персонажів;
3. Імплементація механік покрокових боїв;
4. Реалізація системи збереження даних та багатокористувацької взаємодії;
5. Інтеграція користувацького інтерфейсу та візуальних ефектів;

6. Оптимізація для різних платформ.

Кожен етап передбачав розробку відповідних компонентів та їх інтеграцію в загальну структуру застосунку.

Перший етап розробки передбачав створення базової структури проєкту в середовищі Unity. Для забезпечення модульності та підтримуваності застосунку було використано компонентно-орієнтований підхід з чітким розділенням відповідальності між різними модулями[40].

Структура проєкту організована відповідно до рекомендацій щодо оптимальних практик розробки у середовищі Unity та включає наступні основні каталоги:

```
/Assets
  /Scripts
  /Core        # Ядро застосунку, базові класи та інтерфейси
  /Networking  # Компоненти мережевої взаємодії
  /Characters  # Класи для представлення персонажів
  /Scenes      # Сцени застосунку
    /UI        # Компоненти інтерфейсу користувача
    /Database  # Компоненти системи збереження даних
    /Utils     # Допоміжні утиліти та розширення
    /Prefabs   # Шаблони об'єктів
    /Resources # Ресурси, доступні під час виконання
    /Sprites   # Двовимірні спрайти
    /Animations # Анімації
    /Audio     # Аудіо ресурси
  /Plugins     # Сторонні плагіни та бібліотеки
```

Другим етапом розробки передбачав створення комплексної системи створення й управління персонажами та їх налаштування, яка включає їх створення, кастомізацію та збереження. Встановлення обмеження швидкодії під різні пристрої:

public class QualityManager : MonoBehaviour {

```
void Start() {
    #if UNITY_ANDROID || UNITY_IOS
        QualitySettings.SetQualityLevel(1); // Середня якість для мобільних
        Application.targetFrameRate = 30;
    #elif UNITY_STANDALONE
        QualitySettings.SetQualityLevel(3); // Висока якість для ПК
        Application.targetFrameRate = 60;
    #endif } }
```

Для забезпечення модульності та підтримуваності застосунку було використано компонентно-орієнтований підхід з чітким розділенням відповідальності між різними модулями. Центральним компонентом цієї системи є PlayerCreator, який координує весь процес створення персонажа та взаємодіє з іншими системами застосунку.

Структура системи персонажів організована відповідно до принципів об'єктно-орієнтованого програмування та включає наступні основні компоненти: PlayerCreator (додаток Б) для управління процесом створення, AvatarGenerator для генерації візуального представлення персонажів, CharacterCustomization для налаштування зовнішності та характеристик, та PlayerDatabase для збереження створених персонажів (додаток В). Ця система реалізована в сцені SceneCreator та включає три основні сторінки (Page0, Page1, Page2), які послідовно проводять користувача через процес створення та налаштування персонажа.

Центральним компонентом системи є клас PlayerCreator, який реалізує весь цикл створення персонажа, від введення базової інформації до збереження готового персонажа в базі даних. Компонент забезпечує валідацію введених користувачем даних, координацію роботи підсистем кастомізації та генерації аватарів, а також інтеграцію з системою збереження даних й був представлений у додатку Б. Особливістю реалізації є підтримка як стандартного процесу створення, так і редагування існуючих персонажів з можливістю перегляду змін у реальному часі.

AvatarGenerator відповідає за створення візуального представлення персонажів у ігровому інтерфейсі. Компонент використовує систему префабів Unity для генерації складних ієрархій об'єктів, що включають аватар, фон, рамку, текст імені та кнопки управління. Кожен створений аватар є самостійним GameObject з повним набором компонентів для взаємодії з користувачем, включаючи обробку кліків, анімацію при наведенні курсора та інтеграцію з системою вибору персонажів.

Система кастомізації персонажів реалізована через компонент CharacterCustomization, який забезпечує гнучке налаштування зовнішності персонажів через категоризовану систему частин тіла. Система підтримує п'ять основних категорій кастомізації: голова, тіло, руки, ноги та одяг, кожна з яких може мати різну кількість варіантів відповідно до доступних ресурсів.

Технічна реалізація кастомізації базується на динамічному завантаженні спрайтів з папки Resources та їх застосуванні до відповідних SpriteRenderer компонентів. Система автоматично сканує доступні ресурси при ініціалізації та створює відповідні опції для кожної категорії. Це дозволяє легко додавати нові варіанти зовнішності без зміни коду програми - достатньо розмістити нові спрайти у відповідних папках ресурсів.

Інтерфейс кастомізації реалізований через систему слайдерів, які дозволяють користувачеві інтуїтивно переключатися між доступними варіантами для кожної категорії. SliderController забезпечує синхронізацію між інтерфейсом користувача та системою кастомізації, автоматично застосовуючи зміни до моделі персонажа в реальному часі. Всі налаштування кастомізації зберігаються у вигляді словника Dictionary<string, int>, що забезпечує ефективну серіалізацію та відновлення стану персонажа.

База даних персонажів реалізована через клас PlayerDatabase, який забезпечує повний цикл управління даними: створення, читання, оновлення та видалення персонажів (рис. 3.1.). Система використовує JSON серіалізацію для збереження даних у файловій системі з підтримкою автоматичного створення резервних копій та міграції даних між версіями застосунку.



Рис. 3.1. Приклад збереження файлу зі збереженими даними

Найбільш складним компонентом застосунку є розробка бойової механіки, що забезпечує основний ігровий процес. Центральним елементом є FightController, який координує всі аспекти бою: управління чергою ходів, обробку дій гравців, роботу штучного інтелекту та визначення умов завершення бою. Поле бою представлене системою клітинок PlayerCell, кожна з яких може містити персонажа або робота та забезпечує інтерфейс для взаємодії з користувачем (рис. 3.2а.). Клітинки організовані у дві сітки розміром 3×5 для кожної команди (рис. 3.2б.). Кожна клітинка має унікальний ідентифікатор та

може знаходитися у різних станах: порожня, зайнята, вибрана, атакована або знищена.



Рис. 3.2. Розробка бойової механіки в Unity на сцені SampleScene

а) Інтерфейс користувача б) Шаблон розподілу команди.

3.2. Оптимізація продуктивності для різних платформ

Процес оптимізації включав аналіз вузьких місць продуктивності, розробку адаптивних алгоритмів та реалізацію платформо-специфічних оптимізацій. Ефективне управління пам'яттю реалізовано через комбінацію об'єктних пулів, асинхронного завантаження ресурсів та інтелектуального кешування. Об'єктні пули використовуються для часто створюваних об'єктів, таких як візуальні ефекти, тимчасові UI елементи та ігрові повідомлення. Система автоматично регулює розмір пулів залежно від поточних потреб та доступної пам'яті.

Асинхронне завантаження ресурсів забезпечує плавність роботи інтерфейсу навіть при завантаженні великих ресурсів, таких як текстури персонажів або звукові ефекти. Система пріоритизації забезпечує першочергове завантаження критично важливих ресурсів, тоді як менш важливі елементи завантажуються у фоновому режимі.

Інтелектуальне кешування адаптується до шаблонів використання користувача та характеристик пристрою. Часто використовувані дані зберігаються в оперативній пам'яті для швидкого доступу, тоді як рідко потрібні ресурси можуть бути вивантажені для звільнення місця. Система включає механізми попереднього завантаження для передбачення потреб користувача.

Система адаптивних налаштувань автоматично визначає оптимальні параметри якості залежно від характеристик пристрою та поточної продуктивності. Алгоритм враховує такі фактори, як обсяг доступної пам'яті, потужність процесора, тип графічного адаптера та поточну частоту кадрів.

Для мобільних пристроїв з обмеженими ресурсами система автоматично знижує якість текстур, зменшує кількість візуальних ефектів та оптимізує частоту оновлення інтерфейсу. На потужних настільних системах активуються додаткові візуальні ефекти, підвищується роздільна здатність рендерингу та збільшується цільова частота кадрів.

Динамічна адаптація дозволяє системі реагувати на зміни навантаження під час гри. Якщо продуктивність падає нижче прийнятного рівня, система автоматично знижує якість налаштувань для відновлення плавності роботи. Після стабілізації ситуації якість поступово відновлюється.

Адаптивний інтерфейс користувача забезпечує оптимальний досвід на пристроях з різними розмірами екранів та співвідношеннями сторін. Система автоматично налаштовує розмір та розташування елементів інтерфейсу, забезпечуючи їх доступність та зручність використання на всіх підтримуваних пристроях.

Для сенсорних пристроїв реалізовані додаткові оптимізації: збільшені зони натискання для кнопок, оптимізовані жести для навігації та спеціальні елементи керування для мобільної взаємодії. Настільні версії підтримують розширені можливості управління з клавіатури та миші, включаючи гарячі клавіші та контекстні меню.

Система також включає оптимізації для доступності: підтримка екранних читачів, налаштування контрастності та розміру шрифтів, альтернативні схеми управління для користувачів з обмеженими можливостями.

3.3. Тестування та впровадження застосунку

Комплексне тестування застосунку включало функціональне тестування, тестування продуктивності, тестування сумісності та користувацьке тестування. Кожен етап тестування мав специфічні цілі та використовував відповідні методології для забезпечення високої якості кінцевого продукту.

Функціональне тестування охоплювало всі основні компоненти застосунку. Тестування системи створення персонажів включало перевірку всіх етапів процесу: введення імені, вибір зовнішності, налаштування характеристик та збереження результату. Особливу увагу було приділено валідації введених даних та обробці граничних випадків.

Тестування бойової системи передбачало перевірку всіх типів дій: атак, використання здібностей, переміщення персонажів та роботи штучного інтелекту. Тестувалися різні сценарії бою, включаючи ситуації з множинними учасниками, одночасними діями та мережевими затримками.

Система збереження даних тестувалася на коректність серіалізації та десеріалізації, відновлення після збоїв, міграцію між версіями та роботу з пошкодженими файлами. Особлива увага приділялася тестуванню сценаріїв відновлення після несподіваного завершення роботи застосунку.

Тестування продуктивності проводилося на широкому спектрі пристроїв з різними технічними характеристиками (рис. 3.3.). Вимірювалися ключові метрики: частота кадрів, час завантаження, використання пам'яті, споживання батареї та час відгуку на дії користувача (рис 3.4.).

а)  б) 

Рис. 3.3. Завантаження Warbound в такі пристрої: а) Oppo A12 (ОС - Android); б) HP Elitebook 830 G3 (ОС – Windows).

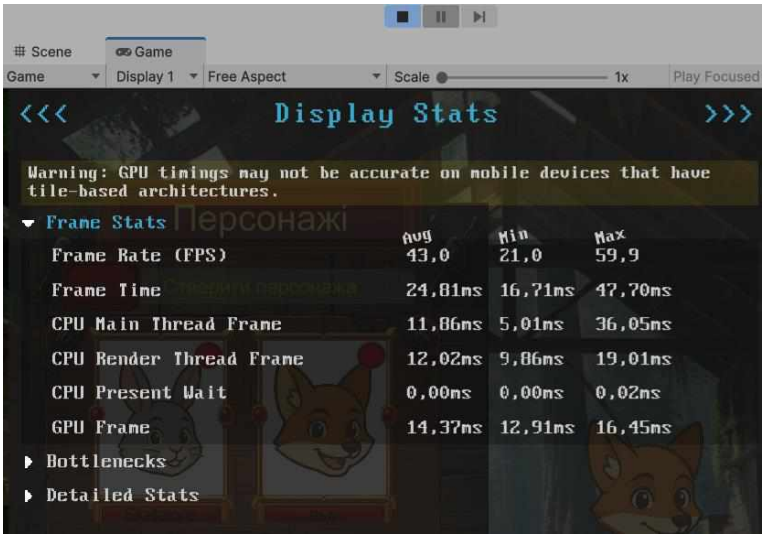


Рис. 3.4. Дослідження навантаження на мобільний процесор.

Результати тестування підтвердили ефективність розробленого кросплатформеного рішення, проте виявили потенціал для подальшого розвитку. Аналіз продуктивності застосунку показав середню частоту кадрів 43 FPS із коливаннями від 21 до 59.5 FPS, що свідчить про стабільну роботу системи з можливостями оптимізації. Розподіл навантаження демонструє, що графічний процесор є основним обмежувальним фактором продуктивності, що створює можливості для цільової оптимізації.

Кросплатформена архітектура Warbound на базі Unity забезпечує основу для масштабування на додаткові платформи. Тестування на Windows та Android підтвердило коректну роботу системи збереження даних через Unity Cloud Save та стабільність багатокористувацької взаємодії. Модульна структура застосунку дозволяє інтегрувати додаткові платформи без кардинальних змін у коді.

Виявлені технічні обмеження включають нестабільність частоти кадрів у ресурсомістких сценах та потребу оптимізації графічної підсистеми. Водночас максимальні значення близько 60 FPS підтверджують потенціал системи для досягнення стандартної продуктивності. Успішна реалізація покрокової бойової системи та інтеграція з хмарними сервісами створюють технологічну основу для розширення функціональності.

Перспективи розвитку включають оптимізацію шейдерів та текстур для покращення продуктивності на мобільних пристроях, розширення підтримки на Linux та macOS через наявну Unity-архітектуру, інтеграцію додаткових соціальних функцій для посилення багатокористувацької взаємодії та впровадження адаптивного балансування складності залежно від характеристик пристрою. Розроблена система збереження даних та кросплатформена синхронізація створюють фундамент для подальшого масштабування проєкту без втрати сумісності з існуючими платформами й створенням більшим функціональним можливостям.

Посилання на сторінку завантаження застосунку:

https://skeletonsnik.itch.io/warbound

Посилання на документацію еволюції розробки починаючи з 2024 року:

https://docs.google.com/document/d/1BliN6eYa3dZNlq3eXVLfRlCKEOqecHnd/edit

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено кросплатформений ігровий застосунок Warbound у середовищі Unity, що підтримує персональні комп'ютери Windows та мобільні пристрої Android. Застосунок реалізує концепцію двовимірного рольового середовища з покроковою системою взаємодії у фентезійному стилі.

Проведено аналіз технологій розробки кросплатформених застосунків та обґрунтовано вибір Unity як основного ігрового рушія. Визначено, що Unity є оптимальним рішенням завдяки потужним інструментам для 2D-графіки, інтегрованій підтримці багатокористувацьких функцій через Unity Services та можливостям хмарного збереження даних через Unity Cloud Save.

Розроблено архітектуру застосунку на основі багаторівневого шаблону проєктування з чітким розділенням на блоки геймплею, візуалізації, головних об'єктів та серверної частини. Використано ключові патерни проєктування для забезпечення модульності та розширюваності системи. Центральною моделлю даних є клас Player, який інкапсулює характеристики персонажа та забезпечує серіалізацію для збереження.

Створено систему збереження даних, що поєднує локальне кешування з використанням JSON-серіалізації та резервне копіювання. Реалізовано PlayerDatabase для управління даними гравців з підтримкою створення резервних копій та міграції між версіями. Система забезпечує надійність даних та можливість відновлення після збоїв.

Розроблено механіки покрокових боїв на основі сітки клітинок 3×5 для кожної команди з використанням компонентно-орієнтованої архітектури Unity. Реалізовано FightController для координації бойових процесів, PlayerCell для представлення клітинок поля бою та систему ефектів через StatusEffect. Бойова система включає базові атаки, спеціальні здібності та взаємодію з роботами.

Для оптимізації продуктивності застосовано ефективні структури даних та алгоритми. Використано двовимірний масив для представлення ігрового поля з

O(1) доступом за координатами, об'єктні пули для управління ресурсами та систему адаптивних налаштувань якості залежно від характеристик пристрою.

Створено систему створення та кастомізації персонажів через PlayerCreator, AvatarGenerator та CharacterCustomization з підтримкою п'яти категорій налаштування зовнішності. Реалізовано динамічне завантаження спрайтів з папки Resources та інтуїтивний інтерфейс через систему слайдерів.

Проведено комплексне тестування застосунку, включаючи функціональне тестування всіх компонентів та тестування продуктивності. Аналіз показав середню частоту кадрів 43 FPS з коливаннями від 21 до 59.5 FPS, що свідчить про стабільну роботу системи з потенціалом для оптимізації. Розподіл навантаження демонструє збалансованість між CPU та GPU з можливістю цільової оптимізації графічної підсистеми.

Застосунок успішно протестовано на платформах Windows та Android, підтверджено коректну роботу системи збереження даних та стабільність покрокової бойової системи. Виявлені можливості для подальшої оптимізації включають покращення роботи з шейдерами та текстурами, розширення підтримки додаткових платформ та інтеграцію соціальних функцій.

Таким чином, мету роботи досягнуто - створено функціональний кросплатформений ігровий застосунок, що демонструє ефективне використання Unity для розробки 2D-ігор з оптимізованою продуктивністю та надійною системою збереження даних.

.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Авада Медіа. Розробка ігор на Unity. URL: https://avada-media.ua/en/services/razrabotka-igr-na-unity/ (дата звернення: 17.04.2025).

2. Бойко О.В. Методологічні підходи до розробки сучасних програмних застосунків. Київ: Наука і Техніка, 2023. 288 с.

3. Ващенко П.Л. Оптимізація процесів збереження даних у кросплатформених застосунках. Інформаційні технології та кібербезпека. 2024. Вип. 5(12). С. 34-41.

4. Вікіпедія. Unity (ігровий рушій). URL: https://uk.wikipedia.org/wiki/Unity_(ігровий_рушій) (дата звернення: 20.03.2025).

5. Грищенко А.О., Потапова К.Р. Аналіз ефективності різних підходів до проєктування кросплатформених рішень. Вісник Національного технічного університету України «КПІ». Серія: Інформатика, управління та обчислювальна техніка. 2023. №40. С. 78-85.

6. IT Прогер. Основи движка Unity 2025 з нуля. URL: https://itproger.com/ua/course/unity (дата звернення: 03.05.2025).

7. Коваленко Д.С. Архітектурні патерни проєктування кросплатформених застосунків: теорія і практика. Одеса: ОНУ ім. І.І. Мечникова, 2024. 356 с.

8. Литвиненко О.В. Розробка багатокористувацьких застосунків: виклики та рішення. Київ: Комп'ютерні технології, 2023. 412 с.

9. Мельник І.С., Петренко В.О. Методологія розробки багатокористувацьких кросплатформених застосунків. Наукові записки НаУКМА. Комп'ютерні науки. 2023. Т. 6. С. 112-120.

10. Морозов О. В. Оптимізація бізнес-процесів програмних застосунків у фінансовій грамотності за допомогою інструменту unity. Математичні методи та цифрові технології в економіці. 2025.

11. Розробка ігор Unity 3D. Спільно Спейс. URL: https://spilno.space/unity_game_design (дата звернення: 11.02.2025).

12. Троценко С.М. Інструменти оптимізації продуктивності кросплатформених програмних продуктів. Вісник Харківського національного університету радіоелектроніки. 2024. №2. С. 45-53.

13. Унецький С.В. Сучасні технології розробки програмного забезпечення. Харків: Технічний університет, 2024. 298 с.

14. Юніті: Розробка Ігор на Провідній Платформі. URL: https://unity.softico.ua/ (дата звернення: 15.02.2025).

15. Яковенко Р.В. Оптимізація збереження даних у хмарних сервісах для програмних застосунків. Кібербезпека та інформаційні технології. 2023. №8. С. 67-75.

16. BookAuthority. The Best New Game Development Books To Read In 2025. URL: https://bookauthority.org/books/new-game-development-books (дата звернення: 02.03.2025).

17. Cognitive Market Research. Tactical RPG Market Report. URL: https://www.cognitivemarketresearch.com/tactical-rpg-market-report?/tactical-rpg-market-report (дата звернення: 19.04.2025).

18. Cubix Blogs. Best Game Development Engine For MMORPG. URL: https://www.cubix.co/blog/best-game-development-engine-for-mmorpg-games/ (дата звернення: 17.04.2025).

19. DataIntelo. Global Role-Playing Games Market. URL: https://dataintelo.com/report/global-role-playing-games-market (дата звернення: 03.05.2025).

20. Dev.to. From Mobile to Desktop: How Cross-Platform Apps Transform User Experiences. URL: https://dev.to/mohitb/from-mobile-to-desktop-how-cross-platform-apps-transform-user-experiences-4n50 (дата звернення: 12.03.2025).

21. FGFactory. 10 Best Mobile Games 2023 Made with Unity Engine. URL: https://fgfactory.com/top-10-unity-built-mobile-games-of-2023 (дата звернення: 16.03.2025).

22. Firebase. Get started with Cloud Storage for Unity. URL: https://firebase.google.com/docs/storage/unity/start (дата звернення: 03.05.2025).

23. Game-Ace. Your Full Guide to Unity Performance Optimization. URL: https://game-ace.com/blog/unity-performance-optimization/ (дата звернення: 03.05.2025).

24. GameDev Stack Exchange. Saving player's progress in a Unity game. URL: https://gamedev.stackexchange.com/questions/113829/saving-players-progress-in-a-unity-game (дата звернення: 03.05.2025).

25. GameDesignSkills. Combat Design, Mechanics and Systems for Satisfying Game Combat. URL: https://gamedesignskills.com/game-design/combat-design/ (дата звернення: 12.12.2024).

26. Gamedev DOU. NewZoo: Japanese Gamers Behavior. URL: https://gamedev.dou.ua/news/newzoo-japanese-gamers-behavior/ (дата звернення: 03.05.2025).

27. Geary D., Copes S. Cross-Platform Development with Unity. Cambridge: MIT Press, 2024. 478 p.

28. GeeksforGeeks. 10 Best Unity Books For Game Development[2025]. URL: https://www.geeksforgeeks.org/unity-books-for-game-development/ (дата звернення: 15.04.2025).

29. Hackr.io. The 10 Best Unity Books to Learn Game Development in 2025. URL: https://hackr.io/blog/best-unity-books (дата звернення: 13.04.2025).

30. Halpern J. Database Implementation for Cross-Platform Applications. Journal of Software Development and Technologies. 2023. Vol. 8(4). P. 78-92.

31. Info Gamer Hub. Cloud Saving Service - Cross Platform Native Plugins: Essential Kit. URL: https://www.infogamerhub.com/cross-platform-native-plugins-cloud/ (дата звернення: 21.03.2025).

32. Itch.io. Top games made with Unity tagged Turn-Based Combat. URL: https://itch.io/games/made-with-unity/tag-turn-based-combat (дата звернення: 21.03.2025).

33. Itch.io. Top games tagged 2D and slay-the-spire. URL: https://itch.io/games/tag-2d/tag-slay-the-spire (дата звернення: 21.03.2025).

34. Khan S., Smith M. Cloud-Based Data Solutions for Modern Applications. IEEE Journal of Cloud Computing. 2023. Vol. 12(3). P. 45-52.

35. Makaka.org. Unity Optimization Tips — Optimize Unity Game, Tutorial 2025. URL: https://makaka.org/unity-tutorials/optimization (дата звернення: 18.12.2024).

36. Mezha.Media. Made in Ukraine: Ukrainian Games 2023–2025. Patch 1.2. URL: https://mezha.media/en/articles/made-in-ukraine-ukrainian-games-2023-2025-patch-1-2-19-games/ (дата звернення: 27.10.2025).

37. MMORPG.GG. The 18 Best 2D MMOs to Play in 2025. URL: https://mmorpg.gg/the-best-2d-mmos/ (дата звернення: 30.03.2025).

38. OST Agency. Cross-platform apps in 2025: the secret to speed, scalability, and cost efficiency. URL: https://blogs.ost.agency/cross-platform-apps-in-2025-the-secret-to-speed-scalability-and-cost-efficiency/ (дата звернення: 06.02.2025).

39. Parker J., Wilson D. Performance Optimization Techniques for Multi-platform Applications. International Journal of Software Engineering. 2023. Vol. 14(2). P. 234-248.

40. Quora. For game development, which 2D and 3D cross-platform game engines are best? URL: https://www.quora.com/For-game-development-which-2D-and-3D-cross-platform-game-engines-are-best-Corona-2D-and-Unity-3D (дата звернення: 01.05.2025).

41. Unity Documentation. PSD Importer. URL: https://docs.unity3d.com/Packages/com.unity.2d.psdimporter@1.1/manual/index.html (дата звернення: 15.02.2025).

ДОДАТКИ

Додаток А

Програмна реалізація сцени SampleScene

Скрипт PlayerCell.cs

```
using UnityEngine;
using UnityEngine.UI;
using Core.Data;

public class PlayerCell : MonoBehaviour
{
  [Header("Налаштування клітинки")]
   public string cellId;  // Ідентифікатор клітинки (наприклад, A1, B2)
   public bool isEnemy;   // Чи є персонаж ворогом

  [Header("Візуальні елементи")]
   public GameObject indicator;  // Індикатор для всіх станів
   public Image healthBar;
   public Text nicknameText;
   public Text healthText;

   // Стан клітинки
   private bool isSelected = false;
   private bool hasTurn = true;
   private bool isDefeated = false;

   // Компоненти
   private ColorChanger colorChanger;
   private Button cellButton;
   private Player playerData;
   private int currentHealth;

   private void Awake()
   {
      colorChanger = GetComponent<ColorChanger>();
      if (colorChanger == null)
      {
         colorChanger = gameObject.AddComponent<ColorChanger>();
      }

      cellButton = GetComponent<Button>();
      if (cellButton == null) {
```

Продовження додатка А

```
            cellButton = gameObject.AddComponent<Button>(); }
        cellButton.onClick.AddListener(OnClick);
        if (indicator == null) // Створюємо індикатор, якщо його немає {
            indicator = new GameObject("Indicator");
            indicator.transform.SetParent(transform);
            indicator.transform.localPosition = Vector3.zero;
            RectTransform rectTransform = indicator.AddComponent<RectTransform>(); // Додаємо компоненти для відображення
            rectTransform.anchorMin = Vector2.zero;
            rectTransform.anchorMax = Vector2.one;
            rectTransform.offsetMin = Vector2.zero;
            rectTransform.offsetMax = Vector2.zero;
            Image indicatorImage = indicator.AddComponent<Image>();
            indicatorImage.color = new Color(1, 1, 1, 0.2f); }
        if (colorChanger != null) // Встановлюємо нормальний стан {
            colorChanger.SetNormalState(); } } }

    public void Init(string cellId, Player playerData, CharacterData characterData = null) // Ініціалізація з даними персонажа {
        this.cellId = cellId;
        this.playerData = playerData;
        if (playerData != null) // Встановлюємо початкове здоров'я {
            currentHealth = playerData.GetHealth(); } else {
            currentHealth = 100; // За замовчуванням } 
        // Оновлюємо відображення
        UpdateUI(); }

    private void UpdateUI() // Оновлює відображення UI елементів {
        // Оновлюємо ім'я персонажа
        if (nicknameText != null && playerData != null) {
            nicknameText.text = playerData.GetNickname(); }
        // Оновлюємо полосу здоров'я
        if (healthBar != null && playerData != null) {
            healthBar.fillAmount = (float)currentHealth / playerData.GetHealth(); }
        // Оновлюємо текст здоров'я
        if (healthText != null && playerData != null) {
            healthText.text = $"{currentHealth}/{playerData.GetHealth()}"; } }

    public void OnClick() // Обробник кліку користувача на клітинку {
        if (isDefeated) return;
        FightController fightController = FightController.Instance;
```

```
            if (fightController == null) return;
            if (fightController.IsAttackMode) {
                if (isEnemy) // Перевірка умови, чи в режимі атаки гравець {
                    fightController.AttackPlayer(this); } }
            else { if (!isEnemy) // Звичайний клік - вибір або інформація {
                    fightController.SelectPlayerCell(this); } else {
                    ShowCharacterInfo(); } } }

        private void ShowCharacterInfo() // Показує інформацію про персонажа
            CharacterInfoPanel infoPanel = FindFirstObjectByType<CharacterInfoPanel>();// Шукаємо інформаційну панель
            if (infoPanel != null && playerData != null) {
                infoPanel.ShowCharacterInfo(playerData, null); }
            Debug.Log($"Інформація про персонажа в клітинці {cellId}: {playerData?.GetNickname()}"); }
        public void TakeDamage(int damage) {  if (playerData != null) {
                currentHealth -= damage; // Застосовуємо пошкодження
      // Перевіряємо, чи персонаж ще живий
                if (currentHealth <= 0) {
                    currentHealth = 0;
                    SetDefeated(true); } // Оновлюємо відображення
                UpdateUI() ;} }

        public void SetSelected(bool selected) {
            isSelected = selected;
            if (colorChanger != null) {  if (selected) {
                    colorChanger.SetSelectedState();}
                else if (isDefeated) {
                    colorChanger.SetDefeatedState(); }
                else {
                    colorChanger.SetNormalState(); } } }

        public void SetDefeated(bool defeated)    // Встановлює статус клітинки як переможеної {
            isDefeated = defeated;
            if (colorChanger != null) { if (defeated) {
                    colorChanger.SetDefeatedState(); } else {
                    colorChanger.SetNormalState(); } } }
        public bool IsDefeated() { return isDefeated; }
        public void UseTurn() // Використання ходу { hasTurn = false; }
        public bool HasTurn() { return hasTurn && !isDefeated; }
```

```
    public void ResetTurn() { hasTurn = true; } public void Clear() // Очищає клітинку персонажа для завершення битви {
        // Знищуємо всіх дітей (персонаж і його компоненти)
        foreach (Transform child in transform) {
            if (child.gameObject != indicator) { Destroy(child.gameObject); } }
        isSelected = false;
        hasTurn = true;
        isDefeated = false;
        playerData = null;
        currentHealth = 0;
        if (colorChanger != null) {
            colorChanger.SetNormalState(); } }
    public Player GetPlayerData() // Повертає дані гравця { return playerData; }
    private void OnDestroy() { if (cellButton != null) // Відписуємося від подій {
            cellButton.onClick.RemoveListener(OnClick); } } }
```

Скрипт FightController.cs (скорочений варіант)

```
public class FightController : MonoBehaviour {
    public static FightController Instance { get; private set; }
    private int currentTurn = 0;
    private bool isBattleActive = false;
    private List<BattleAction> pendingActions = new List<BattleAction>();
    void Awake() {
        // Реалізація патерну Singleton
        if (Instance == null) {
            Instance = this;
            DontDestroyOnLoad(gameObject); } else {
            Destroy(gameObject); } }
    public void StartBattle() {
        isBattleActive = true;
        currentTurn = 0;
        // Ініціалізація поля бою
        InitializeBattlefield();
        // Початок фази планування
        StartPlanningPhase();       }       }
```

Скрипт BattleAction.cs

```
[System.Serializable]
public class BattleAction {
    public enum ActionType { Attack, Defend, UseSkill, Move }
```

```
    public ActionType type;
    public int sourceId;
    public int targetId;
    public int skillId; // для UseSkill
    public Vector2Int targetPosition; // для Move
    public float priority; // визначає порядок виконання
    public BattleAction(ActionType actionType, int source, int target) {
        type = actionType;
        sourceId = source;
        targetId = target;
        // Розрахунок пріоритету на основі швидкості персонажа
        priority = CalculatePriority(); }    }
```

Скрипт StatusEffect.cs

```
using UnityEngine;
// Клас для представлення ефектів статусу (бафів/дебафів)
[System.Serializable]
public class StatusEffect
{
    public string id;          // Унікальний ідентифікатор ефекту
    public string name;        // Назва ефекту
    public string description; // Опис ефекту
    public Sprite icon;        // Іконка ефекту
    public int duration;       // Тривалість ефекту в ходах
    public bool isPositive;    // Чи є ефект позитивним  чи негативним

    // Модифікатори характеристик
    public int healthModifier;         // Модифікатор здоров'я
    public int defenseModifier;        // Модифікатор захисту
    public float absorptionModifier;   // Модифікатор поглинання шкоди
    public int damageModifier;         // Модифікатор шкоди
    public float critChanceModifier;   // Модифікатор шансу критичного удару
    public float critMultiplierModifier; // коефіцієнт критичного удару
    // Конструктор для створення ефекту статусу
    public StatusEffect(string id, string name, string description, Sprite icon, int
duration, bool isPositive)  { this.id = id;  this.name = name;
        this.description = description;
        this.icon = icon;
        this.duration = duration;
        this.isPositive = isPositive; }
```

```
    public static StatusEffect CreateHealingEffect(int amount, int duration) {
        StatusEffect effect = new StatusEffect(
            "healing",
            "Зцілення",
            $"Відновлює {amount} HP за хід",
            Resources.Load<Sprite>("Icons/Heal"), // Шлях до іконки
            duration,
            true);
    effect.healthModifier = amount; return effect;    }
    // Створює ефект отрути
      public static StatusEffect CreatePoisonEffect(int damage, int duration)    {
     StatusEffect effect = new StatusEffect(
            "poison",
            "Отрута",
            $"Завдає {damage} шкоди за хід",
            Resources.Load<Sprite>("Icons/Poison"), // Шлях до іконки
            duration,
            false ); // негативний ефект
        effect.healthModifier = -damage;
        return effect;    }
      // Створює ефект посилення захисту
public static StatusEffect CreateDefenseBoostEffect(int amount, int duration)  {
        StatusEffect effect = new StatusEffect(
            "defense_boost",
            "Посилений захист",
            $"Збільшує захист на {amount}",
            Resources.Load<Sprite>("Icons/Shield"), // Шлях до іконки
            duration,
            true ); // Позитивний ефект
        effect.defenseModifier = amount;
        return effect; }
      // Створює ефект посилення атаки
     public static StatusEffect CreateAttackBoostEffect(int amount, int duration) {
        StatusEffect effect = new StatusEffect(
            "attack_boost",
            "Посилена атака",
            $"Збільшує пошкодження на {amount}",
            Resources.Load<Sprite>("Icons/Sword"), // Шлях до іконки
            duration,
            true );
        effect.damageModifier = amount;
        return effect; } }
```

Додаток Б

Програмна реалізація сцени SceneCreator

Скрипт PlayerCreator.cs (скорочений варіант)

```
public class PlayerCreator : MonoBehaviour {
  [Header("UI Elements")]
   public InputField nicknameInput;
   public Transform avatarContainer;
   public Button createButton;
   public Button randomizeButton;
  [Header("Components")]
   private AvatarGenerator avatarGen;
   private CharacterCustomization customization;
   private PlayerDatabase database;

   private Player currentPlayer;
   void Start(){
      avatarGen = GetComponent<AvatarGenerator>();
      customization = GetComponent<CharacterCustomization>();
      database = PlayerDatabase.Instance;
      createButton.onClick.AddListener(CreateCharacter);
      randomizeButton.onClick.AddListener(RandomizeAppearance); }

   void CreateCharacter()        {
      string nickname = nicknameInput.text.Trim();
      if (string.IsNullOrEmpty(nickname) || nickname.Length < 3) // Валідація
нікнейму       {
         ShowError("Нікнейм повинен містити мінімум 3 символи");
         return; }
      // Перевірка унікальності імені користувача
      if (database.PlayerExists(nickname)) {
         ShowError("Гравець з таким нікнеймом вже існує");
         return; }

      // Створення нового гравця
      currentPlayer = new Player(nickname);
      currentPlayer.customization = customization.GetCurrentSettings();
      // Збереження в базу даних
      database.AddPlayer(currentPlayer);
      database.SaveDatabase();
      SceneManager.LoadScene("MainMenu");    // Перехід до головного меню
}       }
```

Продовження додатка Б

Скрипт CharacterCustomization.cs

```
using System.Collections.Generic;
using UnityEngine;
[System.Serializable]
public class BodyPartOption {  public string partName; public Sprite sprite; }
[System.Serializable]
public class BodyPartCategory {
    public string categoryName; // "Голова", "Тіло", тощо
    public List<BodyPartOption> options = new List<BodyPartOption>();
    public int selectedIndex = 0; }

public class CharacterCustomization : MonoBehaviour {
   [Header("Посилання на об'єкти відображення")]
    public SpriteRenderer headRenderer;
    public SpriteRenderer bodyRenderer;
    public SpriteRenderer armsRenderer;
    public SpriteRenderer legsRenderer;
    public SpriteRenderer clothesRenderer;
   [Header("Шляхи до ресурсів")]
    public string resourceBasePath = "Person/";
   [Header("Налаштування відображення")]
    public bool forceVisible = true;
    public List<BodyPartCategory> bodyPartCategories = new
List<BodyPartCategory>();    // Категорії частин тіла
    void Awake() {
        InitializeBodyParts(); }
    void Start() {
        ApplyCharacterCustomization();
        // Гарантуємо, що об'єкт активний і видимий
        if (forceVisible) {
            EnsureVisible(); } }
    // Гарантує, що всі рендерери видимі та активні
    private void EnsureVisible() {
        SpriteRenderer[] renderers = new SpriteRenderer[] {
            headRenderer, bodyRenderer, armsRenderer, legsRenderer,
clothesRenderer };
        foreach (SpriteRenderer renderer in renderers)  {
            if (renderer != null) {
                // Переконуємося, що об'єкт активний
                renderer.gameObject.SetActive(true);
                renderer.enabled = true;  // Робимо рендерер видимим
                // Переконуємося, що всі батьківські об'єкти активні
```

Продовження додатка Б

```
                Transform parent = renderer.transform.parent;
                while (parent != null) {
                    parent.gameObject.SetActive(true);
                    parent = parent.parent; }
                // Явно встановлюємо колір без прозорості
                Color color = renderer.color;
                color.a = 1f;
                renderer.color = color; } } }
        // Ініціалізація категорій частин тіла та завантаження спрайтів
        private void InitializeBodyParts() {
            // Створюємо категорії частин тіла
            string[] categoryNames = { "Голова", "Тіло", "Руки", "Ноги", "Одяг" };
foreach (string categoryName in categoryNames) {
                // Завантажуємо спрайти з папки ресурсів
                Sprite[] sprites =
Resources.LoadAll<Sprite>($"{resourceBasePath}{categoryName}");
                Debug.Log($"Намагаємось завантажити спрайти з папки:
{resourceBasePath}{categoryName}");
                Debug.Log($"Знайдено спрайтів: {(sprites != null ? sprites.Length :
0)}");
        if (sprites != null && sprites.Length > 0) {
                // Виводимо імена знайдених спрайтів у режимі розробки
                foreach (Sprite sprite in sprites) {
                    Debug.Log($" - Знайдено спрайт: {sprite.name}"); }
                BodyPartCategory category = new BodyPartCategory {
categoryName = categoryName };// Створюємо нову категорію
                for (int i = 0; i < sprites.Length; i++)  // Додаємо опції для цієї
категорії
                { category.options.Add(new BodyPartOption {
                        partName = $"{categoryName}_{i}",
                        sprite = sprites[i] });
                bodyPartCategories.Add(category);
                Debug.Log($"Завантажено {sprites.Length} спрайтів для
'{categoryName}'"); } else {
                Debug.LogWarning($"Не знайдено спрайтів для '{categoryName}'
у папці {resourceBasePath}{categoryName}");
                bodyPartCategories.Add(new BodyPartCategory { categoryName =
categoryName }); } } }
        // Встановлює частину тіла за індексом
        public void SetBodyPart(string categoryName, int index) {
            BodyPartCategory category = bodyPartCategories.Find(c =>
c.categoryName == categoryName);
```

Продовження додатка Б

```
            if (category != null && index >= 0 && index < category.options.Count) {
                category.selectedIndex = index;
                ApplyCharacterCustomization(); } }
        // Встановлює частину тіла за значенням повзунка (0-1)
        public void SetBodyPartBySliderValue(string categoryName, float
sliderValue) {
            BodyPartCategory category = bodyPartCategories.Find(c =>
c.categoryName == categoryName);
        if (category != null && category.options.Count > 0) {
                // Перетворюємо значення повзунка (0-1) в індекс (0-count-1)
               int index = Mathf.FloorToInt(sliderValue * (category.options.Count - 1));
            category.selectedIndex = Mathf.Clamp(index, 0, category.options.Count - 1);
                ApplyCharacterCustomization(); } }
        // Застосовує поточні налаштування до моделі персонажа
        public void ApplyCharacterCustomization() {
      foreach (BodyPartCategory category in bodyPartCategories)  {
                if (category.options.Count == 0 || category.selectedIndex < 0 ||
                  category.selectedIndex >= category.options.Count) continue;
                BodyPartOption selectedOption =
category.options[category.selectedIndex];
                // Призначаємо спрайт відповідному рендереру
                switch (category.categoryName) {
                  case "Голова": if (headRenderer != null) {
                        headRenderer.sprite = selectedOption.sprite;
                        headRenderer.enabled = true;
                        headRenderer.gameObject.SetActive(true);  } break;
                  case "Тіло": if (bodyRenderer != null) {
                        bodyRenderer.sprite = selectedOption.sprite;
                        bodyRenderer.enabled = true;
                        bodyRenderer.gameObject.SetActive(true); } break;
                  case "Руки": if (armsRenderer != null) {
                        armsRenderer.sprite = selectedOption.sprite;
                        armsRenderer.enabled = true;
                        armsRenderer.gameObject.SetActive(true); }
                     break;
                 case "Ноги": if (legsRenderer != null) {
                        legsRenderer.sprite = selectedOption.sprite;
                        legsRenderer.enabled = true;
                        legsRenderer.gameObject.SetActive(true); } break;
                  case "Одяг": if (clothesRenderer != null) {
                        clothesRenderer.sprite = selectedOption.sprite;
                        clothesRenderer.enabled = true;
```

Продовження додатка Б

```
                    clothesRenderer.gameObject.SetActive(true); } break; }
        FixZForRenderers(); }
            if (forceVisible) // Переконуємося, що все видно
            { EnsureVisible(); } }
        // Отримує поточний вибір усіх частин персонажа
        public Dictionary<string, int> GetCharacterCustomization() {
            Dictionary<string, int> result = new Dictionary<string, int>();
            foreach (BodyPartCategory category in bodyPartCategories) {
                result[category.categoryName] = category.selectedIndex; }
            return result; }
        // Встановлює вибір усіх частин персонажа
        public void SetCharacterCustomization(Dictionary<string, int>
customization) {
            if (customization == null) return;
            foreach (KeyValuePair<string, int> item in customization) {
                SetBodyPart(item.Key, item.Value); }
        ApplyCharacterCustomization(); // Застосовуємо кастомізацію до моделі
        }
        // Встановлює Z=0 для всіх SpriteRenderer в об'єкті
        private void FixZForRenderers() {
            SpriteRenderer[] renderers =
GetComponentsInChildren<SpriteRenderer>();
            foreach (SpriteRenderer renderer in renderers) {
                if (renderer != null)  {
                    Vector3 pos = renderer.transform.localPosition;
                    pos.z = 0;
                    renderer.transform.localPosition = pos;  }  }  }  }
```

Додаток В

Програмна реалізація бази даних

Скрипт PlayerDatabase.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using UnityEngine;
using Core.Data;
public class PlayerDatabase : MonoBehaviour {
private static PlayerDatabase instance;
   public static PlayerDatabase Instance    {
      get  { if (instance == null) {
            instance = FindFirstObjectByType<PlayerDatabase>();
            if (instance == null) {
               GameObject go = new GameObject("PlayerDatabase");
               instance = go.AddComponent<PlayerDatabase>();
            }   } return instance;  }    }

  [Header("Налаштування збереження")]
  [SerializeField] private string databaseFilename = "game_database.json";
  [SerializeField] private bool createBackups = true;
  [SerializeField] private int maxBackupCount = 5;
   private string databaseDirectoryPath;
   private string databaseFilePath;
   public GameData gameData = new GameData();
   private Account currentAccount;
   private Player currentPlayer;
   public Action OnDatabaseUpdated;
   public Action<string> OnError;
  [Serializable]
   public class GameData{
 public List<Account> accounts = new List<Account>(); }
   private void Awake() {
      if (instance != null && instance != this)
      {   Destroy(gameObject);
         return; }
      instance = this;
      transform.SetParent(null);
      DontDestroyOnLoad(gameObject);
      InitializePaths();
      LoadDatabase(); }
```

Продовження додатка В

```
        private void InitializePaths()    {       string dataPath = Application.dataPath;
            databaseDirectoryPath = Path.Combine(dataPath, "Data", "Database");
            if (!Directory.Exists(databaseDirectoryPath)) {
                Directory.CreateDirectory(databaseDirectoryPath);
                Debug.Log($"✅ Створено директорію для бази даних:
{databaseDirectoryPath}"); }
            databaseFilePath = Path.Combine(databaseDirectoryPath,
databaseFilename); }
        // Методи для роботи з аккаунтами
            public void CreateAccount(string accountName) {
        var account = new Account();
            account.accountName = accountName;
            account.accountId = System.Guid.NewGuid().ToString();
            gameData.accounts.Add(account);
            SaveDatabase();    }

            public Account GetAccount(string accountId) {
            return gameData.accounts.Find(a => a.accountId == accountId);  }

 public void SetCurrentAccount(Account account){ currentAccount = account; }
            public Account GetCurrentAccount() { return currentAccount; }
            public void AddPlayer(Player player) // Методи для роботи з гравцями
            {  if (currentAccount != null) {
                player.Id = currentAccount.players.Count + 1;
                currentAccount.players.Add(player);
                Debug.Log($"➕ Додано гравця: {player.GetNickname()}");
                Debug.Log($"👤 ID гравця: {player.Id}");
                Debug.Log($"👥 Всього гравців в акаунті
{currentAccount.accountName}: {currentAccount.players.Count}");
                Debug.Log($"🏢 Всього акаунтів: {gameData.accounts.Count}");
                SaveDatabase();
                OnDatabaseUpdated?.Invoke(); } else {
                Debug.LogError("❌ Поточний акаунт не встановлено!"); } }
            public void AddPlayerToAccount(Player player)  {  AddPlayer(player); }
            public Player GetPlayer(string nickname) {
        if (currentAccount == null) return null;
            return currentAccount.players.Find(p => p.GetNickname() == nickname); }
         public Player GetPlayerByNickname(string nickname) {
            return GetPlayer(nickname); }
            public void SetCurrentPlayer(Player player)  { currentPlayer = player;  }
```

Продовження додатка В

```
        public Player GetCurrentPlayer()   {   return currentPlayer;    }
        public List<Player> GetAllPlayers(){
      if (currentAccount == null) return new List<Player>();
          return new List<Player>(currentAccount.players);   }
        public List<Player> GetAllPlayersForCurrentAccount() {
      return GetAllPlayers(); }
        public void UpdatePlayer(Player updatedPlayer) {
          if (currentAccount != null) {
            int index = currentAccount.players.FindIndex(p => p.Id ==
updatedPlayer.Id);
            if (index >= 0) {
              currentAccount.players[index] = updatedPlayer;
              if (currentPlayer != null && currentPlayer.Id == updatedPlayer.Id){
      currentPlayer = updatedPlayer; }
              SaveDatabase();
              OnDatabaseUpdated?.Invoke();}  }  }

        public void RemovePlayer(string nickname) { if (currentAccount != null) {
            Player playerToRemove = currentAccount.players.Find(p =>
p.GetNickname() == nickname);
            if (playerToRemove != null) {
              currentAccount.players.Remove(playerToRemove);
              SaveDatabase();
              OnDatabaseUpdated?.Invoke(); } } }
        // Методи для роботи з роботами
        public void AddRobotToPlayer(RobotData robot) {
          if (currentPlayer != null) {
            currentPlayer.robot = robot;
            SaveDatabase(); } }
        public RobotData GetRobot(string robotId) {
       if (currentPlayer == null) return null;
          return currentPlayer.robot?.robotId == robotId ? currentPlayer.robot :
null; }
        public List<RobotData> GetAllRobotsForCurrentPlayer() {
          List<RobotData> robots = new List<RobotData>();
          if (currentPlayer != null && currentPlayer.robot != null) {
            robots.Add(currentPlayer.robot); }
          return robots;  }
        // Методи збереження/завантаження
        public void SaveDatabase() { try {
```

Продовження додатка В

```
            Debug.Log($"🔄 Спроба збереження бази даних. Кількість акаунтів: {gameData.accounts.Count}");
            if (createBackups && File.Exists(databaseFilePath)) {
               BackupDatabase(); }
            string json = JsonUtility.ToJson(gameData, true);
            File.WriteAllText(databaseFilePath, json);
            Debug.Log($"✅ База даних збережена успішно в: {databaseFilePath}");
            Debug.Log($"📄 Вміст JSON: {json}");
            OnDatabaseUpdated?.Invoke(); }
         catch (Exception e) {
            Debug.LogError($"❌ Помилка при збереженні бази даних: {e.Message}");
            Debug.LogError($"🔍 Stack trace: {e.StackTrace}");
            OnError?.Invoke($"Помилка збереження: {e.Message}"); } }
      public void LoadDatabase()  {  try  {
            if (File.Exists(databaseFilePath)){
               string json = File.ReadAllText(databaseFilePath);
               gameData = JsonUtility.FromJson<GameData>(json);
               Debug.Log($"✅ База даних завантажена успішно. Знайдено {gameData.accounts.Count} аккаунтів.");
               OnDatabaseUpdated?.Invoke(); } else {
               Debug.Log("ℹ️ Файл бази даних не знайдено. Буде створено нову базу даних."); gameData = new GameData(); } }  catch (Exception e){
            Debug.LogError($"❌ Помилка при завантаженні бази даних: {e.Message}");
            OnError?.Invoke($"Помилка завантаження: {e.Message}");
            gameData = new GameData(); }  }
      public bool RestoreFromBackup(string backupFileName) { try  {
            string backupPath = Path.Combine(databaseDirectoryPath, backupFileName);
            if (File.Exists(backupPath)) {
               // Копіюємо дані з резервної копії в основний файл
               File.Copy(backupPath, databaseFilePath, true);
               LoadDatabase();// Перезавантажуємо дані
               Debug.Log($"✅ База даних відновлена з резервної копії: {backupFileName}");
               return true; } else {
               Debug.LogError($"❌ Резервна копія не знайдена: {backupFileName}"); return false;  } }
         catch (Exception e) {
```

Продовження додатка В

```
            Debug.LogError($"❌ Помилка при відновленні з резервної копії: {e.Message}");
            OnError?.Invoke($"Помилка відновлення: {e.Message}");
            return false; } }
        private void BackupDatabase() { try {
            string timestamp = DateTime.Now.ToString("yyyyMMdd_HHmmss");
            string backupPath = Path.Combine(databaseDirectoryPath, $"game_database_backup_{timestamp}.json");
            if (File.Exists(databaseFilePath)) {

                File.Copy(databaseFilePath, backupPath); // Створюємо копію
                Debug.Log($"✅ Створено резервну копію бази даних: {backupPath}");

      #if UNITY_EDITOR // Зареєструємо файл в AssetDatabase для Unity Editor
                UnityEditor.AssetDatabase.Refresh(); #endif
                CleanupOldBackups(); // Видаляємо старі копії
            } }  catch (Exception e) {
            Debug.LogError($"❌ Помилка при створенні резервної копії: {e.Message}");
        OnError?.Invoke($"Помилка резервного копіювання: {e.Message}");}  }
        public List<string> GetAvailableBackups() {
            List<string> backups = new List<string>();  try {
                if (Directory.Exists(databaseDirectoryPath)) {
                    string[] files = Directory.GetFiles(databaseDirectoryPath, "game_database_backup_*.json"); foreach (string file in files) {
                        backups.Add(Path.GetFileName(file));  } } } catch (Exception e) {
            Debug.LogError($"❌ Помилка при отриманні списку резервних копій: {e.Message}"); } return backups; }private void CleanupOldBackups() { try {
            string[] backupFiles = Directory.GetFiles(databaseDirectoryPath, "game_database_backup_*.json");
            if (backupFiles.Length <= maxBackupCount) return;
            Array.Sort(backupFiles, (a, b) => File.GetCreationTime(a).CompareTo(File.GetCreationTime(b)));
            int filesToDelete = backupFiles.Length - maxBackupCount;
            for (int i = 0; i < filesToDelete; i++)  {
                File.Delete(backupFiles[i]);
                Debug.Log($"🗑️ Видалено стару резервну копію: {Path.GetFileName(backupFiles[i])}");  }  }  catch (Exception e) {
            Debug.LogError($"❌ Помилка при очищенні старих резервних копій: {e.Message}"); } } }
```