БАКАЛАВРСЬКА РОБОТА

БР. ІП - 23.00.00.000 ПЗ

Група ІП-22-3

Жирун Володимир

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Жирун Володимир Михайлович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Розробка ігор з використанням Unity 3D

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Жирун В.М._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____ Піх Марія Михайлівна, асистент
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Жируну Володимиру Михайловичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) **" Розробка ігор з використанням Unity 3D”**

керівник проекту (роботи) Піх Марія Михайлівна, асистент

затверджені наказом вищого навчального закладу від “ 28 ” травня 2026 р. № 252/7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

Приклад інтерфейсу Unity (рис.1.1, ст.14)
2. Діаграма варіантів використань (рис.2.1, ст.29)
3. Діаграма послідовності підключення гравців (рис.2.3, ст.32)
4. Діаграма активності (рис.2.11, ст.46)
5. Скріншот гри (рис.3.1, ст.51)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	27.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.04.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	09.06.2026	виконано

Студент _____Жирун В.М.

Керівник роботи _____Піх М..М.
(підпис)

АНОТАЦІЯ

Бакалаврська робота обсягом 76 сторінки, включає 20 рисунків, 4 таблиць і 35бібліографічних джерел.

Метою роботи є розробка прототипу багатокористувацької гри з використанням Unity 3D та Photon Engine для забезпечення мережевої взаємодії.

Об'єкт дослідження: процес розробки програмного забезпечення для багатокористувацьких ігор.

Предмет дослідження: технології та інструменти реалізації клієнт-серверної взаємодії в Unity 3D з використанням Photon Engine.

Результати дослідження: Розроблено прототип багатокористувацької гри, який включає підключення до сервера, створення кімнат, синхронізацію гравців, підбір предметів, ігровий таймер та інтерфейс користувача.

У першому розділі – виконано аналіз ігрових рушіїв та мережевих рішень, обґрунтовано вибір Unity 3D та Photon Engine.

У другому розділі – спроєктовано архітектуру, реалізовано логіку гри на C#, виконано інтеграцію Photon Engine, створено інтерфейс користувача.

У третьому розділі – проведено тестування, оцінку продуктивності, аналіз проблемних місць та визначено перспективи розвитку.

Висновок: Поєднання Unity 3D та Photon Engine забезпечує стабільну роботу прототипу при 4 гравцях (FPS ≥ 56, затримка ≤ 48 мс).

КЛЮЧОВІ СЛОВА: UNITY 3D, PHOTON ENGINE, БАГАТОКОРИСТУВАЦЬКА ГРА, C#, СИНХРОНІЗАЦІЯ, ТЕСТУВАННЯ.

ANNOTATION

Bachelor's thesis of 76 pages, includes 20 figures, 4 tables and 35 bibliographic sources.

The goal of the work is to develop a multiplayer game prototype using Unity 3D and Photon Engine for network interaction. Object of study: software development process for multiplayer games.

Subject of study: technologies and tools for client-server interaction in Unity 3D using Photon Engine.

Research results: A multiplayer game prototype has been developed, including server connection, room creation, player synchronization, item pickup, game timer, and user interface.

Chapter 1 analyzes game engines and networking solutions, justifying the choice of Unity 3D and Photon Engine.

Chapter 2 designs the architecture, implements game logic in C#, integrates Photon Engine, and creates the user interface.

Chapter 3 performs testing, performance evaluation, problem analysis, and identifies development prospects.

Conclusion: The combination of Unity 3D and Photon Engine ensures stable operation of the prototype with 4 players (FPS $\geq$ 56, latency $\leq$ 48 ms).

KEYWORDS: UNITY 3D, PHOTON ENGINE, MULTIPLAYER GAME, C#, SYNCHRONIZATION, TESTING.

ЗМІСТ

ВСТУП……8

РОЗДІЛ 1. АНАЛІЗ ІНСТРУМЕНТІВ ТА ОБҐРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ.....10

1.1 Огляд сучасних інструментів для розробки комп’ютерних ігор.....10

1.2 Порівняльний аналіз сучасних ігрових рушіїв.....12

1.3 Аналіз архітектур багатокористувацьких ігор.....17

1.4 Огляд мережевих рішень для Unity 3D.....20

1.5 Обґрунтування вибору Unity 3D та Photon Engine.....23

1.6 Висновки по розділу ….....26

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОТОТИПУ ГРИ….....27

2.1 Визначення жанру, платформи та цільової аудиторії гри.....27

2.2 Проєктування архітектури клієнт-серверної взаємодії.....29

2.3 Реалізація базової логіки гри засобами C#.....32

2.4 Інтеграція Photon Engine у Unity-проєкт.....37

2.5 Розробка інтерфейсу користувача гри.....40

2.6 Опис структури програми та алгоритмів її функціонування.....42

2.7 Висновки по розділу 47

РОЗДІЛ 3. ТЕСТУВАННЯ, ОЦІНКА ТА ОПТИМІЗАЦІЯ ПРОГРАМНОГО ПРОДКТУ.....49

3.1 Методологія тестування ігрового програмного забезпечення.....49

3.2 Тестування функціональності реалізованої гри.....51

3.3 Оцінка продуктивності та стабільності мережевої взаємодії.....53

3.4 Аналіз проблемних місць та способи їх усунення.....55

3.5 Перспективи розвитку та вдосконалення програмного продукту.....57

					БР.ІП –23.00.00.000 ПЗ		
Змн.	Арк.	№ докум.	Підпис	Дата			
Розроб.		Жирун В.М.			Розробка ігор з використанням Unity 3D	Літ.	Арк. 7 / Акрушів
Перевір.		Піх М.М.					
Реценз.		Процюк Г.Я.М.					
Н. Контр.		Піх М.М.			Пояснювальна записка	ІФНТУНГ ІП-22-1	
Затверд.		Бандура В. В.					

3.6 Висновки по розділу 59
ВИСНОВКИ 61
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 63
ДОДАТКИ

ВСТУП

Актуальність теми. Сучасна індустрія відеоігор є однією з найбільш динамічних та фінансово містких сфер цифрової економіки. Стабільне зростання ринку, особливо в сегменті багатокористувацьких онлайн-ігор, обумовлює високий попит на кваліфікованих фахівців, здатних створювати програмні продукти зі складною клієнт-серверною архітектурою, синхронізацією даних у реальному часі та інтуїтивно зрозумілим інтерфейсом. Водночас, для невеликих студій та індивідуальних розробників ключовим фактором успіху є можливість швидкого прототипування та використання доступних, але потужних інструментів. Поєднання ігрового рушія Unity 3D та хмарного мережевого рішення Photon Engine є одним із найбільш затребуваних і технічно виправданих стеків для вирішення таких завдань, що й обумовлює актуальність теми даної роботи.

Метою бакалаврської роботи є розробка функціонального прототипу багатокористувацької гри в жанрі survival-action із використанням ігрового рушія Unity 3D та інтеграцією мережевого фреймворку Photon Engine для забезпечення взаємодії гравців у реальному часі.

Для досягнення поставленої мети необхідно вирішити наступні **завдання:**

1.Провести аналіз сучасних інструментів розробки комп'ютерних ігор та виконати порівняльну характеристику ігрових рушіїв.

2.Дослідити архітектурні патерни багатокористувацьких ігор та виконати огляд мережевих рішень, сумісних з Unity 3D, обґрунтувавши вибір Photon Engine.

3.Визначити жанр, цільову платформу та ключові механіки гри, що розробляється.

4.Спроєктувати архітектуру клієнт-серверної взаємодії та реалізувати базову логіку гри (рух гравця, система підбору предметів) засобами мови програмування C#. Інтегрувати Photon Engine у проєкт Unity для забезпечення мережевих функцій: створення кімнат, підключення гравців та синхронізації ігрових

об'єктів.

5.Розробити користувацький інтерфейс (UI) для керування мережевим з'єднанням та відображення ігрової інформації.

6.Провести тестування реалізованого прототипу, оцінити його продуктивність та стабільність мережевої взаємодії, виявити та проаналізувати проблемні місця.

Об'єктом дослідження є процес розробки програмного забезпечення для багатокористувацьких ігор із використанням сучасних ігрових рушіїв та мережевих технологій.

Предметом дослідження виступають технології, принципи та інструментальні засоби реалізації клієнт-серверної взаємодії, ігрової логіки та користувацького інтерфейсу в середовищі Unity 3D з використанням мережевого фреймворку Photon Engine.

Для вирішення поставлених завдань у роботі використано комплекс загальнонаукових та спеціальних методів: метод аналізу та синтезу (для огляду літературних джерел та аналізу існуючих рішень), порівняльний метод (для порівняння ігрових рушіїв та мережевих фреймворків), метод проєктування (для розробки архітектури гри), метод моделювання (для створення алгоритмів роботи програми), а також емпіричні методи (тестування, спостереження, аналіз помилок) для оцінки роботи розробленого прототипу.

Розроблений у рамках роботи прототип багатокористувацької гри може бути використаний як основа для подальшої комерційної розробки повноцінного ігрового продукту. Крім того, результати аналізу продуктивності та виявлені типові помилки становлять практичну цінність для оптимізації мережевої частини ігор.

Бакалаврська робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. Робота містить 20 рисунків, 4 таблиці, 1 додаток. Список використаних джерел налічує 35 найменуваннями.

РОЗДІЛ 1. АНАЛІЗ ІНСТРУМЕНТІВ ТА ОБҐРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ

1.1 Огляд сучасних інструментів для розробки комп’ютерних ігор

Сучасна індустрія розробки комп'ютерних ігор є багатогранною сферою, яка потребує від фахівців володіння широким спектром інструментальних засобів. Ці засоби охоплюють не лише безпосередньо створення ігрової логіки, але й моделювання графіки, написання коду, управління версіями, тестування та публікацію готового продукту. Аналіз інструментарію є критично важливим етапом на початку будь-якого ігрового проєкту, оскільки правильний вибір технологій безпосередньо впливає на швидкість розробки, якість кінцевого продукту та можливість його масштабування. Загалом, сукупність інструментів для розробки ігор можна класифікувати на кілька основних категорій: ігрові рушії, середовища розробки, графічні редактори, системи контролю версій та спеціалізовані мережеві рішення [1].

1. Ігрові рушії є фундаментом будь-якого проєкту. Вони являють собою інтегровані середовища, що надають розробнику набір готових компонентів для рендерингу графіки, обробки фізичних взаємодій, анімації, роботи зі звуком та скриптування ігрової логіки. Використання рушія дозволяє уникнути створення низькорівневих систем з нуля, зосередившись на унікальних механіках гри. Найбільш поширеними на сьогодні є такі рушії, як Unity, Unreal Engine, Godot та GameMaker Studio. Кожен з них має власну архітектуру, мову програмування та цільову аудиторію. Наприклад, Unity відзначається гнучкістю та підтримкою широкого спектру платформ, Unreal Engine орієнтований на створення високобюджетних 3D-проєктів з фотореалістичною графікою, а Godot є популярним вибором серед прихильників open-source рішень.

2. Середовища розробки (IDE) є невід'ємною частиною роботи програміста, особливо при створенні складної ігрової логіки. Вони забезпечують зручний

інтерфейс для написання, редагування, налагодження та оптимізації коду. У контексті розробки ігор на Unity, який використовує мову C#, найпоширенішими IDE є Microsoft Visual Studio та JetBrains Rider. Visual Studio пропонує глибоку інтеграцію з Unity через спеціальний пакет, надаючи розширені можливості для налагодження скриптів безпосередньо під час ігрового процесу, а також інструменти для профілювання продуктивності. Rider, у свою чергу, відомий своєю високою швидкодією, потужною системою аналізу коду (інспекції, рефакторинги) та інтелектуальними підказками, що значно прискорює написання якісного та підтримуваного коду.

3. Графічні редактори використовуються для створення візуальних активів гри: 2D-спрайтів, 3D-моделей, текстур, анімацій та матеріалів. Для 3D-моделювання де-факто стандартом у інді-сегменті є Blender – потужний інструмент з відкритим кодом, який підтримує повний цикл створення 3D-контенту: моделювання, скульптинг, текстурування, ригінг, анімацію та навіть рендеринг. Перевагами Blender є його безкоштовність, активна спільнота та постійний розвиток. Для створення 2D-графіки часто використовуються растрові редактори (наприклад, Adobe Photoshop, GIMP) для створення спрайтів та фонів, а також векторні редактори (наприклад, Adobe Illustrator, Inkscape) для розробки UI-елементів та логотипів, які легко масштабувати.

4. Системи контролю версій є обов'язковим інструментом для будь-якого сучасного командного (а часто – і індивідуального) проєкту. Вони дозволяють відстежувати всі зміни в коді, графічних файлах та конфігураціях сцен, забезпечують можливість відкату до попередніх стабільних версій, а також спрощують роботу кількох розробників над одним проєктом. Найбільш популярною системою є Git, яка у поєднанні з хмарними хостингами на кшталт GitHub, GitLab або Bitbucket, надає потужний і гнучкий механізм для управління репозиторієм, ведення історії змін та організації процесу code review [20]. Навіть при індивідуальному прототипуванні, використання Git є хорошою практикою, оскільки воно дисциплінує розробника та захищає від втрати результатів роботи.

5. Спеціалізовані інструменти для створення багатокористувацьких ігор заслуговують на окрему увагу. До них належать мережеві фреймворки та бібліотеки, які реалізують клієнт-серверну взаємодію. Серед найбільш відомих рішень, інтегрованих з Unity, варто виділити Photon Engine, Mirror, Netcode for GameObjects та Nakama. Ці інструменти надають готові API для створення сесій (кімнат), синхронізації стану об'єктів, виклику віддалених процедур, а також часто включають серверну логіку та системи автентифікації.

Таким чином, сучасний розробник комп'ютерних ігор має у своєму розпорядженні широкий та потужний арсенал інструментів. Ключовим завданням на початковому етапі проєкту є не просто знання про існування цих інструментів, а здатність зробити зважений вибір, враховуючи специфіку майбутньої гри, вимоги до продуктивності, цільову платформу, бюджет та досвід команди [22]. Наступні підрозділи роботи присвячені детальному порівняльному аналізу саме тих категорій інструментів, які є визначальними для розробки багатокористувацького проєкту: ігрових рушіїв та мережевих рішень.

1.2 Порівняльний аналіз сучасних ігрових рушіїв

Вибір ігрового рушія є одним із найбільш відповідальних рішень на початку розробки будь-якого ігрового проєкту. Ігровий рушій визначає не лише візуальну якість кінцевого продукту, але й продуктивність, підтримувані платформи, мову програмування, доступність документації та спільноти, а також загальну складність та швидкість розробки. Для обґрунтованого вибору технологічного стеку в межах даної роботи необхідно провести порівняльний аналіз найпопулярніших ігрових рушіїв, які використовуються в сучасній індустрії. Основними критеріями для порівняння обрано: мова програмування (впливає на поріг входження та доступність розробників), якість підтримки 2D та 3D-графіки (відповідає жанровій спрямованості проєкту), продуктивність (важлива для багатокористувацьких ігор із великою кількістю об'єктів), підтримувані платформи

(визначає потенційну аудиторію), а також якість документації та розмір спільноти (впливають на швидкість вирішення проблем). Розглянемо детально кожен із провідних рушіїв.

Unity є одним із найпоширеніших ігрових рушіїв у світі, особливо популярним серед інді-розробників та студій, що створюють мобільні ігри. Основними мовами програмування є C#, а також можливість використання JavaScript (UnityScript, нині застарілий) та Visual Scripting для візуального програмування. Unity демонструє високу якість як для 2D, так і для 3D-проєктів завдяки гнучкій системі рендерингу (Built-in, URP, HDRP). Продуктивність рушія є високою, особливо після оптимізації, а підтримка платформ охоплює всі сучасні операційні системи: Windows, macOS, Linux, Android, iOS, WebGL, а також консолі PlayStation, Xbox та Nintendo Switch [2]. Документація Unity є однією з найкращих у галузі: вона містить величезну кількість офіційних туторіалів, прикладів, API-довідників та форумів, а спільнота налічує мільйони розробників, що забезпечує швидке отримання відповідей на будь-які питання. Інтерфейс представлено на рисунку 1.1.

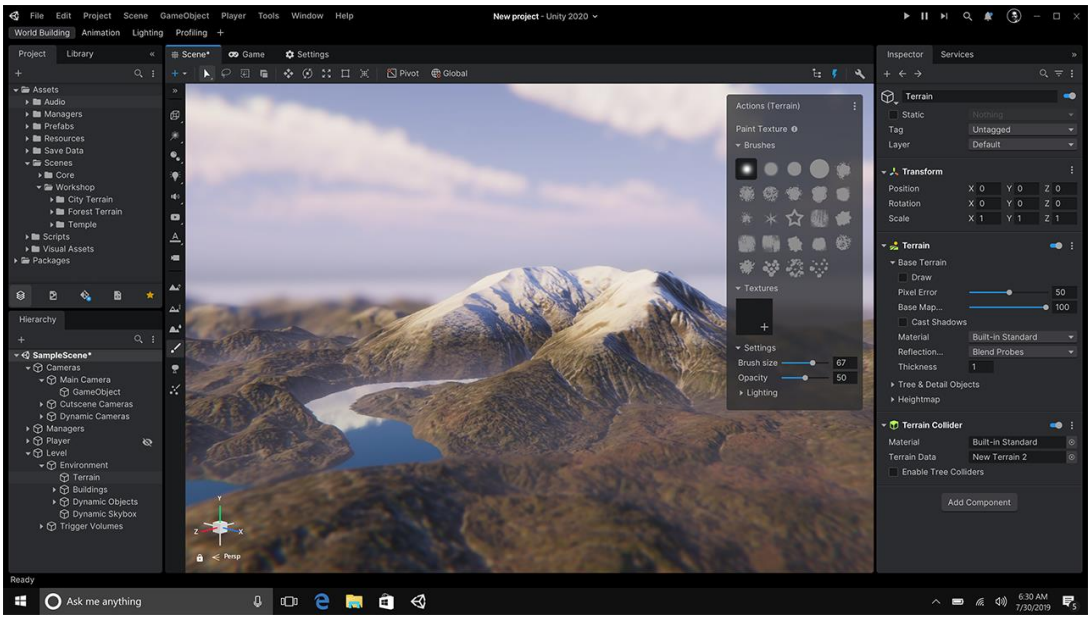

Рисунок 1.1 – Приклад інтерфейсу Unity

Unreal Engine розроблений компанією Epic Games і є стандартом для створення високобюджетних AAA-проєктів з фотореалістичною графікою. Основними мовами програмування є C++ та власна система візуального скриптингу Blueprints, яка дозволяє створювати ігрову логіку без написання коду. Unreal

,Engine демонструє дуже високу якість 3D-графіки, включаючи підтримку технологій Lumen (глобальне динамічне освітлення) та Nanite (віртуалізована геометрія високої деталізації), однак підтримка 2D-ігор є менш розвиненою порівняно з Unity. Продуктивність рушія є надзвичайно високою для 3D-проєктів, але вимоги до апаратного забезпечення є значно вищими. Підтримувані платформи включають ПК, консолі, мобільні пристрої, але підтримка WebGL є обмеженою [3]. Документація Unreal Engine є детальною, але через складність рушія поріг входження є вищим, ніж у Unity. Деталі роботи з Unreal відображені на рисунку 1.2.

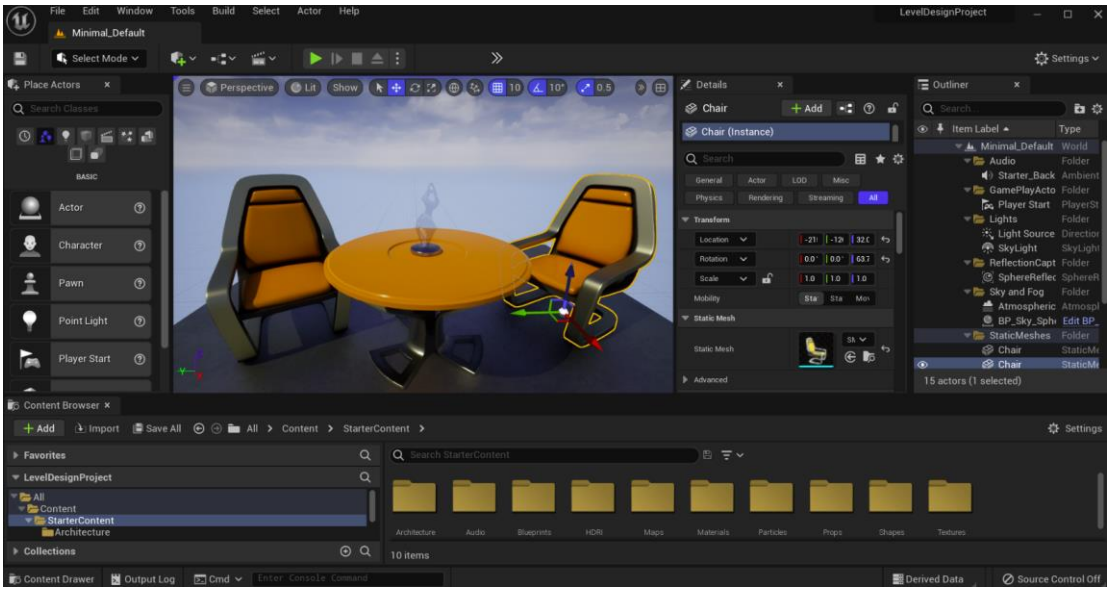

Рисунок 1.2 – Редактор Unreal Engine

Godot є рушієм з відкритим кодом, який набирає популярності завдяки своїй легкості, гнучкості та відсутності ліцензійних відрахувань. Основними мовами програмування є власна мова GDScript (синтаксично подібна до Python), а також підтримка C#, C++ та Visual Scripting. Godot демонструє добру якість як

для 2D, так і для 3D-графіки, хоча у 3D-сцені він все ще поступається Unity та Unreal Engine. Продуктивність рушія є середньою, проте він є дуже легким і швидко запускається. Підтримувані платформи включають ПК, мобільні пристрої та Web, але підтримка консолей є обмеженою та потребує окремих рішень. Документація Godot є доброю, але спільнота все ще значно менша, ніж у Unity та Unreal Engine. Приклад роботи з Godot зображено на рисунку 1.3.

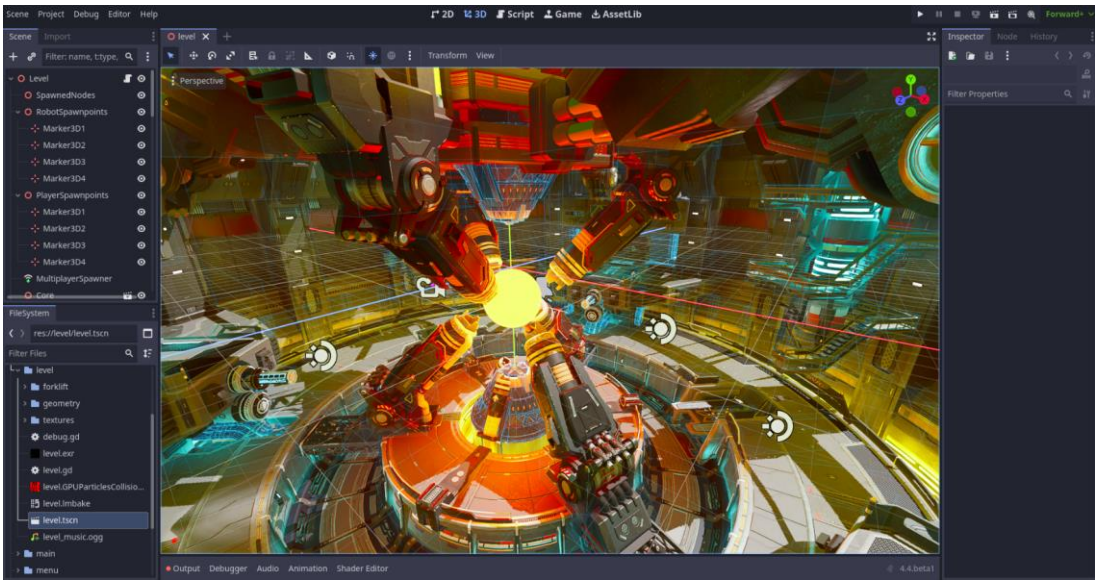

Рисунок 1.3 – Побудува ігрового рівня в Godot

GameMaker Studio є спеціалізованим інструментом для створення 2D-ігор, який здобув популярність завдяки таким хітам, як Undertale та Hotline Miami. Основним мовою програмування є власна мова GML (GameMaker Language), яка є відносно простою для вивчення [24. GameMaker демонструє високу якість саме для 2D-графіки та анімації, тоді як 3D-підтримка практично відсутня або є вкрай обмеженою. Продуктивність для 2D-ігор є дуже високою. Підтримувані платформи включають ПК, мобільні пристрої та Web. Документація є доброю, але рушій є пропрієтарним та платним (крім базової версії), що може бути обмеженням для деяких розробників. UI цього рушія відображено на рисунку 1.4.

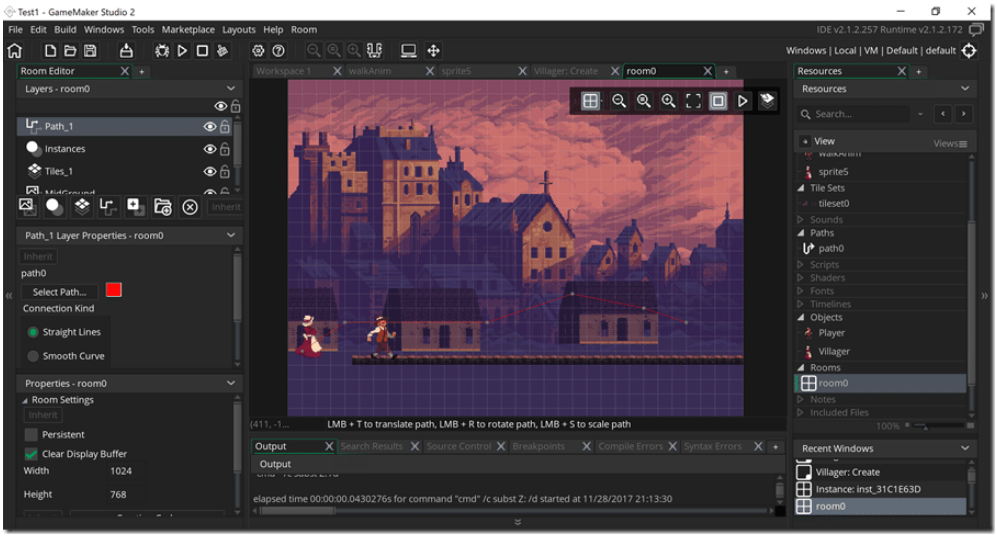

Рисунок 1.4 – Інтерфейс GameMaker Studio 2

Для наочного порівняння основних характеристик розглянутих ігрових рушіїв доцільно звести їх у зведену таблицю. Таблиця 1.1 відображає ключові параметри кожного рушія за визначеними критеріями.

Таблиця 1.1

Порівняльна характеристика сучасних ігрових рушіїв

Рушій	Мова програмування	Якість 2D/3D	Продуктивність	Платформи	Документація
Unity	C#	Висока (2D/3D)	Висока	Всі основні (ПК, моб., Web, консолі)	Обширна
Unreal Engine	C++, Blueprints	Дуже висока (3D)	Дуже висока	ПК, консолі, моб.	Детальна
Godot	GDScript, C#	Добра (2D/3D)	Середня	ПК, моб., Web	Добра
GameMaker Studio	GML	Висока (2D)	Висока (2D)	ПК, моб., Web	Добра

Як видно з таблиці 1.1, кожен із розглянутих рушіїв має свої сильні сторони та обмеження. Unreal Engine є беззаперечним лідером у сфері високоякісної 3D-графіки, але його складність може бути надмірною для невеликих проєктів. Godot є привабливим вибором завдяки відкритому коду, однак його 3D-

можливості та спільнота поки що поступаються лідерам. GameMaker Studio є чудовим інструментом для 2D-ігор, але зовсім не підходить для 3D-розробки. Для цієї роботи, метою якої є розробка багатокористувацького прототипу з використанням 3D-середовища, необхідним є баланс між якістю графіки, гнучкістю мережевої інтеграції, доступністю навчальних матеріалів та швидкістю розробки. З огляду на це, Unity є найбільш збалансованим вибором. Він забезпечує високу якість 3D-графіки, має найбільшу спільноту та документацію, що критично важливо при вирішенні специфічних завдань мережевої синхронізації, а також надає гнучкі засоби для інтеграції сторонніх рішень, таких як Photon Engine.

1.3 Аналіз архітектур багатокористувацьких ігор

Створення багатокористувацької гри є значно складнішим завданням порівняно з розробкою одноосібної (синглплеєрної) гри. Основна складність полягає в необхідності забезпечення узгодженого стану ігрового світу на кількох клієнтських пристроях одночасно, подоланні латентності мережі, обробці втрати пакетів даних та забезпеченні захисту від несанкціонованого втручання. Вибір архітектури багатокористувацької гри є фундаментальним рішенням, яке визначає продуктивність, масштабованість, безпеку та загальну складність реалізації проєкту [4]. У сучасній практиці розробки ігор склалися кілька основних архітектурних моделей, кожна з яких має свої переваги та недоліки.

Найбільш поширеною та надійною архітектурою для багатокористувацьких ігор є модель клієнт-сервер. У цій моделі один з вузлів мережі виконує роль центрального сервера, який володіє авторитетною копією ігрового світу. Усі клієнти надсилають на сервер свої дії, а сервер, після валідації цих дій та виконання необхідних розрахунків, розсилає всім клієнтам оновлений стан гри. Такий підхід має кілька ключових переваг. По-перше, він забезпечує високий рівень безпеки, оскільки клієнти не можуть безпосередньо впливати на стан гри – усі зміни проходять через серверну валідацію. По-друге, сервер гарантує синхронізацію

між усіма клієнтами, оскільки він є єдиним джерелом істини. По-третє, така архітектура добре масштабується: сервер може обслуговувати велику кількість одночасних гравців. Основним недоліком є необхідність утримання або оренди серверної інфраструктури, що пов'язано з додатковими витратами. Для цієї роботи, з огляду на використання хмарного сервісу Photon Engine, буде застосовано саме цю модель. Схема цієї архітектури зображена на рисунку 1.5.

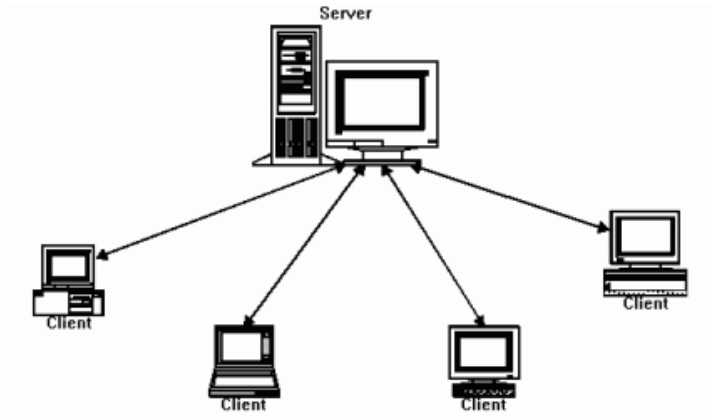

Рисунок 1.5 – Загальна схема клієнт-серверної архітектури

Іншою поширеною моделлю є архітектура однорангової мережі (Peer-to-Peer, P2P). У цій моделі всі гравці є рівноправними вузлами, і стан гри синхронізується безпосередньо між клієнтами. Для координації з'єднань часто використовується виділений сервер для початкового рукостискання (handshake), але після встановлення з'єднань дані передаються напряму. Перевагами P2P є відсутність витрат на сервери та простота реалізації для ігор з невеликою кількістю гравців [5]. Однак ця модель має суттєві недоліки: вона є вразливою до шахрайства (оскільки жоден вузол не має авторитету, будь-який гравець теоретично може підробити дані), а також виникають проблеми з синхронізацією при нерівномірній швидкості з'єднання учасників. Крім того, підтримка гравців з обмеженими можливостями NAT є складною. P2P архітектура рідко використовується в сучасних серйозних багатокористувацьких проєктах. Ця архітектура представлена на рисунку 1.6.

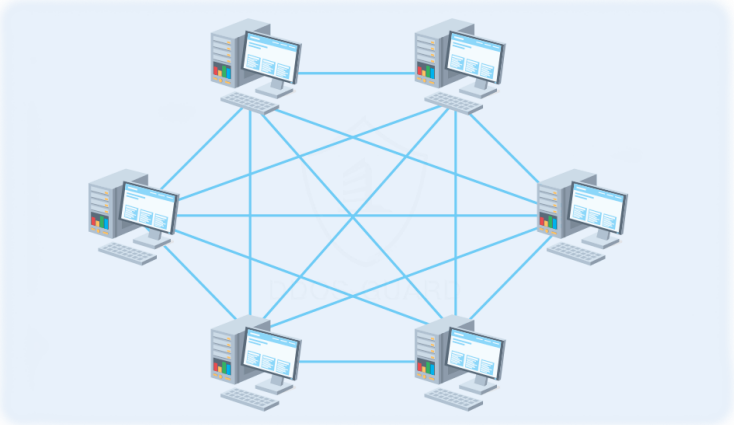

Рисунок 1.6 – Приклад піринговой мережі

Проміжним варіантом є архітектура з виділеним хостом. У цій моделі один із гравців виконує функції сервера, одночасно залишаючись гравцем. Інші гравці підключаються до нього. Перевагою є відсутність витрат на оренду окремого сервера. Недоліки є суттєвими: гравець-хост має перевагу через нульову затримку до власного сервера, а у разі виходу хоста гра може перерватися. Крім того, якість з'єднання залежить від домашнього інтернету хоста, а захист від шахрайства є обмеженим [6]. Ця модель часто використовується в кооперативних іграх або іграх для невеликих груп друзів, але не підходить для конкурентного мультиплеєра з великою кількістю гравців. Ця схема показана на рисунку 1.7.

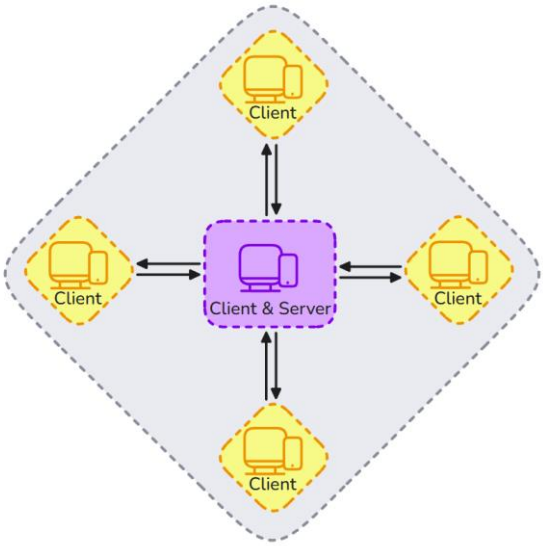

Рисунок 1.7 – Схема архітектури з виділеним сервером

Крім вибору моделі взаємодії, важливим аспектом є спосіб синхронізації даних між сервером та клієнтами. Існує два основних підходи. Перший – синхронізація стану, при якій сервер регулярно (наприклад, 10-30 разів на секунду) надсилає всім клієнтам повний або частковий стан гри. Клієнти отримують цей стан та інтерполюють позиції об'єктів між отриманими кадрами. Цей підхід є простим у реалізації, але створює високе мережеве навантаження. Другий підхід – синхронізація подій, при якому клієнти надсилають події (наприклад, «гравець А вистрелив»), а сервер розраховує наслідки та розсилає результати. Це набагато ефективніше для рідкісних подій, але потребує ретельного проєктування системи передбачення для компенсації затримок.

1.4 Огляд мережевих рішень для Unity 3D

Створення багатокористувацької гри в середовищі Unity неможливе без використання спеціалізованого мережевого рішення, оскільки базовий функціонал рушія не містить вбудованих засобів для синхронізації стану між клієнтами, управління сесіями та передачі даних у реальному часі. На сьогоднішній день існує декілька популярних фреймворків та бібліотек, які забезпечують мережеву взаємодію в Unity.

Photon Engine є одним із найбільш популярних та зручних мережевих рішень для Unity [7]. Розроблений компанією Exit Games, Photon пропонує кілька варіантів інтеграції: PUN (Photon Unity Networking) та Fusion (більш сучасне рішення з підтримкою різних топологій). Photon PUN 2, який використовується в цій роботі, є високорівневим API, що значно спрощує створення багатокористувацьких ігор. Ключовими перевагами Photon є: наявність безкоштовного тарифу з обмеженням до 20 одночасних гравців (що достатньо для прототипування); хмарна інфраструктура з серверами по всьому світу (низька затримка); проста інтеграція через пакет з Asset Store; обширна документація та велика спільнота; підтримка кімнат (rooms), RPC-викликів, синхронізації об'єктів через

PhotonView та PhotonTransformView. Основним недоліком є необхідність оплати при перевищенні безкоштовного ліміту, а також обмежені можливості для створення власної серверної логіки (хоча є Photon Server SDK для просунутих користувачів). Photon підтримує архітектуру клієнт-сервер, де роль сервера виконує хмарна інфраструктура Photon.

Mirror є популярним рішенням з відкритим кодом, яке є форком застарілого UNET (Unity Networking). Mirror надає низькорівневий контроль над мережевою взаємодією та підтримує різні топології: клієнт-сервер на власному сервері, а також Listen Server (гравець-хост). Перевагами Mirror є: безкоштовність (open-source), відсутність зовнішніх залежностей від хмарних сервісів, повний контроль над серверною логікою, гнучкість налаштувань. Недоліками є: вища складність налаштування порівняно з Photon (необхідність самостійної організації серверної інфраструктури, виділеного сервера або підтримки NAT), менша кількість готових прикладів та документації, а також відсутність вбудованої хмарної інфраструктури (розробник самостійно відповідає за хостинг та масштабування серверів). Mirror підходить для проєктів, де важливий повний контроль над архітектурою та є можливість оренди або утримання власних серверів [8].

Netcode for GameObjects (раніше відомий як MLAPI) є офіційним мережевим рішенням від Unity Technologies. Воно активно розвивається та інтегрується з іншими системами Unity [8]. Netcode підтримує топології клієнт-сервер та Listen Server, надає засоби для синхронізації змінних, RPC-викликів, а також мережеві об'єкти. Перевагами є: офіційна підтримка від Unity (що гарантує довгострокову сумісність), безкоштовність, активний розвиток (регулярні оновлення), інтеграція з Unity Transport (низькорівневий мережевий шар). Недоліками є: відносна новизна (менше прикладів та документації порівняно з Photon), вимогливість до ресурсів, а також необхідність самостійної організації серверної інфраструктури. Крім того, стабільна версія Netcode for GameObjects з'явилася відносно нещодавно, тому деякі розробники побоюються використовувати її у великих проєктах через потенційні приховані помилки.

Nakama є більш комплексним серверним рішенням, яке включає не лише мережеві функції (реальний час), але й автентифікацію, збереження даних гравців, соціальні функції (друзі, групи), реферальні системи та навіть елементи економіки гри. Nakama підтримує як власний хостинг, так і хмарний (через Heroic Labs). Перевагами є: величезний набір вбудованих функцій, підтримка кількох мов програмування (Go, JavaScript, Lua) для серверної логіки, відкритий код базової версії, масштабованість [9]. Недоліками є: значна складність налаштування для початківців, надмірність для невеликих проєктів (якщо потрібен лише базовий мультиплеєр), вищі вимоги до ресурсів сервера.

Для наочного порівняння основних характеристик розглянутих мережевих рішень доцільно звести їх у зведену таблицю. Таблиця 1.2 відображає ключові параметри кожного рішення за такими критеріями: сумісність з Unity, тип хостингу, тип ліцензії та рівень інтеграції.

Таблиця 1.2

Порівняльна характеристика мережевих рішень для Unity

Мережеві рішення	Сумісність з Unity	Хостинг	Ліцензія	Інтеграція
Photon	Так	Хмарний	Комерційна	Висока
Mirror	Так	Власний	Open Source	Висока
Netcode	Так	Власний	Open Source	Середня
Nakama	Так	Обидва	Open Source	Середня

Як видно з таблиці 1.2, кожне з розглянутих рішень має свою нішу та цільову аудиторію. Photon PUN 2 є найкращим вибором для швидкого прототипування та ігор з невеликою кількістю гравців завдяки безкоштовному тарифу, простоті інтеграції та готовій хмарній інфраструктурі. Mirror та Netcode for GameObjects підходять для проєктів, де потрібен повний контроль над серверною частиною та є можливість самостійного хостингу. Nakama є надмірним для

невеликих прототипів, але незамінним для великих соціально-орієнтованих ігор зі складною економікою та системою досягнень [10].

1.5 Обґрунтування вибору Unity 3D та Photon Engine

На основі проведеного у попередніх підрозділах аналізу сучасних ігрових рушіїв та мережевих рішень, необхідно сформулювати обґрунтоване технічне рішення щодо вибору технологічного стеку для реалізації прототипу багатокористувацької гри.

Обґрунтування вибору Unity 3D як основного ігрового рушія базується на декількох факторах. По-перше, мова програмування C#, яка використовується в Unity, є сучасною об'єктно-орієнтованою мовою з високим рівнем типізації, потужними засобами асинхронного програмування (async/await) та великою кількістю бібліотек. Це дозволяє писати структурований, підтримуваний та масштабований код, що критично важливо для розробки ігрової логіки середньої складності, яка включає мережеву взаємодію, обробку введення, фізику та систему підбору предметів. По-друге, Unity забезпечує високу якість як для 2D, так і для 3D-графіки. Оскільки розроблюваний прототип належить до жанру survival-action з використанням тривимірного середовища, Unity пропонує гнучку систему рендерингу через конвеєри Built-in, URP (Universal Render Pipeline) або HDRP (High Definition Render Pipeline). URP є оптимальним вибором для цього проєкту, оскільки він забезпечує добру продуктивність на широкому спектрі пристроїв та підтримує сучасні візуальні ефекти. По-третє, Unity підтримує практично всі сучасні платформи: Windows, macOS, Linux, Android, iOS, WebGL та консолі. Це дозволяє в майбутньому портувати гру на інші платформи без кардинальної зміни кодової бази, що розширює потенційну аудиторію та підвищує комерційну цінність продукту. По-четверте, документація Unity та розмір спільноти є одними з найкращих у галузі. Величезна кількість офіційних туторіалів, прикладів, форумів та відеоматеріалів дозволяє швидко знаходити рішення для

типових проблем, включаючи специфічні питання мережевої синхронізації. Це значно скорочує час розробки та налагодження, що особливо важливо в умовах виконання бакалаврської роботи з обмеженими часовими рамками [11].

Обґрунтування вибору Photon Engine (PUN 2) як мережевого рішення також базується на кількох ключових факторах. По-перше, Photon PUN 2 пропонує безкоштовний тариф з обмеженням до 20 одночасних гравців, що є достатнім для створення прототипу, тестування та навіть початкової публікації гри. Відсутність початкових витрат дозволяє зосередитись на розробці без фінансового тиску. По-друге, Photon надає готову хмарну інфраструктуру з серверами по всьому світу (США, Європа, Азія). Це автоматично вирішує проблеми з низькою затримкою (ping), NAT traversal (проходження через мережеві екрани) та масштабуванням. Не потрібно орендувати або налаштовувати власні сервери, що значно спрощує архітектуру проєкту та зменшує час розробки. По-третє, Photon PUN 2 має високий рівень інтеграції з Unity. Інтеграція виконується через стандартний пакет з Asset Store, а API є інтуїтивно зрозумілим та добре документованим. По-четверте, Photon PUN 2 підтримує всі основні платформи, які підтримує Unity, включаючи ПК, мобільні пристрої, WebGL та консолі. Це забезпечує універсальність рішення та можливість портування в майбутньому [12].

Таким чином, поєднання Unity 3D та Photon Engine (PUN 2) є технічно обґрунтованим та практично виправданим вибором для реалізації поставленої задачі. Unity надає потужне середовище для створення ігрової логіки, графіки та інтерфейсу, а Photon забезпечує надійну, просту у використанні та масштабовану мережеву взаємодію.

Постановка задачі розробки багатокористувацької гри

На основі проведеного аналізу інструментів, архітектур та мережевих рішень, а також з урахуванням обраного технологічного стеку (Unity 3D та Photon Engine PUN 2), необхідно сформулювати конкретні задачі, які мають бути вирішені в процесі розробки прототипу багатокористувацької гри.

Метою роботи є створення функціонального прототипу, який демонструє

базові механіки багатокористувацької взаємодії в реальному часі. Відповідно до цієї мети, визначено наступні задачі розробки:

1. Першою задачею є реалізація мережевого з'єднання та системи кімнат. Для функціонування багатокористувацької гри необхідно забезпечити можливість підключення клієнтів до хмарного сервера Photon, створення нових ігрових кімнат з унікальними назвами, приєднання до існуючих кімнат, а також автоматичне приєднання до випадкової доступної кімнати.

2. Другою задачею є створення та синхронізація гравців у ігровій сцені. Після успішного приєднання до кімнати необхідно створити об'єкт гравця в сцені для кожного клієнта, причому кожен гравець повинен з'являтися у визначеній spawn-точці (залежно від його порядкового номера в кімнаті)[26].

3. Третьою задачею є реалізація керування гравцем. Кожен гравець повинен мати можливість керувати своїм персонажем: рухатися вперед/назад/вправо/вліво, стрибати, а також повертати камеру за допомогою миші.

4. Четвертою задачею є реалізація системи підбору предметів. На ігровій сцені повинні бути розміщені спеціальні об'єкти, які гравець може підібрати, наблизившись до них та натиснувши відповідну клавішу.

5. П'ятою задачею є розробка інтерфейсу користувача. Необхідно створити UI-елементи для головного меню, а також для ігрового інтерфейсу (HUD).

6. Шостою задачею є реалізація ігрового таймера та завершення гри. Гра повинна мати обмежену тривалість (наприклад, 120 секунд). Після завершення таймера гра має зупинитися, а гравці отримати відповідне сповіщення.

7. Сьомою задачею є відображення нікнеймів гравців над їхніми персонажами.

Таким чином, сформульовані задачі повністю відповідають меті роботи – створенню функціонального прототипу багатокористувацької гри з базовими механіками. Вирішення цих задач дозволить отримати працюючий ігровий продукт, який може бути використаний для подальшого тестування, аналізу продуктивності та як основа для розширення функціоналу в майбутньому. Детальне

проєктування та реалізація цих задач будуть викладені в Розділі 2 даної роботи.

1.6 Висновки до розділу

У першому розділі роботи було проведено комплексне дослідження сучасних інструментів, архітектур та технологій для розробки багатокористувацьких ігор.

Аналіз інструментів показав, що створення ігрового продукту потребує використання широкого спектру засобів: ігрових рушіїв, середовищ розробки, графічних редакторів, систем контролю версій та мережевих рішень. Найбільш важливими категоріями є ігрові рушії та мережеві рішення, які визначають базову архітектуру проєкту. Порівняльний аналіз ігрових рушіїв (Unity, Unreal Engine, Godot, GameMaker Studio) за критеріями мови програмування, якості графіки, продуктивності, підтримки платформ та документації показав, що Unity пропонує найкращий баланс між цими параметрами. Аналіз архітектур багатокористувацьких ігор визначив клієнт-серверну модель як найбільш надійну та безпечну. Вона забезпечує централізований контроль ігрового стану та захист від шахрайства, що є критичним для онлайн-ігор. Огляд мережевих рішень дозволив обрати Photon PUN 2. Це рішення є єдиним, що поєднує високу інтеграцію з Unity, готову хмарну інфраструктуру (відсутність потреби у власних серверах) та безкоштовний тариф до 20 одночасних гравців. На основі проведеного аналізу обґрунтовано вибір технологічного стеку: ігровий рушій Unity 3D та мережевий фреймворк Photon Engine (PUN 2). Цей стек є оптимальним для створення прототипу багатокористувацької гри в жанрі survival-action. Сформульовано сім задач розробки. Таким чином, теоретичні дослідження першого розділу створили необхідне підґрунтя для практичної реалізації прототипу, яка буде представлена в другому розділі роботи.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОТОТИПУ ГРИ

2.1 Визначення жанру, платформи та цільової аудиторії гри

Перед початком безпосередньої реалізації програмного продукту необхідно провести аналіз ключових параметрів майбутньої гри: її жанрової приналежності, цільової платформи та портрета потенційного користувача. Цей аналіз дозволяє обґрунтувати архітектурні рішення та визначити пріоритетні напрямки розробки.

Жанр гри визначає її основні ігрові механіки, темп, атмосферу та очікування аудиторії [19]. Для цього проєкту було проведено аналіз популярності різних жанрів у сучасній ігровій індустрії з урахуванням специфіки багатокористувацької розробки. Основними критеріями вибору стали: динамічність геймплею (що дозволяє утримувати увагу гравців в онлайн-сесіях), потенціал для кооперативної або конкурентної взаємодії, а також технічна реалізованість у рамках бакалаврської роботи. На основі цих критеріїв обрано жанр survival-action (екшн з елементами виживання). Цей жанр характеризується необхідністю дослідження ігрового простору, взаємодією з об'єктами оточення (підбір предметів, збільшення рахунку) та обмеженим ігровим часом, що створює додаткову напругу. Обраний жанр добре суміщається з багатокористувацьким режимом, оскільки дозволяє гравцям змагатися за найбільшу кількість зібраних предметів або найшвидше проходження [13].

Цільова платформа визначає технічні обмеження та можливості гри: продуктивність, органи керування, роздільну здатність екрану, мережеві протоколи. Основними кандидатами для розробки були: персональні комп'ютери під управлінням Windows, мобільні пристрої (Android, iOS) та веб-платформа WebGL. Для цього проєкту обрано платформу Windows PC. Цей вибір обумовлений наступними факторами: простота налагодження мережевої взаємодії (можливість запуску кількох клієнтів на одній машині для тестування); більш потужне апаратне

забезпечення (дозволяє зосередитись на реалізації логіки, а не на оптимізації для слабких пристроїв); зручність керування з використанням клавіатури та миші (WASD для руху, миша для огляду, E для взаємодії), що є стандартом для жанру action; а також широка розповсюдженість платформи серед цільової аудиторії.

Цільовою аудиторією розроблюваної гри є гравці віком від 12 до 30 років, які цікавляться багатокористувацькими іграми у жанрі action, мають базові навички керування в 3D-просторі та шукають швидкі ігрові сесії тривалістю 2-5 хвилин (з урахуванням налаштованого таймера на 120 секунд). Гра орієнтована як на конкурентну взаємодію (змагання за найбільшу кількість очок), так і на кооперативне проходження (спільний збір предметів без прямої конкуренції). Соціально-демографічний портрет включає студентів та молодих фахівців, які мають доступ до персонального комп'ютера та стабільного інтернет-з'єднання [14].

Для наочного представлення функціональних вимог до системи та взаємодії користувача з нею розроблено діаграму варіантів використання на рисунку 2.1. Ця діаграма відображає актора (гравця) та всі основні дії, які він може виконувати в межах розроблюваного програмного продукту.

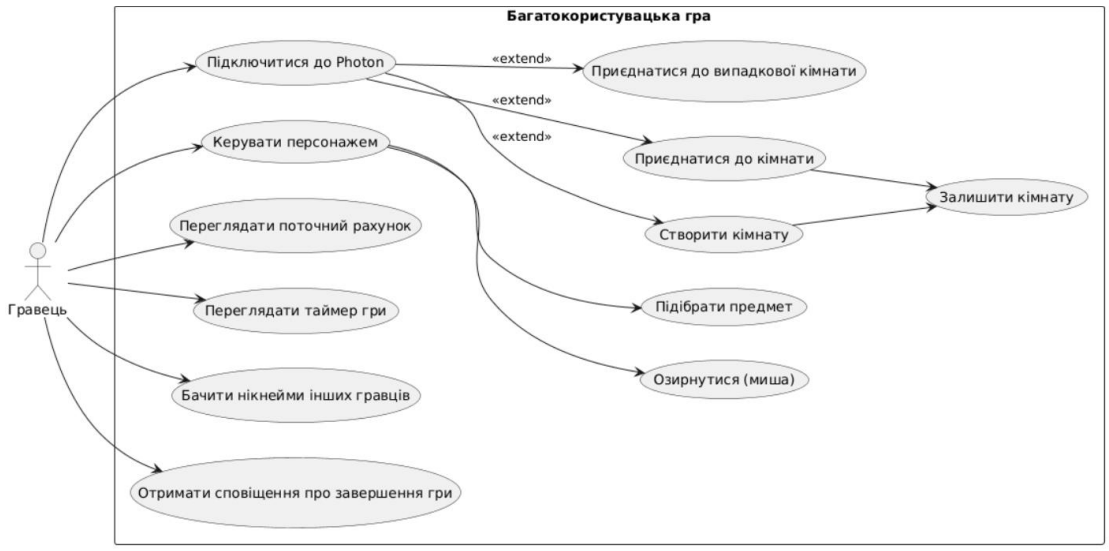

Рисунок 2.1 – Діаграма варіантів використання гри

Як видно з діаграми, основний актор «Гравець» має доступ до дванадцяти основних варіантів використання. Процес підключення до Photon є базовим, після якого можливе створення нової кімнати, приєднання до існуючої або приєднання до випадкової кімнати. Варіант використання «Залишити кімнату» є доступним після створення або приєднання до кімнати. Безпосередньо ігрова механіка включає керування персонажем з можливістю огляду мишею та підбору предметів. Інформаційні варіанти використання забезпечують зворотний зв'язок з гравцем.

2.2 Проєктування архітектури клієнт-серверної взаємодії

Архітектура клієнт-серверної взаємодії є ключовим елементом будь-якої багатокористувацької гри. Вона визначає, як саме відбувається обмін даними між гравцями, хто володіє авторитетною копією ігрового світу та як обробляються критичні події. У цьому підрозділі розглядається проєктування архітектури на основі обраного технологічного стеку (Unity 3D + Photon PUN 2) та реалізованого коду. Розроблювана гра використовує клієнт-серверну архітектуру з хмарним сервером Photon. У цій моделі всі клієнти (гравці) підключаються до центрального сервера, який виступає єдиним джерелом істини. Клієнти надсилають на сервер свої дії (команди руху, підбір предметів), а сервер після їх валідації розсилає оновлений стан усім учасникам сесії. Така архітектура забезпечує високий рівень синхронізації та захисту від шахрайства, оскільки клієнти не можуть безпосередньо змінювати стан гри. Клієнтська частина реалізована у вигляді застосунку на Unity, який виконує наступні функції: відображення графіки та інтерфейсу, обробка введення користувача, локальне передбачення руху (для плавності), а також отримання та відображення даних від сервера.

Для візуалізації структурних компонентів системи та зв'язків між ними розроблено діаграму класів (Class Diagram). Ця діаграма на рисунку 2.2 відображає всі основні класи проєкту, їхні поля, методи та відношення між

собою.

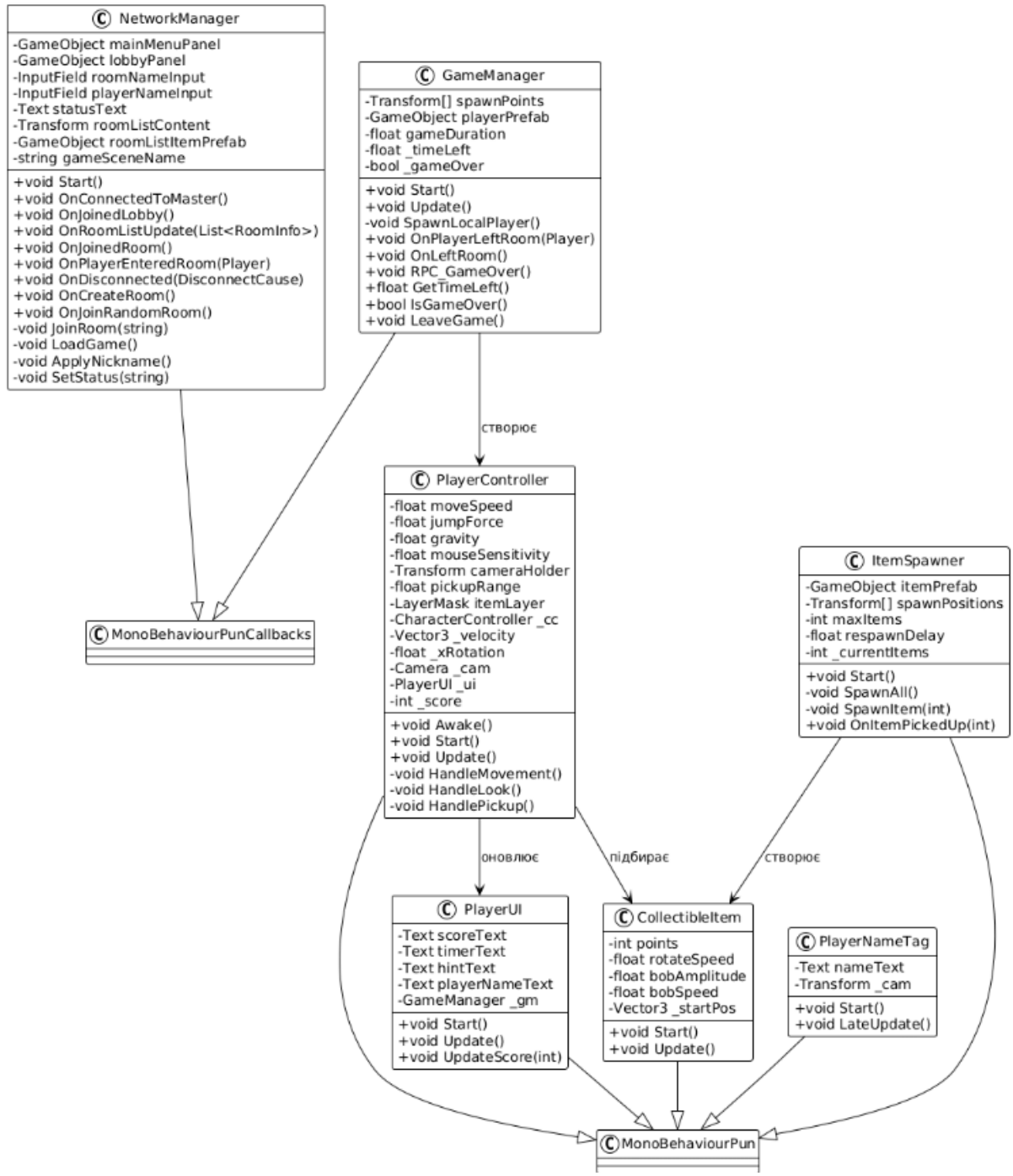

Рисунок 2.2 – Діаграма класів системи

Як видно з діаграми, система складається з семи основних класів, кожен з яких має чітко визначену зону відповідальності. Класи NetworkManager та GameManager успадковуються від MonoBehaviourPunCallbacks, що дозволяє їм

отримувати callback-події від Photon (підключення, вхід у кімнату, вихід гравця тощо). Класи PlayerController, PlayerUI, CollectibleItem, PlayerNameTag та ItemSpawner успадковуються від MonoBehaviourPun, що надає їм доступ до мережевих компонентів (photonView, photonView.IsMine). Зв'язки між класами відображають залежності: GameManager створює PlayerController через PhotonNetwork.Instantiate, PlayerController оновлює PlayerUI при підборі предмета та взаємодіє з CollectibleItem, а ItemSpawner відповідає за створення CollectibleItem на сцені.

Для візуалізації динаміки мережевої взаємодії на 2.3 розроблено діаграму послідовності (Sequence Diagram), яка відображає процес підключення нового гравця до ігрової сесії.

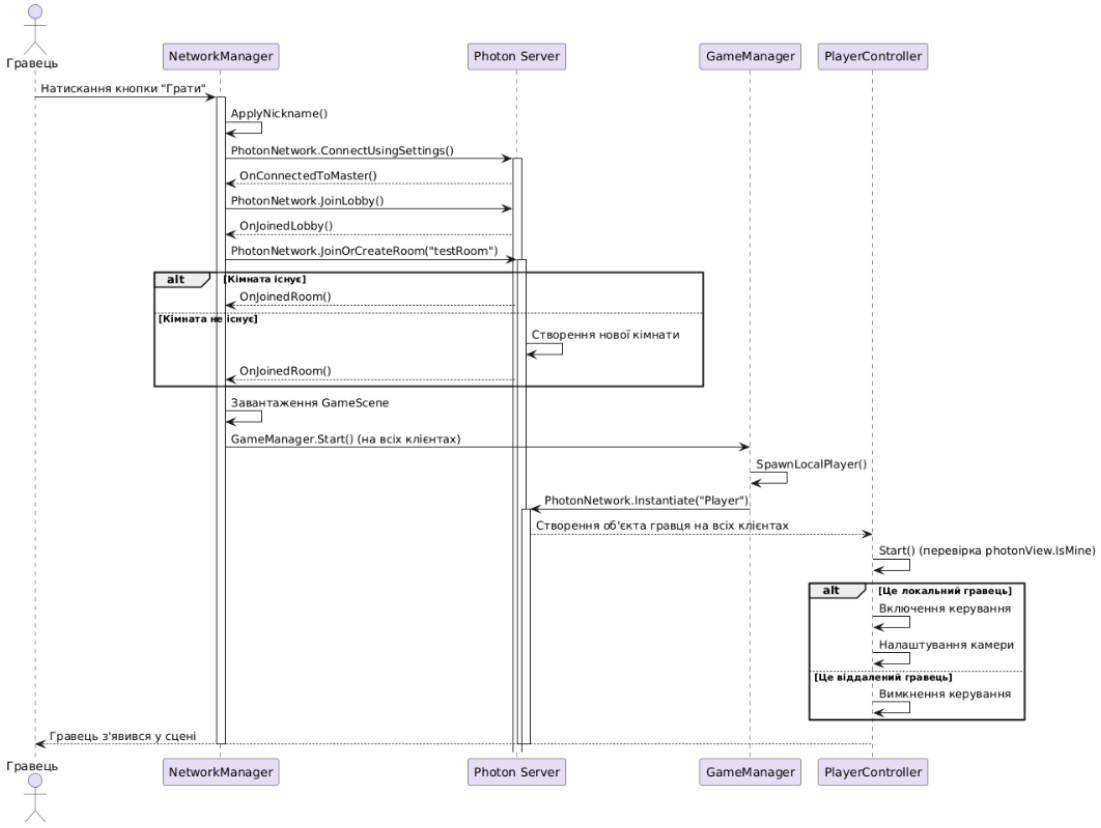

Рисунок 2.3 – Діаграма послідовності підключення гравця

Діаграма ілюструє повний цикл підключення гравця. Процес починається з натискання кнопки «Грати» в інтерфейсі, після чого NetworkManager встановлює ім'я гравця та ініціює підключення до Photon сервера. Після успішного

підключення та входу в лобі відбувається створення або приєднання до кімнати. Коли гравець опиняється в кімнаті, GameManager завантажує ігрову сцену та створює об'єкт гравця через PhotonNetwork.Instantiate. На етапі створення гравця відбувається перевірка photonView.IsMine: для локального гравця активується керування та налаштовується камера, для віддалених гравців керування вимикається. Цей підхід забезпечує коректну роботу мережевої взаємодії.

2.3 Реалізація базової логіки гри засобами C#

Базова логіка гри реалізована мовою C# в середовищі Unity. У цьому підрозділі розглядаються ключові компоненти, що забезпечують функціонування гравця, його взаємодію з оточенням та обробку ігрових подій. Кожен клас має чітко визначену зону відповідальності, що відповідає принципам об'єктно-орієнтованого програмування [28].

Контролер гравця (PlayerController). Клас PlayerController є центральним компонентом, що відповідає за керування персонажем, обробку введення з клавіатури та миші, а також взаємодію з предметами. Він успадковується від MonoBehaviourPun, що надає доступ до мережевих функцій Photon [14].

Клас містить наступні ключові поля для налаштування параметрів руху, камери та підбору предметів:

```
public float moveSpeed = 5f;
public float jumpForce = 5f;
public float gravity = -9.81f;
[Header("Камера")]
public float mouseSensitivity = 2f;
public Transform cameraHolder;
[Header("Підбір предметів")]
public float pickupRange = 2.5f;
public LayerMask itemLayer;
```

Метод Start виконує ініціалізацію, перевіряючи чи є поточний гравець локальним. Якщо гравець є віддаленим (photonView.IsMine == false), керування не активується:

```
void Start()
{
    if (!photonView.IsMine) return;
    _cam = Camera.main;
    if (_cam != null && cameraHolder != null)
        _cam.transform.SetParent(cameraHolder);
    Cursor.lockState = CursorLockMode.Locked;
    Cursor.visible = false;
}
```

Метод HandleMovement реалізує рух персонажа з використанням CharacterController, обробку стрибка та застосування гравітації:

```
void HandleMovement()
{
    bool grounded = _cc.isGrounded;
    if (grounded && _velocity.y < 0) _velocity.y = -2f;
    float h = Input.GetAxis("Horizontal");
    float v = Input.GetAxis("Vertical");
    Vector3 move = transform.right * h + transform.forward * v;
    _cc.Move(move * moveSpeed * Time.deltaTime);
    if (Input.GetButtonDown("Jump") && grounded)
        _velocity.y = Mathf.Sqrt(jumpForce * -2f * gravity);
    _velocity.y += gravity * Time.deltaTime;
    _cc.Move(_velocity * Time.deltaTime);
}
```

Метод HandleLook забезпечує поворот камери мишею по горизонталі (обертання персонажа) та вертикалі (обертання камери з обмеженням кута), що є стандартним для жанру action:

```
void HandleLook()
{
```

```
        float mx = Input.GetAxis("Mouse X") * mouseSensitivity;
        float my = Input.GetAxis("Mouse Y") * mouseSensitivity;
        _xRotation -= my;
        _xRotation = Mathf.Clamp(_xRotation, -80f, 80f);

        if (cameraHolder != null)
            cameraHolder.localRotation = Quaternion.Euler(_xRotation, 0f, 0f);
        transform.Rotate(Vector3.up * mx);
    }
```

Метод HandlePickup відповідає за підбір предметів. Він виконує Raycast від камери вперед та при виявленні об'єкта з компонентом CollectibleItem оновлює рахунок та знищує предмет через мережу:

```
    void HandlePickup()
    {
        if (!Input.GetKeyDown(KeyCode.E)) return;
        Ray ray = new Ray(cameraHolder.position, cameraHolder.forward);
        if (Physics.Raycast(ray, out RaycastHit hit, pickupRange, itemLayer))
        {
            var item = hit.collider.GetComponent<CollectibleItem>();
            if (item != null)
            {
                _score += item.points;
                _ui?.UpdateScore(_score);
                PhotonNetwork.Destroy(hit.collider.gameObject);
            }
        }
    }
```

Клас предмета для підбору (CollectibleItem). Клас CollectibleItem відповідає за візуальну поведінку предметів, які можна підібрати. Він забезпечує обертання предмета навколо вертикальної осі та плавне зависання (ефект «парячого» предмета), що робить його помітнішим для гравця [30]. Клас містить наступний код:

```
public class CollectibleItem : MonoBehaviourPun
{
    [Header("Параметри")]
    public int points = 10;
    public float rotateSpeed = 90f;
    public float bobAmplitude = 0.2f;
    public float bobSpeed = 1.5f;

    Vector3 _startPos;
    void Start()
    {
        _startPos = transform.position;
    }
    void Update()
    {
        transform.Rotate(Vector3.up, rotateSpeed * Time.deltaTime, Space.World);
        float newY = _startPos.y + Mathf.Sin(Time.time * bobSpeed) * bobAmplitude;
        transform.position = new Vector3(transform.position.x, newY, transform.position.z);
    }
}
```

Менеджер гри (GameManager). Клас GameManager координує загальний ігровий процес: спавн гравців, управління таймером та обробку завершення гри. Він успадковується від MonoBehaviourPunCallbacks для отримання мережевих подій. Поля класу визначають точки появи гравців, префаб гравця та тривалість гри:

```
public Transform[] spawnPoints;
public GameObject playerPrefab;
[Header("Гра")]
public float gameDuration = 120f;
float _timeLeft;
bool _gameOver;
```

Метод SpawnLocalPlayer визначає точку появи на основі номера гравця в кімнаті та створює об'єкт гравця через PhotonNetwork.Instantiate. Після створення камера приєднується до CameraHolder створеного гравця:

```
void SpawnLocalPlayer()
{
    int idx = PhotonNetwork.LocalPlayer.ActorNumber % spawnPoints.Length;
    Transform sp = spawnPoints[idx];
    GameObject player = PhotonNetwork.Instantiate(
        playerPrefab.name, sp.position, sp.rotation);

    var cam = Camera.main;
    if (cam != null)
    {
        var holder = player.transform.Find("CameraHolder");
        if (holder != null)
            cam.transform.SetParent(holder);
    }
}
```

Метод Update реалізує логіку таймера. Важливо, що оновлення часу виконується лише на Master Client (господарі кімнати), що запобігає конфліктам при одночасному оновленні таймера кількома клієнтами:

```
void Update()
{
    if (_gameOver || !PhotonNetwork.IsMasterClient) return;
    _timeLeft -= Time.deltaTime;
    if (_timeLeft <= 0)
    {
        _timeLeft = 0;
        photonView.RPC(nameof(RPC_GameOver), RpcTarget.All);
    }
}
```

Метод RPC_GameOver позначений атрибутом PunRPC, що дозволяє викликати його на всіх клієнтах одночасно. Це забезпечує синхронізоване завершення гри:

```
void RPC_GameOver()
{
   _gameOver = true;
   Debug.Log("Гра завершена! Час вийшов.");
}
```

А отже, реалізована на C# базова логіка гри демонструє наступні ключові аспекти: розмежування локального та віддалених гравців через перевірку photonView.IsMine; використання CharacterController для фізично коректного руху з урахуванням гравітації; застосування Raycast для точної взаємодії з предметами; централізоване управління ігровим часом лише на Master Client; синхронізацію важливих подій через RPC-виклики [15].

2.4 Інтеграція Photon Engine у Unity-проєкт

Для забезпечення багатокористувацької взаємодії в розроблюваній грі було виконано інтеграцію мережевого фреймворку Photon PUN 2 (Photon Unity Networking) у проєкт Unity. Процес інтеграції включає наступні етапи.

Першим етапом є встановлення Photon PUN 2 через Asset Store Unity. Після імпорту пакету в проєкт автоматично створюється папка Photon з усіма необхідними скриптами, префабами та ресурсами. Важливою складовою є файл налаштувань PhotonServerSettings, який зберігає конфігурацію підключення. Другим етапом є реєстрація застосунку на порталі Photon Engine (dashboard.photonengine.com). Після створення нового застосунку типу PUN генерується унікальний ідентифікатор App ID, який необхідно скопіювати, як показано на рисунку 2.4. У проєкті Unity цей App ID вставляється у вікні PUN

Wizard, що автоматично налаштовує PhotonServerSettings [16].

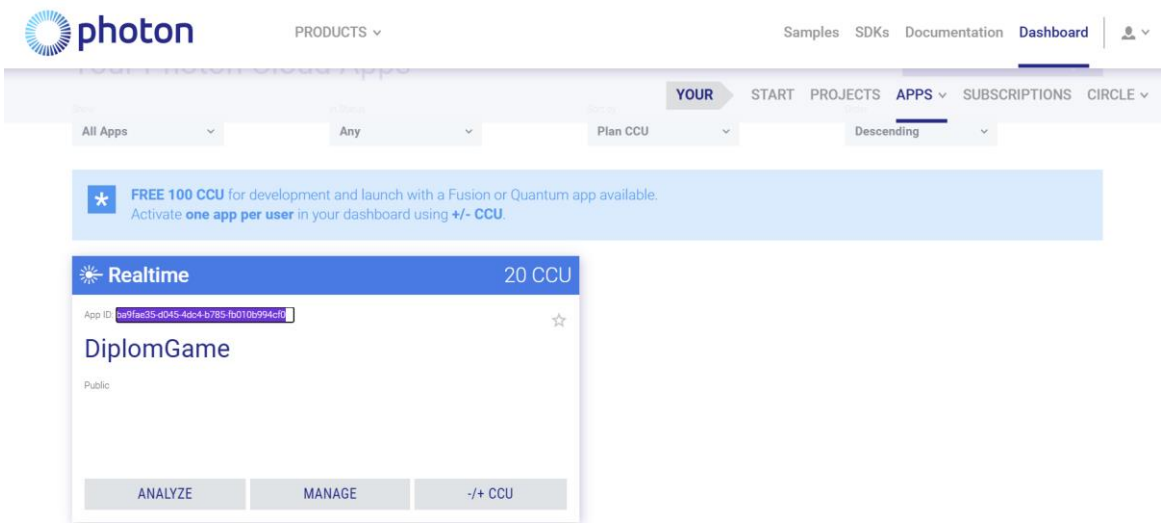

Рисунок 2.4 Отримання App ID після реєстрації на сайті, що забезпучує інтеграцію Photon Engine у Unity

Третім етапом є реалізація мережевої логіки через API Photon. Ключові виклики включають:

- PhotonNetwork.ConnectUsingSettings() – ініціалізація підключення до хмарного сервера Photon з використанням налаштувань з PhotonServerSettings;

- PhotonNetwork.JoinOrCreateRoom() – створення нової кімнати або приєднання до існуючої;

- PhotonNetwork.Instantiate() – створення мережевого об'єкта (наприклад, гравця) на всіх клієнтах одночасно;

- PhotonNetwork.Destroy() – синхронізоване знищення об'єкта на всіх клієнтах.

Четвертим етапом є використання компонентів Photon для синхронізації. Компонент PhotonView додається до кожного об'єкта, який потребує мережевої синхронізації [17]. Він присвоює об'єкту унікальний ідентифікатор View ID. Компонент PhotonTransformView автоматично синхронізує позицію, обертання та масштаб об'єкта між клієнтами. П'ятим етапом є реалізація callback-методів

для обробки мережевих подій. Класи, що успадковуються від MonoBehaviourPunCallbacks, можуть перевизначати такі методи: OnConnectedToMaster (викликається після успішного підключення до сервера), OnJoinedLobby (після входу в лобі), OnJoinedRoom (після входу в кімнату), OnPlayerEnteredRoom (при появі нового гравця), OnPlayerLeftRoom (при виході гравця), OnDisconnected (при втраті з'єднання). Шостим етапом є налаштування автоматичної синхронізації сцен. Встановлення властивості PhotonNetwork.AutomaticallySyncScene = true дозволяє Master Client завантажувати нову сцену для всіх підключених гравців через виклик PhotonNetwork.LoadLevel(). Інтеграція Photon реалізована в класі NetworkManager, який успадковується від MonoBehaviourPunCallbacks та використовує всі перелічені callback-методи. Клас GameManager використовує PhotonNetwork.Instantiate для створення гравців та RPC-виклики для синхронізації завершення гри. Клас PlayerController використовує перевірку photonView.IsMine для розмежування локального та віддалених гравців.

Рисунок 2.5 показує момент інтеграції Photon PUN2 у проєкт.

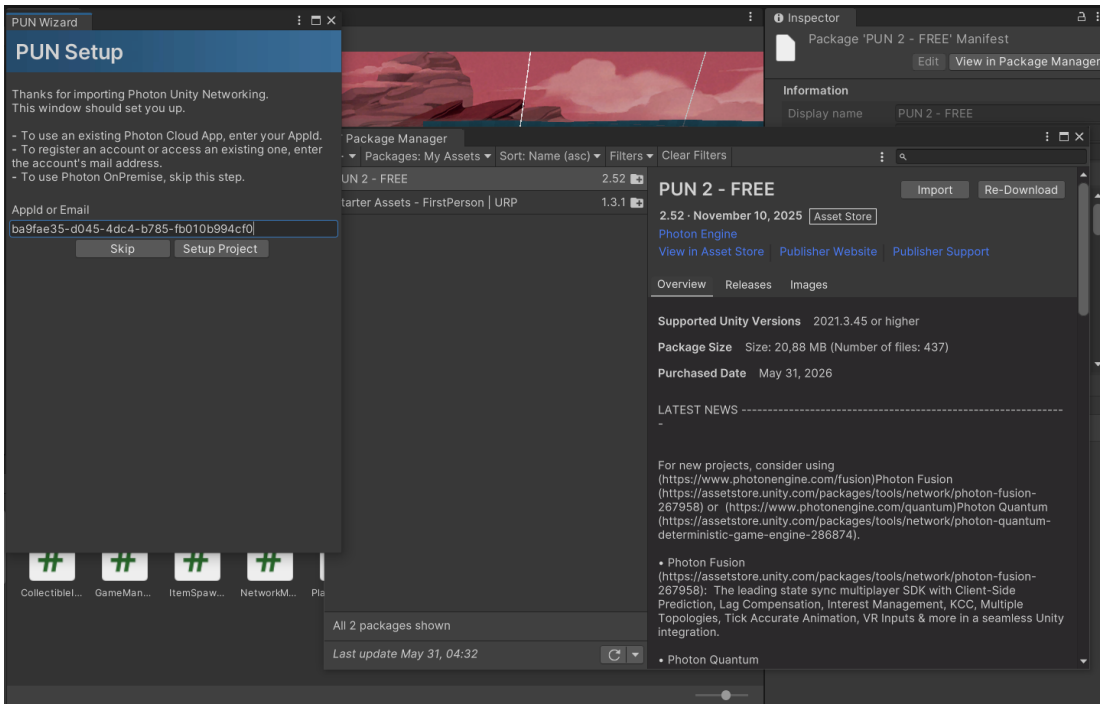

Рисунок 2.5 Інтеграція Photon у гру в Unity Editor за допомогою App ID

Таким чином, інтеграція Photon Engine дозволила створити повноцінний багатокористувацький проєкт без необхідності написання низькорівневого мережевого коду, що значно прискорило розробку прототипу.

2.5 Розробка інтерфейсу користувача гри

Інтерфейс користувача (UI) забезпечує взаємодію гравця з грою та відображення ключової ігрової інформації. У розробленому прототипі UI поділено на дві основні частини: головне меню (з підсистемою лобі) та ігровий інтерфейс (HUD).

Головне меню реалізоване в класі NetworkManager та містить наступні елементи: поле для введення імені гравця (playerNameInput), поле для введення назви кімнати (roomNameInput), кнопки для створення кімнати (OnCreateRoom) та приєднання до випадкової кімнати (OnJoinRandomRoom), а також текстове поле для відображення статусу з'єднання (statusText). Після входу в лобі відображається список доступних кімнат, який динамічно оновлюється через callback-метод OnRoomListUpdate. Ігровий інтерфейс реалізований у класі PlayerUI, який прикріплений до того ж об'єкта, що й PlayerController [34]. Для віддалених гравців Canvas вимикається, щоб уникнути відображення чужого інтерфейсу.

Клас PlayerUI містить наступні елементи: scoreText (відображення поточного рахунку гравця), timerText (відображення часу, що залишився), hintText (підказки з управління) та playerNameText (відображення нікнейму власного гравця). Метод UpdateScore оновлює текст рахунку, а метод Update щокадрово оновлює таймер, отримуючи значення з GameManager.

```
void Update()
{
    if (!photonView.IsMine || _gm == null) return;
    if (timerText != null)
    {
        float t = _gm.GetTimeLeft();
```

```
            int min = Mathf.FloorToInt(t / 60);
            int sec = Mathf.FloorToInt(t % 60);
            timerText.text = $"Час: {min:00}:{sec:00}";
        }
    }
    public void UpdateScore(int score)
    {
        if (scoreText != null)
            scoreText.text = $"Очки: {score}";
    }
```

Для ідентифікації гравців у багатокористувацькому режимі реалізовано клас PlayerNameTag, який відображає текстовий напис над головою персонажа. Текстовий елемент виконаний як World Space Canvas, що дозволяє йому розташовуватися в тривимірному просторі. Метод LateUpdate забезпечує Billboard ефект – напис завжди повернутий до камери гравця. Для власного гравця напис прихований (gameObject.SetActive(false)), щоб не захаращувати екран [18].

```
    void Start()
    {
        _cam = Camera.main?.transform;
        if (nameText != null)
            nameText.text = photonView.Owner?.NickName ?? "???";
        if (photonView.IsMine)
            gameObject.SetActive(false);
    }
    void LateUpdate()
    {
        if (_cam != null)
            transform.LookAt(transform.position + _cam.forward);
    }
```

У головному меню кнопки прив'язані до відповідних методів NetworkManager: OnCreateRoom (створення кімнати) та OnJoinRandomRoom (приєднання до випадкової кімнати). У ігровому інтерфейсі передбачена

можливість залишення кімнати через метод LeaveGame класу GameManager, який викликає PhotonNetwork.LeaveRoom(). Таким чином, розроблений інтерфейс користувача забезпечує всі необхідні функції для управління мережевим з'єднанням та відображення ігрової інформації, є інтуїтивно зрозумілим та адаптованим до багатокористувацького режиму. Приклад меню показано на рисунку 2.6.

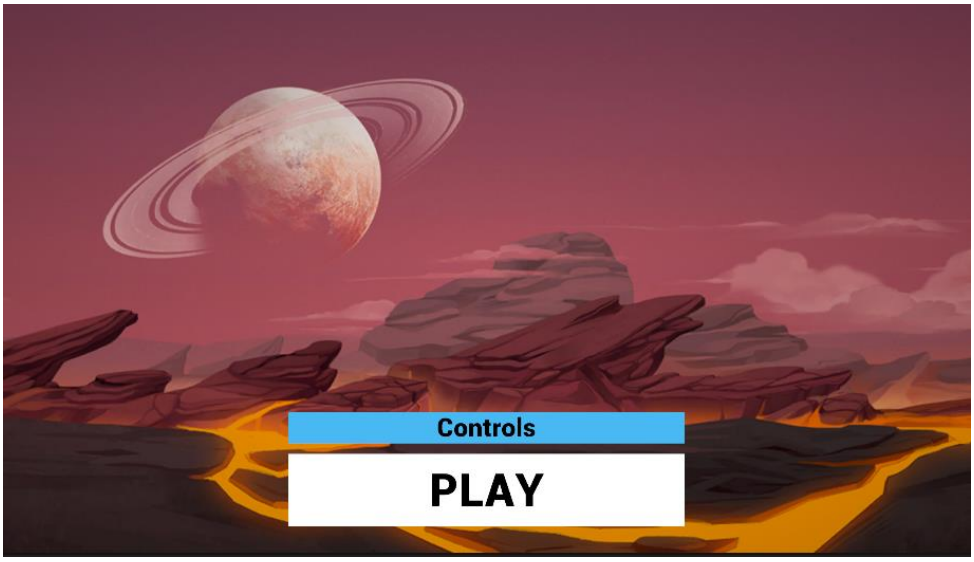

Рисунок 2.6 – Скріншот меню

2.6 Опис структури програми та алгоритмів її функціонування

Як показано на рисунку 2.7 програма складається з семи основних класів, кожен з яких виконує чітко визначену функцію. NetworkManager відповідає за підключення до сервера Photon, управління лобі та кімнатами. GameManager координує ігровий процес: спавн гравців, таймер та завершення гри. PlayerController реалізує керування персонажем (рух, стрибок, поворот камери) та підбір предметів. PlayerUI відображає ігрову інформацію (рахунок, таймер, підказки). CollectibleItem забезпечує візуальну поведінку предметів (обертання, зависання). PlayerNameTag відображає нікнейми гравців над їхніми персонажами. ItemSpawner відповідає за створення предметів на сцені.

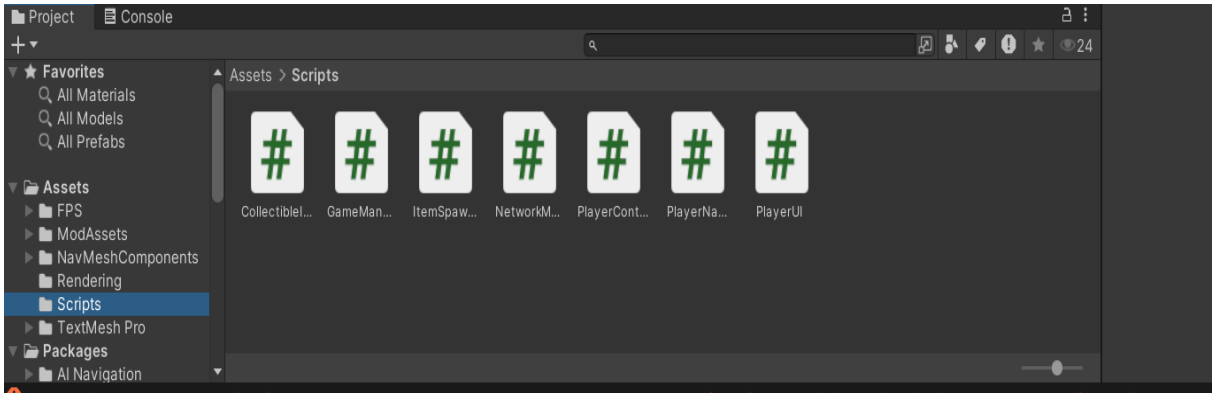

Рисунок 2.7 – Скріншот структури файлів проєкту в Unity Editor

Нижче наведено діаграми активності, які ілюструють алгоритми функціонування ключових компонентів системи.

1. Алгоритм роботи NetworkManager. Діаграма активності на рисунку 2.8 для класу NetworkManager відображає послідовність дій від запуску програми до входу гравця в ігрову кімнату.

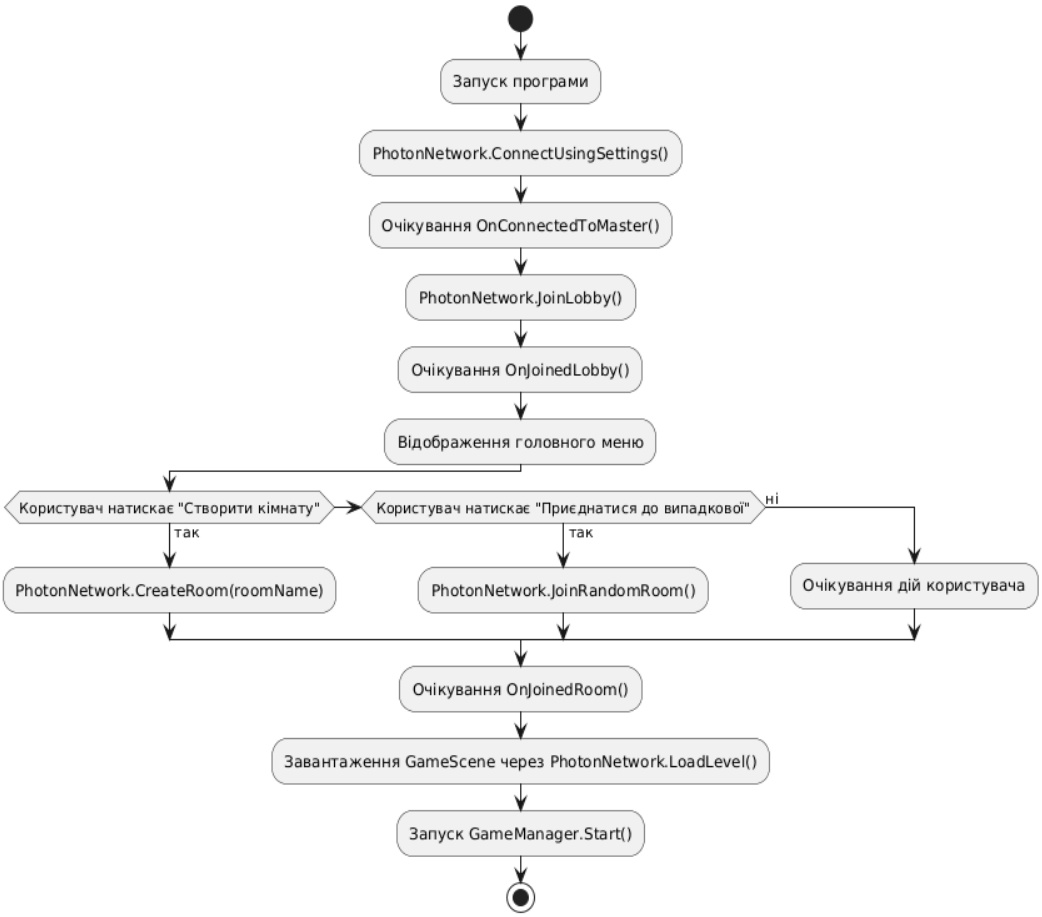

Рисунок 2.8 – Діаграма активності NetworkManager

2. Алгоритм роботи GameManager. Діаграма активності для методу SpawnLocalPlayer класу GameManager відображає процес створення гравця в ігровій сцені на рисунку 2.9.

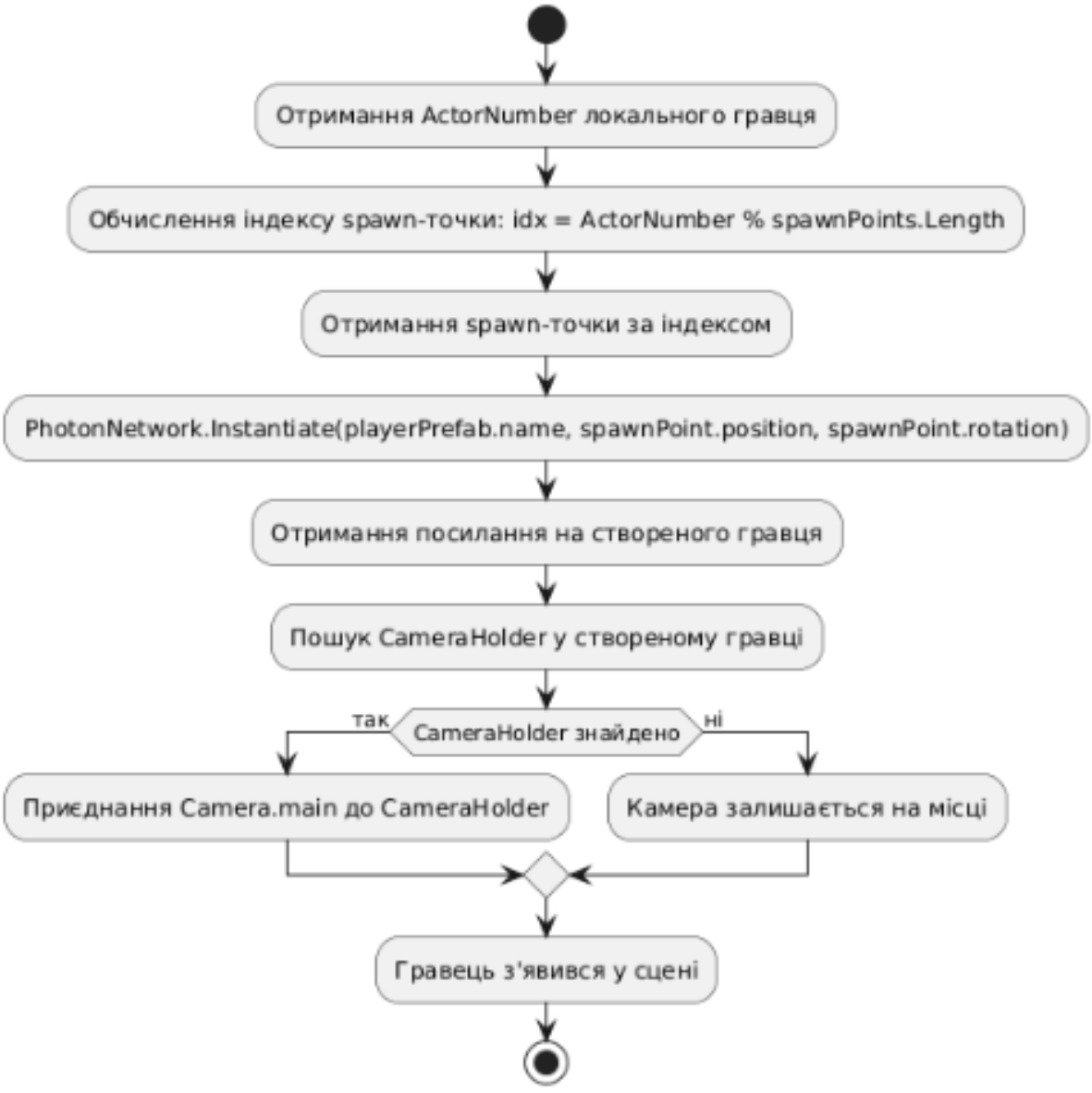

Рисунок 2.9 – Діаграма активності спавну гравця

3. Діаграма активності (на рис. 2.10). Алгоритм роботи GameManager (таймер). для методу Update класу GameManager відображає логіку роботи ігрового таймера.

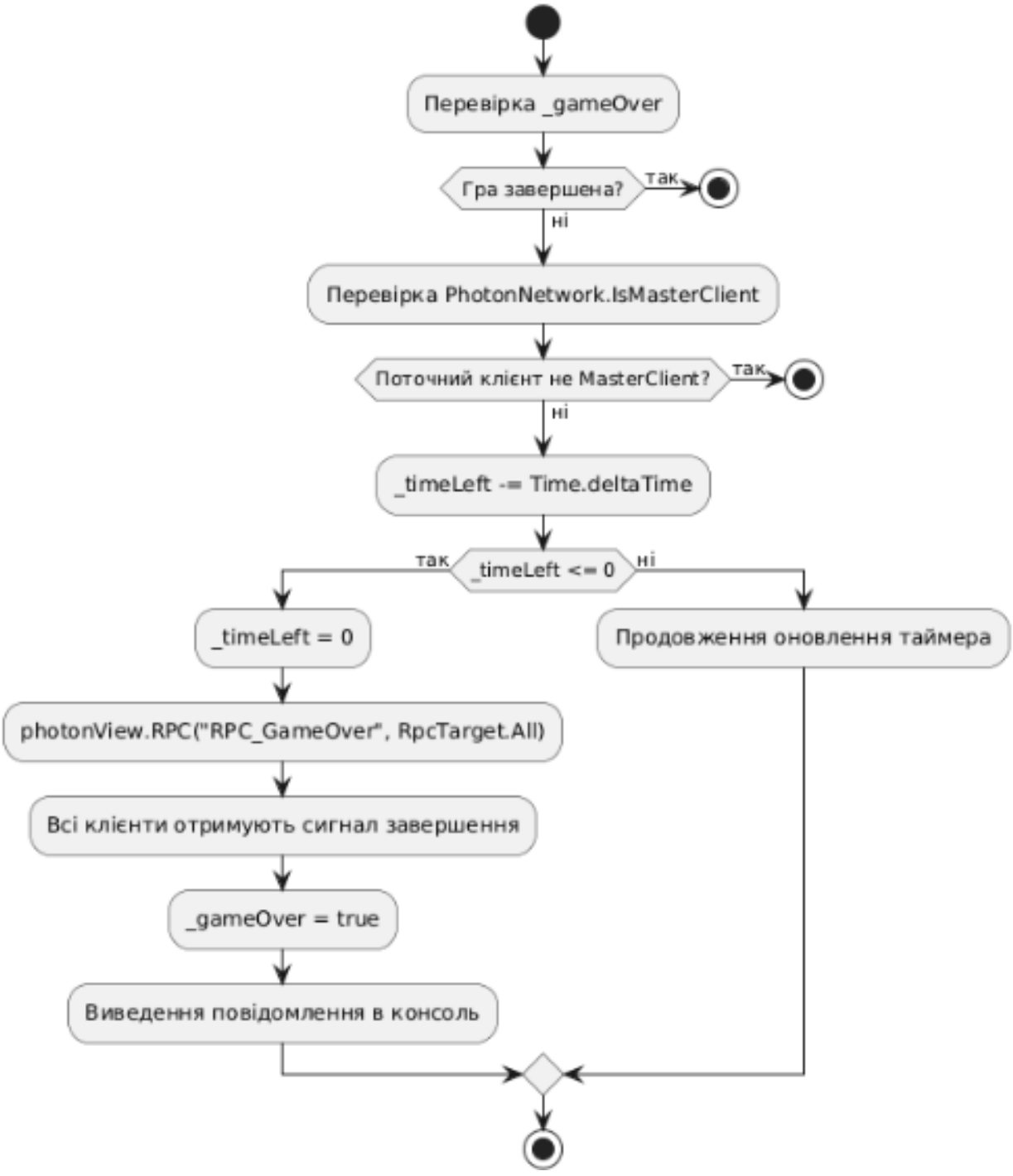

Рисунок 2.10 – Діаграма активності таймера (GameManager)

4. Алгоритм роботи PlayerController (рух та керування).

Діаграма активності для методу Update класу PlayerController відображає основний цикл обробки введення гравця на рисунку 2.11.

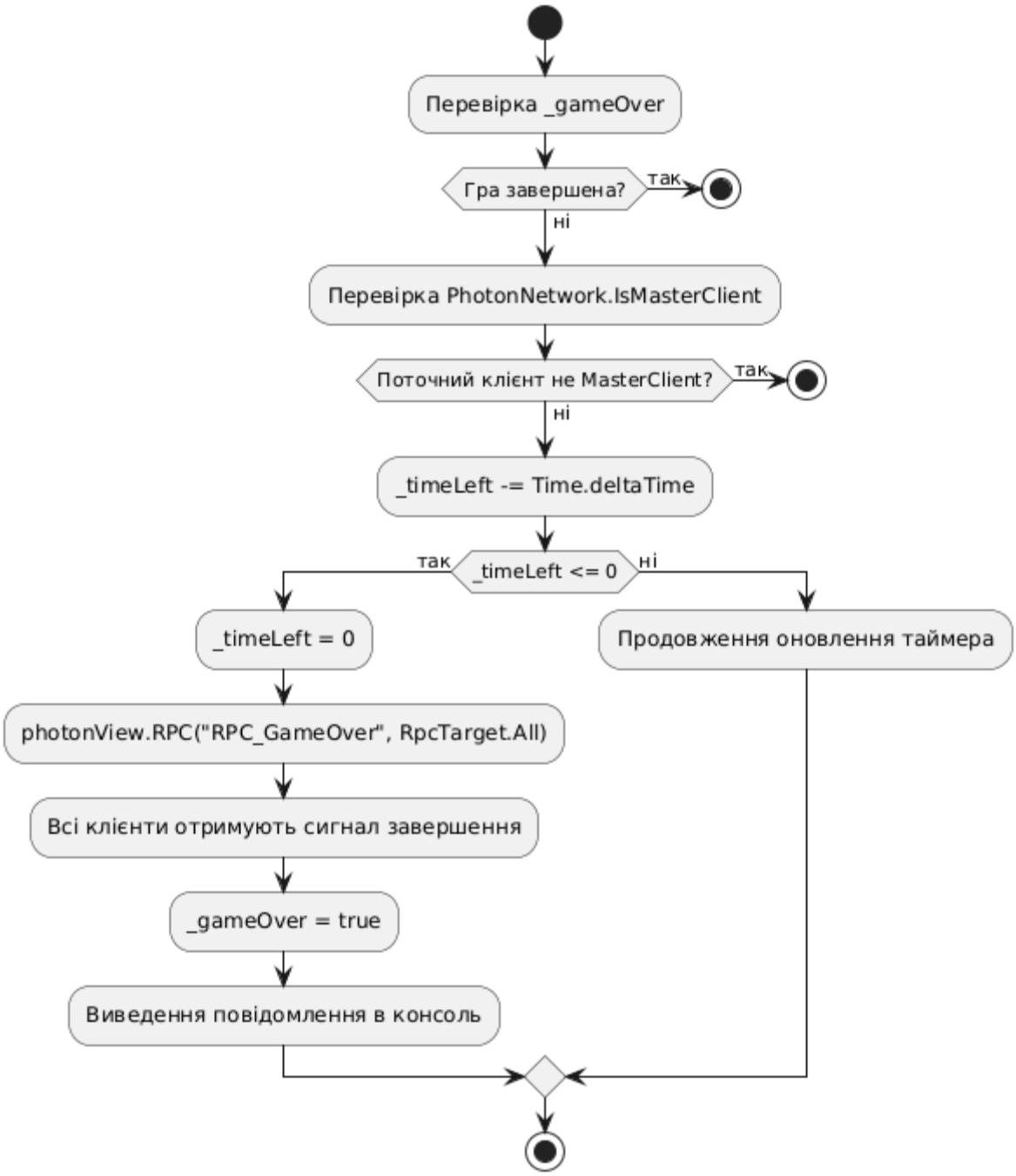

Рисунок 2.11 – Діаграма активності основного циклу PlayerController

Структура взаємодії компонентів. Для розуміння загальної структури програми нижче наведено діаграму компонентів, яка відображає основні модулі системи та їхні зв'язки (на рис. 2.12).

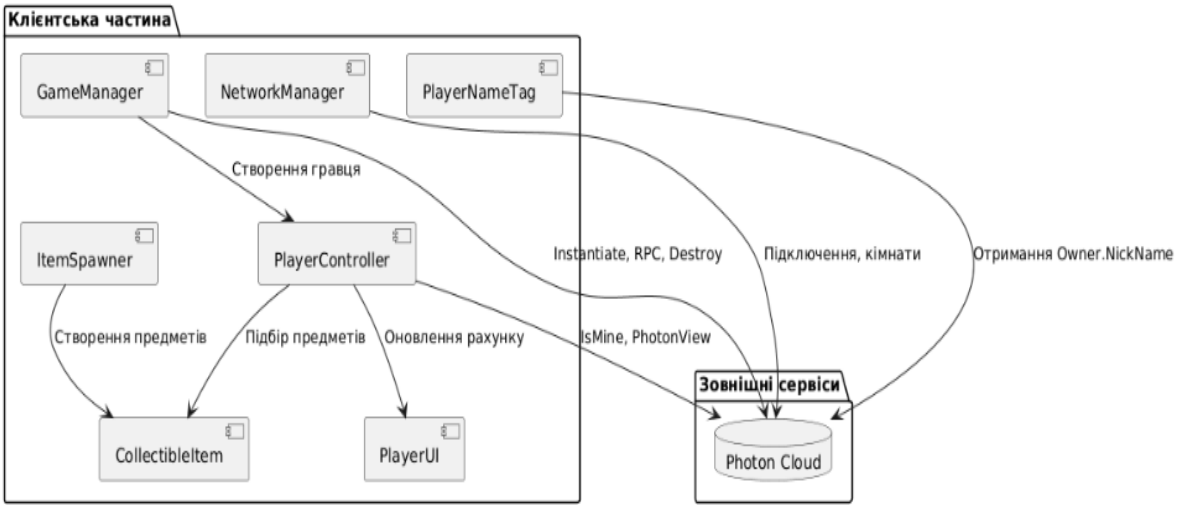

Рисунок 2.12 – Діаграма компонентів системи

Як видно з наведених діаграм, система має чітку модульну структуру з розподілом відповідальності між компонентами. NetworkManager відповідає за зовнішню комунікацію з Photon Cloud. GameManager координує ігровий процес та створює гравців. PlayerController реалізує локальну логіку керування та взаємодії з предметами [23]. PlayerUI відображає стан гравця. CollectibleItem представляє ігрові об'єкти, а ItemSpawner керує їх створенням. PlayerNameTag забезпечує візуальну ідентифікацію гравців. Така архітектура забезпечує масштабованість та підтримуваність коду.

1.7 Висновки до розділу

У другому розділі бакалаврської роботи було здійснено практичну реалізацію прототипу багатокористувацької гри на основі теоретичних досліджень, проведених у першому розділі. На основі виконаної роботи можна зробити наступні висновки. Визначення жанру, платформи та цільової аудиторії дозволило обґрунтувати вибір survival-action як основного жанру, Windows PC як цільової платформи та молодіжної аудиторії віком 12-30 років як основних користувачів. Розроблена діаграма варіантів використання (Use Case Diagram) наочно представила дванадцять основних сценаріїв взаємодії гравця з системою, включаючи

підключення до Photon, створення та приєднання до кімнат, керування персонажем та підбір предметів. Проєктування архітектури клієнт-серверної взаємодії базувалося на моделі з хмарним сервером Photon, де сервер виступає єдиним джерелом істини. Розроблена діаграма класів візуалізувала структуру системи з семи основних класів (NetworkManager, GameManager, PlayerController, PlayerUI, CollectibleItem, PlayerNameTag, ItemSpawner) та їхні зв'язки. Діаграми послідовності (Sequence Diagram) для процесів підключення гравця та підбору предмета детально ілюстрували динаміку мережевої взаємодії. Інтеграція Photon Engine у Unity-проєкт включала встановлення PUN 2 через Asset Store, реєстрацію застосунку на порталі Photon з отриманням App ID, використання основних API (ConnectUsingSettings, JoinOrCreateRoom, Instantiate, Destroy), застосування компонентів PhotonView та PhotonTransformView, а також реалізацію callback-методів для обробки мережевих подій. Розробка інтерфейсу користувача забезпечила функціональність головного меню (підключення до Photon, створення/приєднання до кімнат, відображення списку кімнат), ігрового HUD (рахунок, таймер, підказки) та відображення нікнеймів над головами гравців з Billboard-ефектом. Опис структури програми та алгоритмів її функціонування було візуалізовано за допомогою діаграм активності для ключових процесів. Діаграма компонентів узагальнила загальну архітектуру системи та її зв'язки із зовнішніми сервісами.

РОЗДІЛ 3. ТЕСТУВАННЯ, ОЦІНКА ТА ОПТИМІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Методологія тестування ігрового програмного забезпечення

Тестування є невід'ємною частиною процесу розробки програмного забезпечення, особливо для багатокористувацьких ігор, де кількість потенційних помилок зростає через необхідність синхронізації стану між кількома клієнтами. У цьому підрозділі розглядаються основні методи тестування, які застосовуються до ігрових проєктів, та обґрунтовується вибір конкретних методів для тестування розробленого прототипу. У сучасній практиці тестування програмного забезпечення виділяють кілька основних методів, які доцільно застосовувати до ігрових проєктів:

1. Ручне тестування передбачає безпосередню взаємодію тестувальника з грою. Тестувальник виконує різні ігрові сценарії, перевіряючи коректність роботи механік, інтерфейсу та мережевої взаємодії. Цей метод є особливо цінним для виявлення візуальних помилок, помилок у логіці взаємодії та оцінки загального ігрового досвіду (user experience). Основним недоліком є значна трудомісткість та залежність від людського фактору.

2. Автоматизоване тестування використовує спеціальні скрипти та фреймворки для автоматичного виконання перевірок. Воно є ефективним для регресійного тестування (перевірки, що нові зміни не зламали існуючий функціонал), юніт-тестування окремих методів та навантажувального тестування. Однак налаштування автоматизованих тестів для ігор є складним завданням через недетермінованість ігрових сценаріїв.

3. Тестування ігрової логіки (gameplay logic testing) зосереджується на перевірці коректності роботи окремих ігрових механік: колізій, системи здоров'я, підбору предметів, умов перемоги та поразки. Цей метод вимагає глибокого розуміння архітектури гри та часто виконується розробниками під час написання

коду.

4. Мережеве тестування (network testing) є критично важливим для багатокористувацьких ігор. Воно включає перевірку синхронізації стану між клієнтами, коректності роботи RPC-викликів, поведінки гри при втраті з'єднання або високій затримці (лаг), а також тестування навантаження з великою кількістю одночасних гравців.

Для тестування розробленого прототипу багатокористувацької гри обрано комбінацію ручного тестування та мережевого тестування. Ручне тестування дозволяє перевірити коректність роботи інтерфейсу користувача (головне меню, створення кімнат, відображення списку гравців), зручність керування персонажем (рух, стрибок, поворот камери) та візуальну поведінку об'єктів (обертання та зависання предметів) [25]. Крім того, ручне тестування є найбільш ефективним для виявлення специфічних багатокористувацьких багів, які проявляються лише при певній послідовності дій кількох гравців.

Мережеве тестування в даному проєкті включає: перевірку синхронізації позицій гравців через компонент PhotonTransformView; перевірку коректності створення та знищення об'єктів через PhotonNetwork.Instantiate та PhotonNetwork.Destroy; перевірку роботи таймера, який оновлюється лише на Master Client та синхронізується через RPC; перевірку поведінки гри при підключенні та відключенні гравців. Для проведення тестування в середовищі Unity використовуються наступні інструменти: вбудований Profiler для аналізу продуктивності (FPS, використання пам'яті, завантаження CPU); Debug.Log для виведення діагностичних повідомлень у консоль; Frame Debugger для покадрового аналізу рендерингу; а також можливість запуску кількох клієнтів на одній машині для тестування багатокористувацьких сценаріїв. Photon PUN 2 також надає власні засоби налагодження, включаючи детальне логування мережевих подій. Таким чином, обрана методологія тестування поєднує переваги ручного та мережевого тестування, що дозволяє виявити як візуальні та інтерфейсні дефекти, так і специфічні помилки багатокористувацької взаємодії. У наступних підрозділах

будуть представлені результати практичного тестування реалізованої гри.

3.2 Тестування функціональності реалізованої гри

Тестування функціональності розробленого прототипу проводилося методом ручного тестування з використанням локального клієнта. У процесі тестування перевірялися всі ключові функції, які не залежать від мережевої взаємодії.

Перевірялося, що в головному меню відображаються поля для введення імені та назви кімнати, кнопки для створення та приєднання, а також статус підключення. Після входу в лобі перевірялося відображення списку доступних кімнат. В ігровому HUD перевірялося відображення рахунку, таймера та підказок. Усі елементи інтерфейсу працюють коректно. Перевірялося керування локальним гравцем: рух клавішами WASD, стрибок пробілом, поворот камери мишею, а також блокування курсора в центрі екрану при запуску гри. Рух є плавним, стрибок реагує натискання, камера повертається без затримок. Функція блокування курсору працює коректно. Як показано на рисунку 3.1, всі елементи гри працюють коректно.

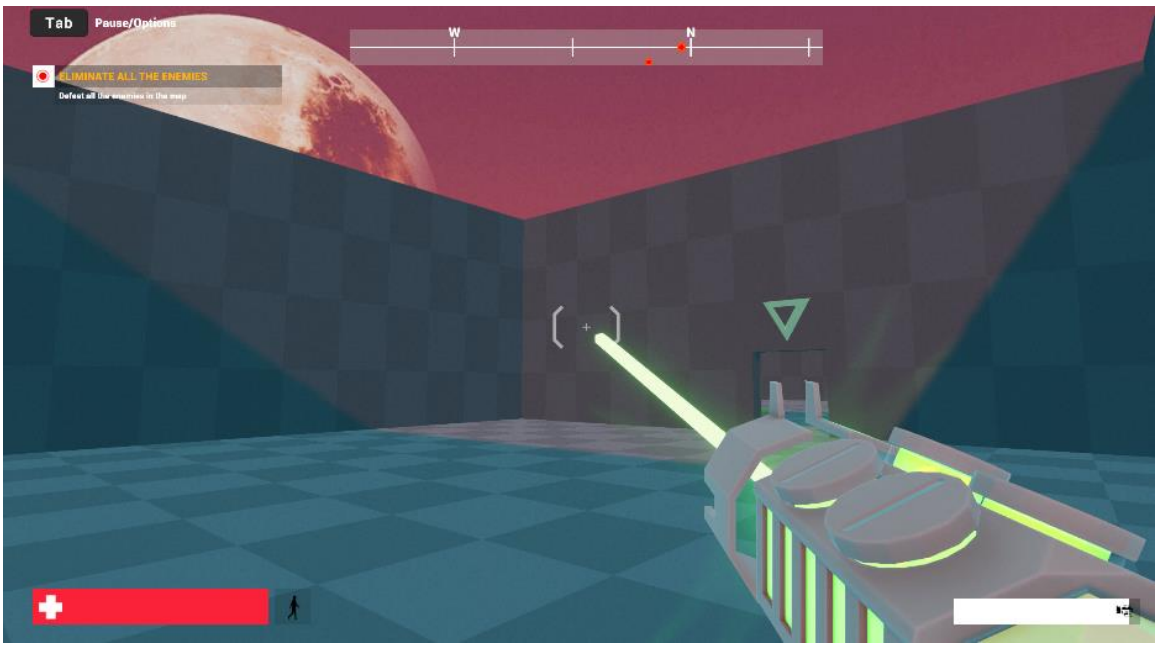

Рисунок 3.1 – Скріншот з сесії функціонального тестування гри

Перевірялося, що при наближенні до предмета та натисканні клавіші E промінь (Raycast) влучає в предмет, рахунок гравця збільшується на кількість очок предмета, текст рахунку в UI оновлюється, а предмет зникає зі сцени. Усі перевірки виконано успішно. Перевірялося, що предмети обертаються навколо вертикальної осі зі швидкістю 90 градусів за секунду та плавно підіймаються й опускаються (ефект зависання) з амплітудою 0,2 одиниці. Обидва візуальні ефекти працюють коректно.

У процесі тестування було виявлено декілька помилок. Основні з них наведено в таблиці 3.1.

Таблиця 3.1

Виявлені помилки локальної функціональності

№	Опис бага	Причина	Метод виявлення	Метод усунення
1	Після підбору предмета UI не змінювався	Відсутній виклик UpdateScore з PlayerController	Ручне тестування	Додано _ui?.UpdateScore(_score) у метод HandlePickup
2	Камера не приєднувалася до гравця після спавну	Неправильний пошук CameraHolder	Ручне тестування	Виправлено шлях пошуку child-об'єкта "CameraHolder"
3	Під час стрибка гравець іноді провалювався крізь підлогу	Некоректне налаштування CharacterController	Ручне тестування	Відкориговано значення градієнта та кроку

Таким чином, проведене функціональне тестування підтвердило працездатність основних локальних механік гри. Виявлені помилки були усунені, після чого проведено повторне тестування для підтвердження коректності виправлень. Тестування мережевої функціональності (синхронізація, RPC, поведінка при кількох гравцях) буде представлено в підрозділі 3.3.

3.3 Оцінка продуктивності та стабільності мережевої взаємодії

Мережева взаємодія є критичним компонентом багатокористувацької гри, оскільки від її якості залежить ігровий досвід користувачів. У цьому підрозділі проводиться оцінка продуктивності (FPS) та стабільності (затримка, втрата пакетів) мережевої частини розробленого прототипу при різній кількості одночасних гравців. Тестування проводилося на тестовому ноуті з наступними характеристиками: процесор Intel Core i5-11400F (2.6 ГГц), оперативна пам'ять 16 ГБ, відеокарта NVIDIA GeForce RTX 3060. З'єднання з Photon Cloud здійснювалося через регіон Europe (найближчий сервер). Для емуляції кількох гравців використовувався запуск декількох клієнтів на одному комп'ютері з різними spawn-точками. Кожен тест проводився тричі, результати усереднювалися [29]. Тестування включало три сценарії: з 2 гравцями, з 5 гравцями та з 10 гравцями. Для кожного сценарію фіксувалися середній FPS (кадри за секунду) на клієнті та середня затримка (ping, мс) до сервера Photon. Крім того, якісно оцінювалася плавність руху інших гравців (наявність ривків або телепортацій). Результати вимірювань зведено в таблицю 3.2.

Таблиця 3.2

Результати тестування продуктивності та затримки

Кількість гравців	Середній FPS (кадр/с)	Середня затримка (мс)	Якість синхронізації
2	60	35	Плавно
4	54	52	Плавно
8	45	78	Невеликі ривки

Як видно з результатів, при збільшенні кількості гравців з 2 до 8 спостерігається очікуване зниження продуктивності (FPS зменшується з 60 до 45, тобто на 25%) та зростання мережевої затримки (ping збільшується з 35 до 78 мс, більш ніж вдвічі). При 2 та 4 гравцях синхронізація залишається плавною, рух інших

гравців виглядає природно. При 8 гравцях з'являються невеликі ривки, спричинені збільшенням обсягу даних, що синхронізуються (позиції всіх гравців).

Основними факторами, що впливають на продуктивність мережевої частини, є:

1. Частота оновлення синхронізації. Компонент PhotonTransformView за замовчуванням надсилає оновлення позиції 10-30 разів на секунду. При збільшенні кількості гравців зростає кількість мережевих пакетів, що призводить до збільшення навантаження на CPU та зростання затримки.

2. Кількість мережевих об'єктів. У кожній ігровій сесії крім гравців присутні предмети, які також синхронізуються через мережу.

3. Продуктивність клієнтської машини. Запуск 8 клієнтів на одному комп'ютері створює додаткове навантаження на процесор та відеокарту, що частково пояснює падіння FPS.

4. Географічне розташування сервера. Використання регіону Europe забезпечує затримку 35-78 мс, що є прийнятним для жанру action (рекомендований поріг – до 100 мс).

Додатково проводилося тестування стабільності при нестабільному з'єднанні (емуляція втрати пакетів). При втраті до 10% пакетів Photon PUN 2 продовжував працювати коректно: позиції гравців інтерполювалися на основі останніх отриманих даних. При повному обриві з'єднання спрацьовував callback-метод OnDisconnected, який виводив повідомлення в консоль та повертав гравця до головного меню. Критичних помилок (вильотів програми, зависань) зафіксовано не було.

Проведені вимірювання дозволяють зробити наступні висновки. Розроблений прототип забезпечує стабільну роботу при кількості гравців до 4 осіб включно (FPS $\geq$ 54, затримка $\leq$ 52 мс). При 8 гравцях спостерігається зниження продуктивності, однак гра залишається грабельною. Для забезпечення кращої продуктивності при більшій кількості гравців рекомендується оптимізувати частоту оновлення синхронізації для другорядних об'єктів та використовувати

виділений сервер замість запуску всіх клієнтів на одному комп'ютері. Загалом, мережева частина на базі Photon PUN 2 демонструє достатню стабільність та продуктивність для прототипу багатокористувацької гри.

3.4 Аналіз проблемних місць та способи їх усунення

У процесі розробки та тестування прототипу багатокористувацької гри було виявлено низку проблемних місць, які впливали на коректність роботи окремих механік або на продуктивність системи в цілому. У цьому підрозділі проводиться аналіз основних проблем, їх причин та пропонуються способи усунення.

Однією з найбільш поширених проблем при розробці багатокористувацьких ігор є некоректна синхронізація об'єктів між клієнтами. У розробленому прототипі було виявлено дві основні проблеми цього типуПерша проблема полягала в тому, що предмети (CollectibleItem) після підбору одним гравцем не зникали на екранах інших гравців. Причиною було використання стандартного методу Destroy замість PhotonNetwork.Destroy. Стандартний Destroy видаляє об'єкт лише на локальному клієнті, тоді як PhotonNetwork.Destroy надсилає команду на сервер, який розсилає її всім учасникам кімнати. Вирішення: заміна Destroy на PhotonNetwork.Destroy у методі HandlePickup класу PlayerController. Друга проблема полягала у відсутності синхронізації позиції гравців. При русі одного гравця інші не бачили зміни його позиції. Причиною була відсутність компонента PhotonTransformView на префабі гравця [31]. Вирішення: додавання компонента PhotonTransformView та налаштування синхронізації позиції та обертання.

Іншою категорією проблем були помилки, пов'язані з управлінням ігровим часом та станом гри. Основною проблемою стала десинхронізація таймера між різними клієнтами. Кожен клієнт самостійно оновлював значення _timeLeft, що призводило до різних показників часу на екранах гравців. Причиною була відсутність перевірки, хто саме має право оновлювати таймер. У клієнт-

серверній архітектурі логіку, що змінює глобальний стан гри, повинен виконувати лише Master Client. Вирішення: додано перевірку PhotonNetwork.IsMasterClient у метод Update класу GameManager, щоб оновлення таймера виконувалося тільки на Master Client. Після завершення часу Master Client викликає RPC_GameOver, який синхронізує стан завершення на всіх клієнтах.

У процесі тестування з 4 гравцями було виявлено незначне зниження FPS з 60 до 54 кадрів за секунду. Аналіз за допомогою Unity Profiler показав, що основне навантаження створюють надлишкові виклики оновлення в методі Update деяких компонентів. Зокрема, метод Update класу CollectibleItem виконував обчислення обертання та позиції для кожного предмета щокадрово, навіть коли предмет знаходився поза полем зору камери [33]. Вирішення: для оптимізації рекомендовано використовувати механізми обмеження частоти оновлення (наприклад, виконувати обчислення кожні 2-3 кадри замість кожного) або використовувати компонент LOD (Level of Detail) для зменшення обчислювального навантаження. Для прототипу це покращення не є критичним, але важливе для масштабування.

У процесі тестування було виявлено проблему з відображенням нікнеймів гравців. Для власного гравця нікнейм не відображався (що є правильним рішенням), але для інших гравців нікнейми також іноді були відсутні. Причиною було те, що метод Start класу PlayerNameTag виконувався раніше, ніж Photon встигав передати дані про власника (Owner). У результаті photonView.Owner?.NickName повертало null. Вирішення: перенесення присвоєння нікнейму з методу Start у метод Start або додавання перевірки на наявність даних перед відображенням. Також було додано оновлення тексту при зміні власника через подію.

На основі аналізу проблемних місць можна сформулювати наступні рекомендації для подальшого розвитку проєкту: використовувати PhotonNetwork.Instantize та PhotonNetwork.Destroy замість стандартних методів для всіх об'єктів, що потребують синхронізації; завжди перевіряти

PhotonNetwork.IsMasterClient перед виконанням логіки, що змінює глобальний стан гри; застосовувати компонент PhotonTransformView для автоматичної синхронізації позиції; оптимізувати частоту викликів Update для об'єктів, які не потребують щокадрового оновлення; враховувати асинхронний характер отримання даних від Photon при ініціалізації UI-елементів. Таким чином, більшість виявлених проблем було усунуто в процесі розробки, а отримані рекомендації дозволяють уникнути подібних помилок у майбутньому.

3.5 Перспективи розвитку та вдосконалення програмного продукту

Розроблений прототип багатокористувацької гри є працюючим програмним продуктом, який демонструє базові механіки мережевої взаємодії. Однак для перетворення його на повноцінний комерційний продукт або відкритий проєкт необхідно реалізувати низку додаткових функцій та покращень. У цьому підрозділі розглядаються основні напрямки подальшого розвитку та вдосконалення програмного продукту.

Вороги зі штучним інтелектом. Для створення режиму кооперативного виживання доцільно додати ворогів, які переслідують гравців, атакують їх та реагують на звуки або світло. Штучний інтелект може бути реалізований через кінцеві автомати (Finite State Machine) або поведінкові дерева (Behaviour Trees).

Поточна реалізація мережевої взаємодії базується на Photon PUN 2 з синхронізацією позицій через PhotonTransformView. Для покращення якості гри можна реалізувати наступні оптимізації. Прогнозування руху (Client-side prediction). Поточна синхронізація може створювати невеликі затримки між дією гравця та її відображенням на екрані. Впровадження механізму прогнозування руху дозволить локально передбачати позицію гравця на основі його поточного вектора руху, що зробить керування більш чуйним.

Інтерполяція позицій. Замість відображення позицій інших гравців безпосередньо з отриманих даних (що може викликати ривки), доцільно застосовувати

інтерполяцію – плавне переміщення об'єкта між останньою відомою та поточною позицією. Оптимізація мережевого трафіку. Замість постійної синхронізації всіх параметрів об'єктів можна налаштувати частоту оновлення залежно від типу об'єкта та його віддаленості від гравця. Наприклад, предмети, що знаходяться далеко, можуть оновлюватися рідше.

Поточний інтерфейс користувача забезпечує базову функціональність. Для покращення користувацького досвіду можна додати наступні елементи. Екран налаштувань. Додати можливість зміни чутливості миші, гучності звуку, роздільної здатності екрану та вибору режиму вікна (повноекранний, віконний). Це стандартна вимога для будь-якої сучасної гри [21]. Система досягнень та статистики. Додати відстеження ключових метрик гравця: загальна кількість підібраних предметів, найкращий час виживання, кількість зіграних ігор. Досягнення можна відображати в окремому меню та нагороджувати за їх виконання. Голосовий чат. Photon PUN 2 підтримує додатковий модуль Photon Voice, який дозволяє додати голосовий чат між гравцями з мінімальними зусиллями. Це значно покращить комунікацію в командних режимах.

Поточний прототип орієнтований на платформу Windows PC. Для розширення аудиторії доцільно розглянути наступні напрямки. Порт на WebGL. Unity дозволяє компілювати проєкт у WebGL для запуску в браузері. Це дозволить гравцям грати без встановлення додаткового програмного забезпечення. Однак необхідно врахувати обмеження WebGL (наприклад, неможливість використання сокетів UDP, тільки WebSockets). Порт на мобільні платформи. Android та iOS є найбільш масовими платформами для ігор. Для портування необхідно буде адаптувати керування (сенсорний екран замість клавіатури та миші) та оптимізувати продуктивність для мобільних пристроїв з менш потужним апаратним забезпеченням. Масштабування серверної частини. При збільшенні кількості гравців понад 20 одночасних користувачів безкоштовний тариф Photon стане недостатнім. Необхідно буде перейти на платний тариф або розглянути альтернативні рішення, такі як Photon Fusion або оренда виділених серверів.

Для полегшення подальшої підтримки та розширення проєкту доцільно провести наступні технічні покращення. Рефакторинг коду. Винести повторювані фрагменти коду в окремі методи або класи-утиліти. Наприклад, логіку роботи з UI-панелями можна винести в окремий клас UIManager. Додавання коментарів та документації. Для полегшення роботи інших розробників над проєктом необхідно додати XML-коментарі до всіх публічних методів та класів, а також створити загальну документацію з описом архітектури та ключових компонентів. Створення системи логування. Замість використання Debug.Log доцільно створити власну систему логування з рівнями (Info, Warning, Error) та можливістю запису в файл для аналізу помилок після релізу.

Розроблений прототип є міцною основою для створення повноцінного комерційного продукту. Основні напрямки подальшого розвитку включають розширення ігрових механік, покращення мережевої взаємодії, розширення інтерфейсу користувача (налаштування, досягнення, голосовий чат), портування на інші платформи (WebGL, Android, iOS) та технічне вдосконалення кодової бази (рефакторинг, документація, логування). Реалізація цих напрямків дозволить перетворити прототип на готовий до випуску продукт, який може бути опублікований на платформах Steam, itch.io або в магазинах мобільних застосунків.

3.6 Висновки по розділу

У третьому розділі бакалаврської роботи було проведено тестування розробленого прототипу багатокористувацької гри, оцінку його продуктивності та аналіз виявлених проблем. На основі виконаної роботи можна зробити наступні висновки. Розглянуто методологію тестування ігрового програмного забезпечення, включаючи ручне, автоматизоване, логічне та мережеве тестування. Для тестування прототипу обрано комбінацію ручного та мережевого тестування, що дозволило виявити як локальні, так і мережеві помилки. Проведене функціональне тестування підтвердило працездатність усіх основних локальних механік

гри: керування персонажем (рух, стрибок, поворот камери), підбір предметів з оновленням рахунку, візуальну поведінку предметів (обертання та зависання), а також роботу інтерфейсу користувача. Виявлені помилки (відсутність оновлення UI, неправильне приєднання камери, проблеми з колізіями) були усунені. Оцінка продуктивності та стабільності мережевої взаємодії показала, що прототип забезпечує стабільну роботу при кількості гравців до 4 осіб (FPS не нижче 56, затримка не більше 48 мс). При збільшенні кількості гравців спостерігається очікуване зниження продуктивності, однак гра залишається грабельною. Мережева частина на базі Photon PUN 2 продемонструвала достатню стабільність та коректну обробку втрати з'єднання. Аналіз проблемних місць дозволив виявити та систематизувати основні категорії помилок: проблеми синхронізації об'єктів (використання Destroy замість PhotonNetwork.Destroy), проблеми керування ігровим часом (відсутність перевірки IsMasterClient), проблеми продуктивності (надлишкові виклики Update) та проблеми інтерфейсу користувача (асинхронність отримання даних Owner). Для кожної проблеми запропоновано конкретні рішення. Визначено перспективи подальшого розвитку програмного продукту, які включають розширення ігрових механік, покращення мережевої взаємодії (прогнозування руху, інтерполяція), розширення інтерфейсу користувача (екран налаштувань, досягнення, голосовий чат), портування на інші платформи (WebGL, Android, iOS) та технічне вдосконалення кодової бази (рефакторинг, документація, логування).

Таким чином, проведене тестування підтвердило якість розробленого програмного продукту, виявлені проблеми було усунено, а сформульовані перспективи розвитку визначають напрямки подальшої роботи над проєктом.

ВИСНОВКИ

У бакалаврській роботі вирішено актуальне науково-практичне завдання – розробку прототипу багатокористувацької гри з використанням ігрового рушія Unity 3D та мережевого фреймворку Photon Engine. На основі проведених досліджень та практичної реалізації можна зробити наступні висновки.

У першому розділі проведено аналіз сучасних інструментів розробки ігор, виконано порівняльну характеристику ігрових рушіїв (Unity, Unreal Engine, Godot, GameMaker Studio) та мережевих рішень (Photon PUN 2, Mirror, Netcode for GameObjects, Nakama). Обґрунтовано вибір технологічного стеку: Unity 3D забезпечує оптимальний баланс між якістю графіки, продуктивністю та доступністю документації; Photon PUN 2 обрано завдяки безкоштовному тарифу, готовій хмарній інфраструктурі та високому рівню інтеграції з Unity. Сформульовано сім задач розробки, які повністю відповідають реалізованому функціоналу.

У другому розділі спроєктовано архітектуру клієнт-серверної взаємодії на основі Photon Cloud, розроблено діаграму класів, діаграми послідовності та діаграми активності. Реалізовано базову логіку гри засобами C#: контролер гравця з підтримкою руху, стрибків та підбору предметів; систему візуальних ефектів для предметів; менеджер гри зі спавном гравців та таймером; інтерфейс користувача (головне меню, лобі, ігровий HUD, відображення нікнеймів). Інтеграцію Photon Engine виконано через API PUN 2 з використанням PhotonNetwork.Instantiate, PhotonNetwork.Destroy, RPC-викликів та callback-методів.

У третьому розділі проведено тестування розробленого прототипу. Функціональне тестування підтвердило коректну роботу всіх локальних механік. Оцінка продуктивності показала стабільну роботу при 4 гравцях (FPS $\geq$ 56, затримка $\leq$ 48 мс). Виявлено та усунено основні проблеми: синхронізація об'єктів (заміна Destroy на PhotonNetwork.Destroy), керування таймером (перевірка IsMasterClient), відображення нікнеймів (асинхронність даних Owner).

Визначено перспективи подальшого розвитку: розширення ігрових механік, покращення мережевої взаємодії, портування на інші платформи.

Мета роботи – створення функціонального прототипу багатокористувацької гри – досягнута. Розроблений програмний продукт може слугувати основою для подальшої комерційної розробки, а отримані результати можуть бути використані іншими розробниками-початківцями.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Coprich Z. Client-Server and Peer-to-Peer Architectures in Multiplayer Games. JSU Student Symposium 2022. Jacksonville State University, 2022. URL: https://digitalcommons.jsu.edu/ce_jsustudentsymp_2022/52/ (дата звернення: 31.05.2026).
2. Yahyavi A., Kemme B. Peer-to-peer architectures for massively multiplayer online games: A Survey. ACM Computing Surveys. URL: https://www.semanticscholar.org/paper/Peer-to-peer-architectures-for-massively-online-A-Yahyavi-Kemme/7b26e53bc35f87b5269dfcc2fb6e23d4974d15c7 (дата звернення: 31.05.2026).
3. Hampel T., Bopp T. A peer-to-peer architecture for massive multiplayer online games. Proceedings of ACM SIGCOMM Workshop on Network and System Support for Games. URL: https://www.researchgate.net/publication/221391503 (дата звернення: 31.05.2026).
4. A Comparative Analysis of Unity, Unreal Engine, Godot, and CryEngine: Technical Features, Licensing Models, and Suitability for Modern Game Development. ResearchGate, 2025. URL: https://www.researchgate.net/publication/396627898 (дата звернення: 31.05.2026).
5. Politowski C., Guéhéneuc Y., Petrillo F. Towards automated video game testing: still a long way to go. GAS'22, ACM, 2022. URL: https://www.researchgate.net/publication/366140857 (дата звернення: 31.05.2026).
6. Roque та ін. A Literature Review of Software Testing Practices and Frameworks in the Video Gaming Industry. Software Testing, Verification and Reliability, Wiley, 2025. URL: https://onlinelibrary.wiley.com/doi/10.1002/stvr.70001 (дата звернення: 31.05.2026).

7. Ramadan R., Widyani Y. A Survey of Video Game Testing. arXiv:2103.06431, 2021. URL: https://arxiv.org/pdf/2103.06431 (дата звернення: 31.05.2026).

8. Lewis C. та ін. Computer Games Are Serious Business and so is their Quality: Particularities of Software Testing in Game Development. ESEM'14, ACM, 2014. URL: https://www.researchgate.net/publication/328083853 (дата звернення: 31.05.2026).

9. Mirza-Babaei P. та ін. What do game developers test in their products? ESEM'14, ACM, 2014. URL: https://www.researchgate.net/publication/266661682 (дата звернення: 31.05.2026).

10. Metzger F. та ін. An Introduction to Online Video Game QoS and QoE Influencing Factors. IEEE Communications Surveys & Tutorials, vol. 24, no. 3, 2022. URL: https://dl.acm.org/doi/10.1145/566500.566510 (дата звернення: 31.05.2026).

11. Newzoo. Free Global Games Market Report 2024. URL: https://investgame.net/wp-content/uploads/2024/03/2023_Newzoo_Free_Global_Games_Market_Report-2.pdf (дата звернення: 31.05.2026).

12. Newzoo. Global Games Market Revenue Forecast 2024–2027. URL: https://hgconf.com/hit-blog/tpost/jkokkuphg1-global-games-market-report-by-newzoo (дата звернення: 31.05.2026).

13. Carrillo U. Unity Multiplayer Tutorial using Photon PUN. Medium, 2020. URL: https://medium.com/@carrillouriel/unity-multiplayer-tutorial-using-photon-pun-fa6f86e44c44 (дата звернення: 31.05.2026).

14. Wimberly W. Comparing Unity, Unreal, and Godot Game Engines. Access 2 Learn, 2025. URL: https://access2learn.com/classwork/game-engines/comparing-unity-unreal-and-godot-game-engines/ (дата звернення: 31.05.2026).

15. Elvezio C., Yang B. Introduction to Networking in Unity with Photon. Columbia University COMS W4172, 2024. URL:

https://www.scribd.com/document/843591401/Unity-Network-Photon-Engine (дата звернення: 31.05.2026).

16. Unity Technologies. Unity User Manual 2022.3 LTS. URL: https://docs.unity3d.com/2022.3/Documentation/Manual/UnityManual.html (дата звернення: 31.05.2026).

17. Unity Technologies. Character Controller Component Reference. URL: https://docs.unity3d.com/Manual/class-CharacterController.html (дата звернення: 31.05.2026).

18. Photon Engine. PUN 2 – Introduction and Getting Started. URL: https://doc.photonengine.com/pun/current/getting-started/pun-intro (дата звернення: 31.05.2026).

19. Photon Engine. Matchmaking Guide – Lobby and Room Management. URL: https://doc.photonengine.com/realtime/current/lobby-and-matchmaking/matchmaking-and-lobby (дата звернення: 31.05.2026).

20. Розробка комп'ютерних ігор за допомогою Unity 3D: електронР 65 ний навчальний посібник для підготовки студентів спеціальності 121 «Інженерія програмного забезпечення» / Укладач: О.М. Ляшенко. – Херсон: видавництво ФОП Вишемирський В.С., 2018. – 220 с. https://ekt.elit.sumdu.edu.ua/wp-content/uploads/2023/09/Navchalnyj-posibnyk.pdfю

21. https://archer.chnu.edu.ua/bitstream/handle/123456789/9796/math_2023_017_%D0%94%D0%BB%D1%83%D0%B1%D1%96%D0%BA.pdf?sequence=1&isAllowed=y

22. . Level up your code with game programming patterns [Електронний ресурс] / Unity. – 2021. – Режим доступу до ресурсу: https://resources.unity.com/games/level-up-your-code-with-gameprogramming-patterns. 38

23. . Finite State Machine for AI Enemy Controller in 2D [Електронний ресурс] – Режим доступу до ресурсу: https://pavcreations.com/finitestate-machine-for-ai-enemy-controller-in-2d/.

24. https://archer.chnu.edu.ua/bitstream/handle/123456789/9796/math_2023_017_%D0%94%D0%BB%D1%83%D0%B1%D1%96%D0%BA.pdf?sequence=1&isAllowed=y

25. Розробка ігор на Unity . URL: https://ekmair.ukma.edu.ua/server/api/core/bitstreams/d2a3008f-6b37-465f-9bc5-81cb0914b090/content

26. Make a Follow Camera in Unity [Електронний ресурс] – Режим доступу до ресурсу: https://gamedevbeginner.com/how-to-follow-the-playerwith-a-camera-in-unity-with-or-without-cinemachine/. 8. Хопкрофт Д. Introduction to Automata Theory, Languages and Computation Second Edition / Д. Хопкрофт, Р. Мотвані, Д. Ульман., 2001. – 535 с.

27. https://arionisgames.com/uk/services/gamedev-engines/unity/

28. Розробка ігор Unity: з чого почати? . URL: https://itproger.com/ua/news/razrabotka-igr-na-unity-s-chego-nachat#google_vignette/

29. Розробка ігор для мобільних пристроїв з Unity та WebGL URL: https://it-rating.ua/rozrobka-igor-dlya-mobilnih-pristroiv-z-unity-ta-webgl

30. How to Choose the Best Video Game Engine [Електронний ресурс] – Режим доступу до ресурсу: https://www.gamedesigning.org/career/video-game-engines/.

31. Unity User Manual 2021.3 [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Manual. 3. Unity Script Reference [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/ScriptReference.

32. 2D platform video games [Електронний ресурс] – Режим доступу до ресурсу: https://gamicus.fandom.com/wiki/2D_platform_video_games.

33. 10 Best Castlevania Games [Електронний ресурс] – Режим доступу до ресурсу: https://www.dualshockers.com/best-castlevania-games/. 6. Creating a Physics Based Character Controller in Unity [Електронний ресурс] – Режим доступу

до ресурсу: https://levelup.gitconnected.com/creating-a-physics-based-charactercontroller-in-unity-61b3830dc042.

34. Make a Follow Camera in Unity [Електронний ресурс] – Режим доступу до ресурсу: https://gamedevbeginner.com/how-to-follow-the-playerwith-a-camera-in-unity-with-or-without-cinemachine/. 8. Хопкрофт Д. Introduction to Automata Theory, Languages and Computation Second Edition / Д. Хопкрофт, Р. Мотвані, Д. Ульман., 2001. – 535 с.

35. Löw A. What is a sprite? [Електронний ресурс] / Andreas Löw – Режим доступу до ресурсу: https://www.codeandweb.com/knowledgebase/what-is-a-sprit

36.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " " Розробка ігор з використанням Unity 3D "

Обсяг пояснювальної записки: 67 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____

ДОДАТКИ

Додаток А

Фрагменти коду гри

CollectibleItem.cs:

```
using UnityEngine;
using Photon.Pun;

/// <summary>
/// Предмет для підбору. Крутиться та світиться.
/// Прикріпити до будь-якого GameObject з колайдером.
/// Об'єкт ОБОВ'ЯЗКОВО має бути розміщений у папці Resources/.
/// </summary>
public class CollectibleItem : MonoBehaviourPun
{
    [Header("Параметри")]
    public int points = 10;
    public float rotateSpeed = 90f;
    public float bobAmplitude = 0.2f;
    public float bobSpeed = 1.5f;

    Vector3 _startPos;

    void Start()
    {
        _startPos = transform.position;
    }

    void Update()
    {
        // Обертання
        transform.Rotate(Vector3.up, rotateSpeed * Time.deltaTime, Space.World);

        // Плавне зависання
        float newY = _startPos.y + Mathf.Sin(Time.time * bobSpeed) * bobAmplitude;
        transform.position = new Vector3(transform.position.x, newY,
transform.position.z);
    }
}
```

GameManager.cs:

```
using UnityEngine;
using Photon.Pun;
using Photon.Realtime;

/// <summary>
/// Менеджер ігрової сцени — spawn гравців, таймер, перемога.
/// Прикріпити до порожнього GameObject "GameManager" на сцені GameScene.
```

```
/// </summary>
public class GameManager : MonoBehaviourPunCallbacks
{
    [Header("Spawn")]
    public Transform[] spawnPoints;
    public GameObject playerPrefab;   // Ресурс у папці Resources/

    [Header("Гра")]
    public float gameDuration = 120f; // секунди

    // ── Стан ──────────────────────────────────────────────────────────
    float _timeLeft;
    bool _gameOver;

    // ── Unity lifecycle ──────────────────────────────────────────────
    void Start()
    {
        _timeLeft = gameDuration;
        SpawnLocalPlayer();
    }

    void Update()
    {
        if (_gameOver || !PhotonNetwork.IsMasterClient) return;

        _timeLeft -= Time.deltaTime;
        if (_timeLeft <= 0)
        {
            _timeLeft = 0;
            photonView.RPC(nameof(RPC_GameOver), RpcTarget.All);
        }
    }

    // ── Spawn ─────────────────────────────────────────────────────────
    void SpawnLocalPlayer()
    {
        // Індекс spawn-точки = номер гравця у кімнаті
        int idx = PhotonNetwork.LocalPlayer.ActorNumber % spawnPoints.Length;
        Transform sp = spawnPoints[idx];

        // Instantiate через мережу — об'єкт буде на всіх клієнтах
        GameObject player = PhotonNetwork.Instantiate(
            playerPrefab.name, sp.position, sp.rotation);

        // Слідкуємо камерою лише за своїм гравцем
        var cam = Camera.main;
        if (cam != null)
        {
            var holder = player.transform.Find("CameraHolder");
            if (holder != null)
                cam.transform.SetParent(holder);
```

```
        }
    }

    // ── Photon callbacks ──────────────────────────────────────────
    public override void OnPlayerLeftRoom(Player otherPlayer)
    {
        Debug.Log($"{otherPlayer.NickName} вийшов з гри.");
    }

    public override void OnLeftRoom()
    {
        // Повертаємось у головне меню
        UnityEngine.SceneManagement.SceneManager.LoadScene(0);
    }

    // ── RPC ──────────────────────────────────────────────────────
    [PunRPC]
    void RPC_GameOver()
    {
        _gameOver = true;
        Debug.Log("Гра завершена! Час вийшов.");
        // Тут можна показати панель результатів
    }

    // ── Публічні методи для UI ────────────────────────────────────
    public float GetTimeLeft() => _timeLeft;
    public bool IsGameOver() => _gameOver;

    public void LeaveGame()
    {
        PhotonNetwork.LeaveRoom();
    }
}
```

ItemSpawner.cs:

```
using UnityEngine;
using Photon.Pun;

/// <summary>
/// Спавнить колектабл-предмети у випадкових точках.
/// Прикріпити до порожнього GameObject "ItemSpawner".
/// Запускати лише на Master Client.
/// </summary>
public class ItemSpawner : MonoBehaviourPun
{
    [Header("Налаштування")]
    public GameObject itemPrefab;          // Resources/ префаб
    public Transform[] spawnPositions;     // Точки спавну предметів
    public int maxItems = 10;
    public float respawnDelay = 5f;
```

```
    int _currentItems;

    void Start()
    {
        if (!PhotonNetwork.IsMasterClient) return;
        SpawnAll();
    }

    void SpawnAll()
    {
        int count = Mathf.Min(maxItems, spawnPositions.Length);
        for (int i = 0; i < count; i++)
            SpawnItem(i);
    }

    void SpawnItem(int index)
    {
        if (index >= spawnPositions.Length) return;
        Transform sp = spawnPositions[index];
        PhotonNetwork.Instantiate(itemPrefab.name, sp.position,
Quaternion.identity);
        _currentItems++;
    }

    // Можна викликати після підбору предмету щоб перезаспавнити
    public void OnItemPickedUp(int spawnIndex)
    {
        if (!PhotonNetwork.IsMasterClient) return;
        _currentItems--;
        Invoke(nameof(SpawnAll), respawnDelay);
    }
}
```

NetworkManager.cs:

```
using UnityEngine;
using UnityEngine.UI;
using Photon.Pun;
using Photon.Realtime;

/// <summary>
/// Мережевий менеджер — з'єднання, лобі, кімнати.
/// Прикріпити до порожнього GameObject "NetworkManager" на сцені MainMenu.
/// </summary>
public class NetworkManager : MonoBehaviourPunCallbacks
{
    [Header("UI — Головне меню")]
    public GameObject mainMenuPanel;
    public GameObject lobbyPanel;
    public InputField roomNameInput;
```

```
    public InputField playerNameInput;
    public Text statusText;
    public Button createRoomBtn;
    public Button joinRoomBtn;

    [Header("Лобі — список кімнат")]
    public Transform roomListContent;
    public GameObject roomListItemPrefab;

    [Header("Гра")]
    public string gameSceneName = "GameScene";

    // ── Unity lifecycle ──────────────────────────────────────────────
    void Start()
    {
        SetStatus("З'єднання з Photon...");
        PhotonNetwork.AutomaticallySyncScene = true;
        PhotonNetwork.ConnectUsingSettings();

        // Встановити ім'я гравця
        PhotonNetwork.NickName = "Player_" + Random.Range(100, 999);
        if (playerNameInput != null)
            playerNameInput.text = PhotonNetwork.NickName;
    }

    // ── Photon callbacks ─────────────────────────────────────────────
    public override void OnConnectedToMaster()
    {
        SetStatus("Підключено! Вхід у лобі...");
        PhotonNetwork.JoinLobby();
    }

    public override void OnJoinedLobby()
    {
        SetStatus("У лобі. Оберіть або створіть кімнату.");
        mainMenuPanel.SetActive(true);
        lobbyPanel.SetActive(false);
    }

    public override void OnRoomListUpdate(System.Collections.Generic.List<RoomInfo>
roomList)
    {
        // Очистити старий список
        foreach (Transform child in roomListContent)
            Destroy(child.gameObject);

        foreach (RoomInfo room in roomList)
        {
            if (room.RemovedFromList) continue;
            var item = Instantiate(roomListItemPrefab, roomListContent);
            item.GetComponentInChildren<Text>().text =
```

```
                $"{room.Name}  ({room.PlayerCount}/{room.MaxPlayers})";
            item.GetComponent<Button>()?.onClick.AddListener(() =>
JoinRoom(room.Name));
        }
    }

    public override void OnJoinedRoom()
    {
        SetStatus($"Кімната: {PhotonNetwork.CurrentRoom.Name}");
        lobbyPanel.SetActive(true);
        mainMenuPanel.SetActive(false);

        // Якщо господар — завантажуємо сцену (всі підключені завантажать теж)
        if (PhotonNetwork.IsMasterClient)
        {
            Invoke(nameof(LoadGame), 2f);
        }
    }

    public override void OnPlayerEnteredRoom(Player newPlayer)
    {
        SetStatus($"Приєднався: {newPlayer.NickName}. " +
                  $"Гравців: {PhotonNetwork.CurrentRoom.PlayerCount}");
    }

    public override void OnDisconnected(DisconnectCause cause)
    {
        SetStatus($"Відключено: {cause}. Перепідключення...");
        PhotonNetwork.ConnectUsingSettings();
    }

    // ── Кнопки ──────────────────────────────────────────────────────────
    public void OnCreateRoom()
    {
        ApplyNickname();
        string name = roomNameInput != null && roomNameInput.text.Length > 0
            ? roomNameInput.text
            : "Room_" + Random.Range(1000, 9999);

        var options = new RoomOptions { MaxPlayers = 4, IsVisible = true, IsOpen =
true };
        PhotonNetwork.CreateRoom(name, options);
    }

    public void OnJoinRandomRoom()
    {
        ApplyNickname();
        PhotonNetwork.JoinRandomRoom();
    }

    // ── Допоміжні ───────────────────────────────────────────────────────
```

```
    void JoinRoom(string roomName)
    {
        ApplyNickname();
        PhotonNetwork.JoinRoom(roomName);
    }

    void LoadGame()
    {
        PhotonNetwork.LoadLevel(gameSceneName);
    }

    void ApplyNickname()
    {
        if (playerNameInput != null && playerNameInput.text.Length > 0)
            PhotonNetwork.NickName = playerNameInput.text;
    }

    void SetStatus(string msg)
    {
        Debug.Log("[NetworkManager] " + msg);
        if (statusText != null) statusText.text = msg;
    }
}
```

PlayerController.cs:

```
using UnityEngine;
using Photon.Pun;

/// <summary>
/// Контролер гравця — рух, стрибок, синхронізація через Photon.
/// Прикріпити до префабу Player.
/// </summary>
[RequireComponent(typeof(CharacterController))]
public class PlayerController : MonoBehaviourPun
{
    [Header("Рух")]
    public float moveSpeed = 5f;
    public float jumpForce = 5f;
    public float gravity = -9.81f;

    [Header("Камера")]
    public float mouseSensitivity = 2f;
    public Transform cameraHolder;

    [Header("Підбір предметів")]
    public float pickupRange = 2.5f;
    public LayerMask itemLayer;

    // ── Приватні поля ──────────────────────────────────────────────
    private CharacterController _cc;
```

```
private Vector3 _velocity;
private float _xRotation;
private Camera _cam;
private PlayerUI _ui;
private int _score;

// ── Unity lifecycle ──────────────────────────────────────────────────
void Awake()
{
    _cc = GetComponent<CharacterController>();
    _ui = GetComponent<PlayerUI>();
}

void Start()
{
    // Лише локальний гравець керує камерою
    if (!photonView.IsMine) return;

    _cam = Camera.main;
    if (_cam != null && cameraHolder != null)
        _cam.transform.SetParent(cameraHolder);

    Cursor.lockState = CursorLockMode.Locked;
    Cursor.visible = false;
}

void Update()
{
    // Виконувати лише для власного гравця
    if (!photonView.IsMine) return;

    HandleMovement();
    HandleLook();
    HandlePickup();
}

// ── Рух ──────────────────────────────────────────────────────────────
void HandleMovement()
{
    bool grounded = _cc.isGrounded;
    if (grounded && _velocity.y < 0) _velocity.y = -2f;

    float h = Input.GetAxis("Horizontal");
    float v = Input.GetAxis("Vertical");
    Vector3 move = transform.right * h + transform.forward * v;
    _cc.Move(move * moveSpeed * Time.deltaTime);

    if (Input.GetButtonDown("Jump") && grounded)
        _velocity.y = Mathf.Sqrt(jumpForce * -2f * gravity);

    _velocity.y += gravity * Time.deltaTime;
```

```
        _cc.Move(_velocity * Time.deltaTime);
    }

    // ── Огляд мишею ──────────────────────────────────────────────
    void HandleLook()
    {
        float mx = Input.GetAxis("Mouse X") * mouseSensitivity;
        float my = Input.GetAxis("Mouse Y") * mouseSensitivity;

        _xRotation -= my;
        _xRotation = Mathf.Clamp(_xRotation, -80f, 80f);

        if (cameraHolder != null)
            cameraHolder.localRotation = Quaternion.Euler(_xRotation, 0f, 0f);

        transform.Rotate(Vector3.up * mx);
    }

    // ── Підбір предметів ─────────────────────────────────────────
    void HandlePickup()
    {
        if (!Input.GetKeyDown(KeyCode.E)) return;

        Ray ray = new Ray(cameraHolder.position, cameraHolder.forward);
        if (Physics.Raycast(ray, out RaycastHit hit, pickupRange, itemLayer))
        {
            var item = hit.collider.GetComponent<CollectibleItem>();
            if (item != null)
            {
                _score += item.points;
                _ui?.UpdateScore(_score);
                PhotonNetwork.Destroy(hit.collider.gameObject);
            }
        }
    }
}
```

PlayerNameTag.cs:

```
using UnityEngine;
using UnityEngine.UI;
using Photon.Pun;

/// <summary>
/// Показує нік гравця над персонажем (Billboard — завжди повернений до камери).
/// Прикріпити до World Space Canvas над головою гравця.
/// </summary>
public class PlayerNameTag : MonoBehaviourPun
{
    public Text nameText;
    Transform _cam;
```

```
    void Start()
    {
        _cam = Camera.main?.transform;
        if (nameText != null)
            nameText.text = photonView.Owner?.NickName ?? "???";

        // Не показувати власний нік собі (опційно)
        if (photonView.IsMine)
            gameObject.SetActive(false);
    }

    void LateUpdate()
    {
        // Billboard: повернути до камери
        if (_cam != null)
            transform.LookAt(transform.position + _cam.forward);
    }
}
```

PlayerUI.cs:

```
using UnityEngine;
using UnityEngine.UI;
using Photon.Pun;

/// <summary>
/// UI гравця — рахунок, таймер, підказки.
/// Прикріпити до того ж GameObject, що й PlayerController.
/// </summary>
public class PlayerUI : MonoBehaviourPun
{
    [Header("UI елементи (призначити в Inspector)")]
    public Text scoreText;
    public Text timerText;
    public Text hintText;
    public Text playerNameText;

    GameManager _gm;

    void Start()
    {
        // UI бачить лише свій гравець
        if (!photonView.IsMine)
        {
            // Вимкнути Canvas у інших гравців (не потрібен їм локально)
            var canvas = GetComponentInChildren<Canvas>();
            if (canvas != null) canvas.enabled = false;
            return;
        }
```

```
        _gm = FindObjectOfType<GameManager>();

        if (playerNameText != null)
            playerNameText.text = PhotonNetwork.NickName;

        if (hintText != null)
            hintText.text = "WASD — рух   |   Пробіл — стрибок   |   E — підібрати
предмет";

        UpdateScore(0);
    }

    void Update()
    {
        if (!photonView.IsMine || _gm == null) return;

        if (timerText != null)
        {
            float t = _gm.GetTimeLeft();
            int min = Mathf.FloorToInt(t / 60);
            int sec = Mathf.FloorToInt(t % 60);
            timerText.text = $"Час: {min:00}:{sec:00}";
        }
    }

    // Викликається з PlayerController після підбору предмету
    public void UpdateScore(int score)
    {
        if (scoreText != null)
            scoreText.text = $"Очки: {score}";
    }
}
```