Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Мультиплеєрна гра жанру «шутер» на рушії Unity

(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-1
Данило РЕЗНІКОВ
(власне ім’я, прізвище)

Спеціальність
123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Комп’ютерна інженерія
(повна назва освітньої програми)

Керівник: ас. Єгор КОРНІЄНКО
(посада, власне ім’я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп’ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Резнікову Данилу Олександровичу
(прізвище, ім’я, по батькові)

1. Тема роботи Мультиплеєрна гра жанру «шутер» на рушії Unity

затверджена наказом по університету від “ 26 ” травня 2025 р. № 424 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 17 червня 2025 р.

3. Вхідні дані до роботи 1) ігровий рушій Unity; 2) мережевий фреймворк: Photon Fusion; 3) протоколи передачі даних: UDP для руху та RPC; 4) ігрові системи.

4. Перелік питань, що потрібно опрацювати у роботі

1) аналіз вимог до мережевого шутера та огляд існуючих рішень;
2) проектування архітектури мережевої взаємодії;
3) реалізація ігрових механік у мережі (рух, стрільба, стан гравця, чат);
4) інтеграція сервісів (Firebase Auth + Photon Fusion);
5) налагодження та логування мережевих подій;
6) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 16 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз вимог і огляд існуючих рішень	20.04.25-26.04.25	
2	Вибір технологій та інструментів	28.05.25-02.06.25	
3	Розробка алгоритмічного забезпечення	03.06.25-05.06.25	
4	Розробка та відлагодження програмних модулів	05.06.25-07.06.25	
5	Оформлення матеріалів кваліфікаційної роботи	07.06.25-08.06.25	
6	Подання кваліфікаційної роботи керівникові та	09.06.25-13.06.25	
	її попередній захист		
7	Подання кваліфікаційної роботи на рецензу-	14.06.25-16.06.25	
	вання		

Дата видачі завдання " 26 " травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ас. Єгор КОРНІЄНКО
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 95 с., 14 рис., 0 табл., 1 дод., 21 джерел.

UNITY, RPC, СИНХРОНІЗАЦІЯ, ІНТЕРПОЛЯЦІЯ, ЕКСТРАПОЛЯЦІЯ, ІГРОВІ СИСТЕМИ, UDP, FIREBASE, PHOTON FUSION 2.

Метою кваліфікаційної роботи є розробка прототипу багатокористувацької гри в жанрі мережевого шутера з використанням ігрового рушія Unity та мережевого фреймворку Photon Fusion 2, із підтримкою базової синхронізації гравців, системи бою, матчмейкінгу та інтеграцією аутентифікації через Firebase.

У ході виконання кваліфікаційної роботи було проведено огляд сучасних рішень у сфері реалізації мультиплеєрних ігор, обґрунтовано вибір технологій та архітектури проекту. Розроблено основні мережеві компоненти: рух, стрільба, стан гравця, HUD, лобі та чат. Реалізовано систему інтерполяції та екстраполяції для згладжування рухів, а також RPC-виклики для обміну подіями. Проведено базове тестування роботи гри в умовах мережевих затримок. Робота містить програмну реалізацію та інструкцію користувача.

ABSTRACT

Bachelor’s thesis: 95 pages, 14 figures, 0 tables, 1 appendices, 21 sources.

UNITY, RPC, SYNCHRONIZATION, INTERPOLATION, EXTRAPOLATION, GAME SYSTEMS, UDP, FIREBASE, PHOTON FUSION 2.

The major goal of this thesis is to develop a prototype of a multiplayer first-person shooter game using the Unity engine and Photon Fusion 2 networking framework, with integrated Firebase authentication and core gameplay systems.

In order to achieve this goal, a comparative analysis of multiplayer networking technologies was conducted. The Photon Fusion framework was chosen based on its performance, scalability, and flexibility. The developed application includes player movement and shooting synchronization, health and respawn systems, matchmaking, in-game chat, and user interface. Interpolation and extrapolation algorithms were implemented to smooth network movement. Testing was carried out under various network conditions to evaluate latency and reliability. The result is a functional multiplayer prototype suitable for further development and optimization.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API – Application Programming Interface (інтерфейс прикладного програмування)

HUD – Head-Up Display (екранна інформаційна панель)

ID – Identifier (ідентифікатор)

JSON – JavaScript Object Notation (формат обміну даними)

RPC – Remote Procedure Call (виклик віддаленої процедури)

UDP – User Datagram Protocol (протокол дейтаграм користувача)

UI – User Interface (інтерфейс користувача)

ВСТУП

У сучасному цифровому світі відеоігри займають важливе місце в індустрії розваг, а багатокористувацькі онлайн-шутери, зокрема FPS (First-Person Shooter) і TPS (Third-Person Shooter), залишаються одними з найпопулярніших жанрів. Їхня популярність пояснюється не лише динамічним геймплеєм та змагальним духом, а й можливістю взаємодії з іншими гравцями в реальному часі, що створює унікальний інтерактивний досвід. Стрімке зростання популярності кіберспорту та вдосконалення інтернет-інфраструктури формують високі очікування користувачів щодо якості онлайн-гри. Гравці прагнуть мінімальних затримок, точної синхронізації між клієнтами й сервером, а також надійного захисту від нечесної гри (cheating), що потребує використання сучасних технологічних рішень.

Враховуючи ці вимоги, особливої актуальності набуває розробка ефективної мережевої архітектури, яка б забезпечувала стабільну, безпечну та масштабовану взаємодію в умовах реального часу. Ігрові рушії, зокрема Unity, у поєднанні з потужними мережевими бібліотеками, такими як Photon Fusion 2, дають змогу реалізовувати сучасні підходи до розробки мережевих ігор. Застосування механізмів лаг-компенсації, інтерполяції та екстраполяції рухів дає змогу згладжувати проблеми з'єднання й покращувати ігровий досвід навіть у складних мережевих умовах.

Таким чином, метою даної роботи є дослідження архітектурно-технологічних рішень для розробки багатокористувацької гри шутерного типу з використанням Unity та Photon Fusion 2. Особлива увага приділяється реалізації мережевої взаємодії, базових ігрових систем, а також впровадженню підходів, що забезпечують якісний, стабільний та зручний для користувача ігровий процес.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Вступ до мережевих шутерів

1.1.1 Еволюція та виклики мультиплеєрних шутерів: історичний та технічний аналіз

Перші спроби шутерів від першої особи з можливістю взаємодії двох гравців з’явилися ще на мейнфреймах у 1973 році, коли Стів Коллі та колеги створили «Maze War» (рисунок 1.1) з підтримкою двох гравців і «стрільбою» між собою. У 1992 році гра Wolfenstein 3D заклала основи жанру, але справжній прорив у мультиплеєрі приніс Doom (1993) із режимом deathmatch по локальній мережі. Поява GoldenEye 007 (1997) на Nintendo 64 вперше зробила чотири гравці в одному матчі стандартом консольного мультиплеєру у шутерах, а Quake III: Arena (1999) довела концепцію чисто мережевої арени.

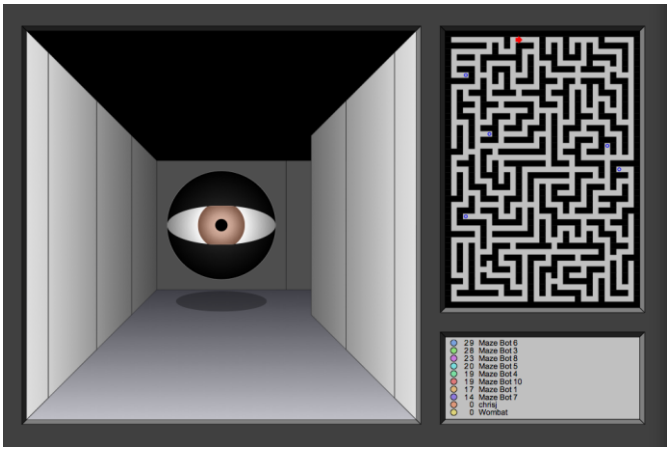

Рисунок 1.1 – Гра «Maze War»

У 2000-х роках з підключенням до Інтернету гру Counter-Strike (2000) оцінили мільйони гравців, а з виходом Battlefield 1942, Halo 2 та Call of Duty 4: Modern Warfare жанр FPS/TPS перетворився на глобальний феномен.

Сьогодні найбільші проєкти, як-от Counter-Strike 2, PUBG: Battlegrounds і Apex Legends, збирають одночасно понад мільйон гравців на платформах Steam та консолях, причому загальна кількість активних геймерів у світі перевищує 3,3 млрд осіб. Жанр FPS займає близько 38 % ринку відеоігор за обсягом аудиторії, а мультиплеєрні режими становлять понад 75 % від усього онлайн-трафіку ігрових серверів. [1]

1.1.2 Аналіз сучасного ринку мультиплеєрних шутерів

Сучасний ринок мультиплеєрних шутерів є висококонкурентним, із провідними проектами, які задають стандарти якості, технологій і геймплею. Для обґрунтування актуальності розробки нової концепції багатокористувацького шутера необхідно оцінити ключових конкурентів на ринку FPS/TPS-шутерів, їхні технічні особливості, сильні та слабкі сторони, щоб виявити прогалини, які може заповнити пропоноване рішення. Аналіз ринку дозволяє визначити вимоги до сучасних мультиплеєрних ігор, такі як низька затримка, ефективна синхронізація, захист від шахрайства та підтримка кросплатформності. За даними аналітичних звітів 2024 року, загальна кількість активних геймерів у світі перевищує 3,3 мільярда осіб, із яких близько 38 % віддають перевагу жанру FPS, а мультиплеєрні режими становлять понад 75 % онлайн-трафіку ігрових серверів. Розглянемо основних представників ринку: Counter-Strike 2, Apex Legends, PUBG: Battlegrounds і Valorant.

Counter-Strike 2 від Valve є еталоном тактичних шутерів, побудованим на власному рушії Source 2 із клієнт-серверною архітектурою. Гра використовує високий tick rate серверів, що забезпечує мінімальну затримку, і розвинену систему лаг-компенсації, яка гарантує плавний геймплей. Вона вирізняється стабільністю серверів завдяки оптимізованій інфраструктурі Valve, ефективною системою античиту VAC, яка підтримує кіберспортивну екосистему, і високою якістю синхронізації дій гравців, що ідеально

підходить для тактичних матчів. Проте закритий код рушія Source 2 обмежує можливості кастомізації для сторонніх розробників, високі вимоги до серверної інфраструктури ускладнюють масштабування для невеликих студій, а фокус на конкурентному геймплеї обмежує підтримку неформальних режимів.

Apex Legends від Respawn Entertainment – це батл-рояль-шутер, який базується на модифікованому рушії Source і хмарній інфраструктурі, що дозволяє підтримувати матчі з шістдесятьма гравцями. Гра відома вдосконаленою системою лаг-компенсації та передбачення рухів, що забезпечує плавний ігровий досвід, підтримкою кросплатформної гри між ПК і консолями, а також високоякісним матчмейкінгом і системою лобі. Однак висока залежність від хмарних серверів підвищує витрати на підтримку, складність інтеграції нових механік через пропрієтарний рушій обмежує гнучкість, а підтримка моддингу чи створення користувацьких серверів є обмеженою.

PUBG: Battlegrounds від Krafton використовує Unreal Engine і підтримує масштабні матчі до ста гравців із клієнт-серверною моделлю. Гра вирізняється можливістю масштабування для великих матчів із підтримкою розподілених серверів, гнучкою системою кастомізації зброї та геймплею, а також широкою підтримкою кросплатформності. Проте періодичні проблеми із затримкою та десинхронізацією через велику кількість гравців, складність оптимізації для слабших систем через високі вимоги Unreal Engine і недостатньо ефективна система захисту від шахрайства порівняно з конкурентами є її слабкими сторонами.

Valorant від Riot Games базується на Unreal Engine із клієнт-серверною архітектурою та власною системою античиту Vanguard. Гра пропонує високий tick rate серверів, що забезпечує мінімальну затримку, розвинену систему проти нечесної гри, яка ефективно протидіє зловмисникам, і оптимізацію для широкого спектра апаратного забезпечення. Підтримка великих матчів обмежена форматом п’ять на п’ять, а високі вимоги до

серверної інфраструктури ускладнюють стабільну роботу.

Аналіз сучасного ринку мультиплеєрних шутерів показує, що провідні проекти, такі як Counter-Strike 2, Apex Legends, PUBG і Valorant, встановлюють високі стандарти якості завдяки розвиненим системам синхронізації, античиту та матчмейкінгу. Однак ці ігри мають спільні обмеження: залежність від пропрієтарних рушіїв, таких як Source чи Unreal Engine, або хмарних інфраструктур, що ускладнює доступ для невеликих студій чи незалежних розробників, обмежена гнучкість у кастомізації мережевих протоколів і серверних рішень, а також проблеми з масштабуванням для великих матчів або слабших систем у деяких проектах. Ці недоліки створюють нішу для розробки нової концепції багатокористувацького шутера, яка використовує доступні інструменти, такі як Unity і Photon Fusion 2, для створення гнучкої, масштабованої та економічно вигідної гри. Актуальність дослідження зумовлена необхідністю створення рішення, яке поєднує передові механізми лаг-компенсації, передбачення рухів і захисту від шахрайства, доступні для реалізації в Unity, і водночас долає обмеження пропрієтарних систем, характерних для лідерів ринку. [1]

1.1.3 Основні виклики мультиплеєрних шутерів

Низька затримка є критичною для конкурентних шутерів: значення понад 100 мс вже помітно погіршують реакцію гравця, тоді як для комфортної гри прагнуть утримувати пінг нижче 50 мс. Окрім відстані до серверів і пропускної здатності мережі, важливу роль відіграють внутрішні параметри гри – tickrate сервера і методи компенсації лагу (interpolation/extrapolation). Unity-документація рекомендує використовувати алгоритми передиктивного руху й базовану на сервері перевірку попадань, щоб приховати спорадичні спайки затримки. З розвитком голосового чату й вбудованих соціальних функцій поширеність нечесної гри, DDoS-атак на

гравців і токсичної поведінки зросла. Дослідження від Offenburg University у Valorant і Overwatch виявило понад 80 % матчів із загальними образами та 14 % із сексуальними домаганнями. Серверні авторитетні моделі та механізми боротьби з шахрайством на основі валідації на стороні сервера дозволяють суттєво знизити кількість шахрайських дій.

Відкриті арени й широкий вибір зброї створюють складну задачу балансування: потрібно впевнитися, що кожна одиниця озброєння має унікальні переваги та недоліки, а комбінації карт і режимів не дають несправедливої переваги одній зі сторін. Регулярні патчі й аналітика телеметрії (kill-death ratio, win rates) допомагають розробникам оперативно коригувати параметри. [1]

1.2 Дослідження засобів реалізації програмної архітектури

1.2.1 Facade

Патерн Facade є ключовим інструментом в архітектурі великих програмних систем, особливо коли йдеться про взаємодію багатьох незалежних компонентів, які спільно реалізують складну поведінку. У своїй суті Facade виступає як посередник, який інкапсулює складну логіку взаємодії кількох об’єктів і надає зовнішнім користувачам або модулям єдиний, спрощений інтерфейс. Це дозволяє зменшити залежність між підсистемами та зробити систему більш зрозумілою, гнучкою й такою, що легше масштабувати. Facade часто використовується там, де внутрішня реалізація включає безліч кроків ініціалізації, викликів у певному порядку або залежностей, які не повинні бути відомі або доступні зовнішнім об’єктам. [2]

У нашому проекті патерн Facade ефективно вписується в структуру класу гравця. В даному випадку, кожен гравець має набір пов’язаних між собою систем: це можуть бути модулі керування переміщенням, стрільбою,

відтворенням анімацій, обробкою отриманих ушкоджень, відображенням UI, взаємодією з мережею через RPC, а також багатьма іншими внутрішніми службами, такими як відтворення ефектів або ведення статистики. Усі ці компоненти є незалежними в технічному сенсі, але повинні працювати синхронно для забезпечення цілісної поведінки персонажа. Реалізація фасаду дозволяє централізовано керувати цими системами – він створює їх під час ініціалізації, встановлює між ними необхідні зв’язки, конфігурує параметри і координує їхню взаємодію під час гри. У структурі ігрової архітектури фасад виконує роль проміжного шару, який забезпечує спрощений доступ до внутрішніх підсистем. Він абстрагує взаємодію зовнішніх об'єктів гри з внутрішньою логікою, дозволяючи звертатися до комплексної функціональності без необхідності знати, які саме модулі відповідають за її реалізацію. Це сприяє гнучкості: зміни в окремих підсистемах, як-от система стрільби (додавання типів зброї, ефектів, логіки перезарядки), можуть виконуватися без впливу на код, що працює з фасадом. Ще одна перевага такого підходу – можливість легко керувати ресурсами та оптимізувати використання пам’яті й обчислювальних потужностей. Наприклад, фасад може контролювати момент створення та знищення об’єктів, вмикати або вимикати певні системи залежно від активності гравця (наприклад, якщо гравець далеко від камери або тимчасово неактивний), а також спрощувати синхронізацію з мережею, оскільки вся комунікація з Fusion проходить через чітко визначені точки. Також фасад полегшує процес тестування та відлагодження. Наявність одного узагальненого інтерфейсу дозволяє створювати мок-об'єкти або спеціальні тести, які симулюють поведінку гравця без необхідності запускати всю гру.

Це суттєво знижує час на перевірку окремих сценаріїв і сприяє побудові стабільної й масштабованої кодової бази. Крім того, з точки зору співпраці в команді, використання фасаду в структурі гравця дозволяє розділити обов’язки між розробниками: один може працювати над внутрішніми системами, а інший – інтегрувати ці системи через фасад без

конфліктів і дублювання логіки.

1.2.2 Observer

Патерн Observer (спостерігач) є фундаментальним поведінковим шаблоном, що дозволяє одному об’єкту (суб’єкту) повідомляти інші об’єкти (спостерігачів) про зміну свого стану без необхідності знати щось про їхню реалізацію. Це досягається через механізм підписки, де спостерігачі реєструються на події й автоматично реагують на їх виникнення. У рамках ігрової архітектури це особливо важливо, коли потрібно зберігати синхронність між ігровими об’єктами, інтерфейсом, мережею та внутрішніми станами системи.

Цей патерн є надзвичайно корисним для реалізації реактивної поведінки, яка автоматично оновлює пов’язані елементи гри, щойно змінюється певний стан. Наприклад, коли гравець отримує ушкодження, система здоров’я змінює своє значення, і ця зміна автоматично поширюється на всі підписані компоненти: UI-індикатори здоров’я, анімаційний контролер, логіку смерті, логіку повідомлення по мережі та звукові ефекти. Тобто замість того, щоб вручну викликати методи оновлення в кожному компоненті, усі ці системи вже підписані на подію, яка відбувається в модулі здоров’я, і реагують на неї самостійно. [3]

У мережевому аспекті застосування цього патерна сприяє ефективній організації обміну даними. Наприклад, коли стан гравця змінюється на сервері (він перезаряджає зброю, перемикає режим стрільби або змінює позу), ця подія може бути синхронізована через Fusion, і відповідний локальний об'єкт сповіщає про це всі підписані на нього системи. Це дозволяє мінімізувати мережевий трафік і централізувати логіку реагування: достатньо один раз змінити стан, і всі, кому це потрібно (UI, анімації, звук), будуть автоматично повідомлені. У цьому проекті це може бути реалізовано як частина системи подій або делегатів. Наприклад, коли клас гравця має

подію OnHealthChanged, її може слухати HealthBarUI, PlayerAnimator, або навіть мережевий компонент, який вирішує, чи потрібно транслювати цю зміну по мережі. Усі ці залежності не пов’язані напряму між собою, що дозволяє змінювати, видаляти або додавати нові системи без ризику зламати інші частини гри. Такий підхід підтримує відкритість до розширення і закритість до модифікацій, згідно з принципами SOLID.

Таким чином, патерн Observer забезпечує ефективну та масштабовану архітектуру реакції на зміни стану, як у локальному, так і в мережевому контексті. Це робить систему більш динамічною, чіткою в розділенні обов’язків та зручною для подальшого розвитку і підтримки.

1.2.3 Єдина точка входу

У контексті архітектури програмного забезпечення, зокрема ігрових проектів, патерн Entry Point (точка входу) виконує роль централізованого місця, з якого починається ініціалізація всієї системи. Це не лише стартовий клас чи сцена, а повноцінна архітектурна концепція, яка дозволяє визначити порядок запуску критичних сервісів, конфігурацій, менеджерів і систем ще до початку активної взаємодії користувача з грою. Такий підхід забезпечує контроль над порядком ініціалізації, дозволяє уникнути помилок залежностей та забезпечує стабільну й передбачувану поведінку системи. [4]

У нашому випадку ця концепція є особливо важливою через складність самої структури гри. У таких іграх велика кількість систем повинні бути готові до взаємодії ще до того, як гравець потрапить у сам матч: це системи авторизації, підключення до сервера, завантаження профілю, створення сесії, конфігурації мережевого з'єднання, налаштування сцен і базових менеджерів (наприклад, аудіо, введення, логування або аналітики). Усе це має запускатися в суворому порядку.

Точка входу може бути реалізована у вигляді спеціальної сцени або класу, який завантажується першим (наприклад, Bootstrap, EntryPoint або

AppInitializer). У цьому класі виконується ініціалізація ключових сервісів гри, таких як менеджери мережевих систем, авторизація та реєстрація за допомогою баз даних, локальне збереження даних, менеджер налаштувань, системи UI та інші об'єкти, які мають існувати постійно протягом усієї сесії. Після успішного налаштування цей стартовий об'єкт виконує перехід до наступного стану – завантаження сцени меню або безпосередньо підключення до лобі гри. Завдяки централізованій точці входу можна уникнути ситуації, коли об'єкти в грі намагаються отримати доступ до ще неініціалізованих сервісів або створюють дублікати однотипних компонентів. Крім того, це дозволяє зручно керувати режимами запуску гри – наприклад, запуск у тестовому режимі без підключення к серверу, або ж запуск із дебаг-логами. Також це корисно для побудови систем діагностики, де ще до старту основної сцени можна зібрати інформацію про стан систем, перевірити наявність оновлень або автентифікувати користувача. Важливо, що Entry Point у мережевих іграх часто виступає і як точка ініціалізації сценаріїв підключення до мережі: вибір режиму (хост, клієнт, shared), створення/приєднання до сесій, попередня підготовка об’єктів синхронізації, ініціалізація шаблонів (prefab-ів) мережевих об’єктів тощо. У мережевих фреймворків ці аспекти винесені в спеціалізовані класи, але порядок, у якому все це конфігурується, залежить саме від центральної точки запуску. Відповідно, Entry Point також відповідає за реєстрацію сцен для мережевого менеджера, валідацію стану підключення, а також за коректний перехід між сценами згідно з логікою гри. Крім технічного контролю, така структура дозволяє значно підвищити читабельність і підтримуваність проєкту. Уся логіка запуску гри сконцентрована в одному місці, що дає змогу легко орієнтуватися в початковому потоці управління, спрощує пошук помилок і дає змогу швидко інтегрувати нові сервіси або змінювати послідовність завантаження. У великих іграх це критично важливо, особливо коли проєктом займається кілька розробників або команда.

Отже, у нашому FPS-шутері патерн Entry Point дозволяє організувати

запуск гри як чітку, контрольовану послідовність подій – від авторизації та підключення до серверу до ініціалізації ігрових систем і завантаження ігрових сцен. Це забезпечує гнучкість, стабільність і масштабованість архітектури, що є необхідним для складної мережевої гри з великою кількістю залежностей і станів.

1.3 Теоретичні основи мережевої синхронізації

1.3.1 Інтерполяція та екстраполяція руху

У багатокористувацьких відеоіграх однією з ключових технічних задач є забезпечення коректної мережевої синхронізації між клієнтами та сервером. Через затримки в передачі даних по мережі (так звану латентність), пакети з інформацією про положення гравців, їхні дії, обертання або стан гри можуть надходити із затримкою або бути втраченими. Це створює ризик того, що дії інших гравців на екрані користувача виглядатимуть неузгодженими, ривковими або зовсім некоректними. Щоб уникнути цих проблем і забезпечити плавне та реалістичне відображення гри на боці клієнта, використовуються такі методи, як інтерполяція та екстраполяція руху (рисунок 1.3).

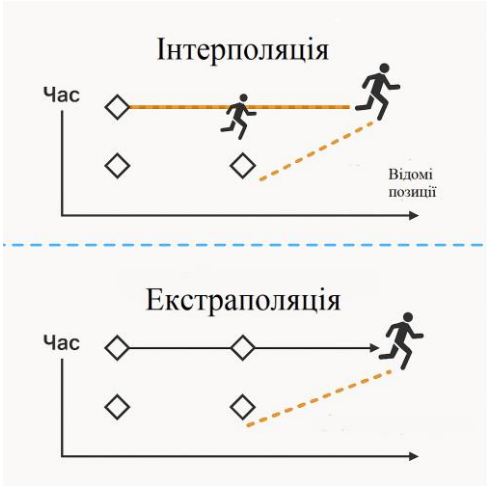

Рисунок 1.3 – Інтерполяція та екстраполяція руху

Інтерполяція ґрунтується на принципі згладжування руху об’єктів, коли клієнт, замість негайного відображення нової позиції об’єкта, обчислює проміжні положення між двома вже отриманими точками. Зазвичай це реалізується шляхом затримки візуалізації на декілька сотень мілісекунд (наприклад, 100-200 мс), щоб дочекатися кількох останніх оновлень від сервера. Клієнт зберігає отримані координати і плавно переміщує об'єкт між цими точками, створюючи враження безперервного і природного руху. Це дозволяє уникати візуальних "ривків", які можуть виникати через мережеві затримки, і підвищує загальну якість візуального сприйняття гри. [5] Інтерполяція працює лише з достовірною інформацією, яку вже отримано від сервера, тому вона точна, проте створює невелику візуальну затримку. Гравець насправді бачить положення об'єктів не в поточний момент часу, а в минулому – приблизно на ті ж самі 100-200 мс.

Екстраполяція, на відміну від інтерполяції, використовує наявну інформацію для передбачення майбутнього положення об’єкта. Коли нові дані від сервера ще не надійшли, клієнт на основі останньої відомої позиції, швидкості та напрямку руху самостійно розраховує, де приблизно повинен бути об’єкт у наступні кілька кадрів. Це дозволяє досягти більш реалістичного відображення гри в режимі реального часу, навіть при нестабільному з'єднанні. Однак екстраполяція має ризики. Якщо об'єкт раптово змінює напрямок або зупиняється, а клієнт продовжує передбачати його рух за старою траєкторією, може виникнути помилка, яка проявляється як стрибок або ривок, коли надходять актуальні координати і відбувається корекція. З цієї причини екстраполяція зазвичай використовується лише на короткі періоди – кілька кадрів або десятки мілісекунд.

У сучасних ігрових мережевих архітектурах зазвичай застосовується гібридний підхід, що поєднує обидві технології. За нормальних умов клієнт використовує інтерполяцію для забезпечення плавного руху, а в разі втрати пакетів, перевищення допустимого часу очікування або нестабільного пінгу – тимчасово вмикає екстраполяцію. Таке поєднання дозволяє досягти

компромісу між точністю та швидкістю реакції, зберігаючи при цьому якісне візуальне відображення всіх гравців у реальному часі. Ці методи є критично важливими для створення відчуття безперервності у грі та справедливості між гравцями, адже навіть кілька мілісекунд затримки можуть вплинути на результат бою, реакцію або прийняття рішень. Реалізація інтерполяції та екстраполяції потребує обчислювальних ресурсів і грамотної побудови логіки на боці клієнта, тому багато рушіїв надають інструменти або бібліотеки, які полегшують реалізацію цих механізмів у проектах будь-якої складності.

1.3.2 Лаг-компенсація

У багатокористувацьких іграх передача даних між клієнтом і сервером відбувається з певною затримкою, яка залежить від фізичної відстані, якості мережі, навантаження на сервер та інших чинників. Ця затримка, або лаг, може викликати розсинхронізацію між діями гравця на своєму екрані та тим, що відбувається в загальній ігровій реальності. Щоб мінімізувати вплив таких розбіжностей і забезпечити стабільний та передбачуваний геймплей, використовуються методи лаг-компенсації. Основні підходи до реалізації лаг-компенсації поділяються на клієнт-сторінні та сервер-сторінні стратегії, кожна з яких має свої переваги, недоліки та області застосування. [6][7]

Клієнтська лаг-компенсація полягає в тому, що гравець бачить свій світ «без затримки» – усі дії, які він виконує, наприклад, постріли, рух або взаємодію з об’єктами, відбуваються негайно на його екрані. Клієнт надсилає команду на сервер, і навіть якщо вона доходить із запізненням, сервер приймає її так, ніби вона була виконана у момент її відправки. Це означає, що сервер "відмотує" стан світу до того часу, коли гравець зробив дію, щоб перевірити, чи вона була коректною. Такий підхід часто використовується у швидких шутерах (наприклад, у Counter-Strike: Global Offensive або Call of Duty), де точність і час реакції мають вирішальне значення. Основною

перевагою цього методу є те, що гравець відчуває майже миттєву реакцію на свої дії, навіть при високому пінгу. Проте цей підхід створює ризик експлуатації: наприклад, гравці з високою затримкою можуть бачити позиції супротивників із запізненням, а стріляють – ніби в минуле, і це може дати їм несправедливу перевагу. Крім того, така система складна в реалізації, бо потребує збереження історії стану світу (replay buffer) на сервері для кожного клієнта.

Серверна лаг-компенсація, навпаки, базується на тому, що всі дії визначаються тільки з точки зору сервера, незалежно від того, коли вони були надіслані клієнтом. У цій моделі сервер обробляє лише ті події, які до нього надійшли, у тому вигляді та в той час, коли вони фактично прибули. Клієнт бачить гру з деякою затримкою, але це забезпечує максимальну чесність: гравці з нижчим пінгом мають перевагу в реакції, але система не піддається маніпуляціям. Такий підхід зазвичай застосовується в іграх, де важливішою є стабільність і цілісність стану гри, ніж абсолютна точність реакції (наприклад, у стратегічних іграх або MMO). Недоліком серверної лаг-компенсації є відчуття "запізнілих" дій: коли гравець стріляє, але ціль вже пересунулася, сервер не зараховує влучання – бо він отримав команду вже після переміщення цілі.

Існує також гібридний підхід, коли певні дії (наприклад, стрільба) обробляються із клієнтською компенсацією, а інші (наприклад, колізії чи пересування) – строго на сервері. Такі рішення намагаються поєднати відчуття швидкого реагування з високим рівнем чесності та передбачуваності гри.

1.3.3 Стиснення, пакетування та порядок доставлення повідомлень

У системах реального часу, зокрема в багатокористувацьких онлайн-іграх, ефективне управління мережею має вирішальне значення для стабільності, швидкодії та якості ігрового досвіду. Передача даних у таких

системах здійснюється за допомогою повідомлень, які повинні бути якомога легшими, доставленими у правильному порядку, та – у критичних випадках – гарантовано доставленими. Це досягається за рахунок трьох фундаментальних процесів: стиснення, пакетування та управління порядком доставляння повідомлень.

Стиснення повідомлень – це зменшення обсягу інформації, що передається мережею, без втрати критичних даних. Це дозволяє зменшити навантаження на канал зв’язку, знизити затримки та мінімізувати втрати пакетів при обмеженій пропускній здатності. У більшості випадків використовується спеціалізоване стиснення, орієнтоване на ігрові дані: наприклад, позиції об’єктів, кути повороту, стани анімацій або дії гравця. Значення, які зазвичай подаються як 32-бітні або 64-бітні числа, можуть бути представлені менш точними, але набагато компактнішими форматами. Замість того, щоб передавати абсолютне значення кожного разу, часто передаються дельта-значення (зміни з останнього стану), що різко скорочує обсяг інформації. Для деяких типів даних можуть використовуватись простіші методи, такі як бітові прапори, об’єднання кількох логічних значень у байт або навіть кастомні схеми побітового кодування. Важливим аспектом є компроміс між точністю даних і ефективністю стиснення: надмірне спрощення може викликати візуальні або геймплейні артефакти, натомість надто «важкі» пакети збільшують затримку й трафік. Наступним рівнем оптимізації є пакетування. В реальному середовищі мережеві протоколи мають певні обмеження щодо кількості повідомлень, які можна відправити за одиницю часу.

Щоб зменшити частоту звернень до мережі та мінімізувати системні витрати на обробку кожного пакету (зокрема, створення заголовків, маршрутизація, черги на відправлення), кілька повідомлень можуть бути об'єднані в один пакет.

Це дозволяє більш ефективно використовувати канал передачі та уникати надмірної фрагментації.

Наприклад, оновлення позицій кількох об’єктів або серія ігрових подій (вистріл, перезарядка, відкриття дверей) можуть бути упаковані в одну порцію даних. Водночас при пакетуванні необхідно враховувати часову чутливість деяких повідомлень.

Наприклад, затримка пакета лише на кілька мілісекунд може зробити деякі дані застарілими. Тому системи часто використовують мікропакетування – тобто об’єднують лише ті повідомлення, які логічно можуть чекати одне одного, не порушуючи цілісності геймплею. Додатково можуть бути застосовані алгоритми маркування важливості чи пріоритетності повідомлень, що дає змогу системі динамічно обирати, які дані пакетувати, а які – відправляти окремо. Контроль порядку доставляння – це ще один важливий аспект у системі передачі повідомлень, оскільки IP-протоколи самі по собі не гарантують ні порядок, ні навіть факт доставки пакетів. Для деяких типів даних (наприклад, події натискання кнопки, початок анімації, постріл) критично важливо, щоб вони були отримані в тому порядку, в якому були надіслані. Інакше гра може неправильно інтерпретувати ситуацію, створюючи візуальні й логічні збої. У таких випадках застосовуються номери послідовності, які дозволяють приймаючій стороні перевіряти, чи не пропущено жодного повідомлення та чи немає помилок у порядку. Повідомлення, що приходять із запізненням або у неправильному порядку, можуть бути проігноровані, поставлені в чергу або повторно відтворені в потрібній послідовності. Для ще більшої надійності часто використовуються механізми підтвердження доставки (ACK/NACK), таймери очікування та повторні спроби відправлення. Проте такий підхід може створювати додаткову затримку, тому використовується тільки там, де це дійсно потрібно. Для менш критичних даних, наприклад, для частих оновлень позицій гравця, порядок доставки зазвичай не гарантується – останнє актуальне повідомлення просто замінює попереднє, навіть якщо деякі кадри було втрачено. Це дозволяє досягти високої швидкодії за рахунок допустимої втрати точності. Комбінування цих трьох механізмів – стиснення,

пакетування та управління порядком – створює гнучку й ефективну архітектуру мережевого обміну. У складних ігрових середовищах, особливо при великій кількості одночасних користувачів, від правильності реалізації цих механізмів залежить не лише комфорт гравця, а й функціональна коректність гри загалом. Завдяки оптимізації трафіку вдається забезпечити більш швидку реакцію, знизити навантаження на сервер і клієнти, уникнути перевантаження мережі та мінімізувати ризик розсинхронізації ігрового стану. У результаті створюється основа для стабільного, чесного та масштабованого багатокористувацького досвіду. [7]

1.4 Огляд мережевих фреймворків

1.4.1 Photon Fusion 2

Одним із провідних рішень для створення високопродуктивних мережевих ігор є Photon Fusion 2 – сучасний мережевий рушій, розроблений компанією Exit Games як наступник Photon Bolt. Fusion спеціально створений для ігор у реальному часі з високими вимогами до швидкодії, точності синхронізації та масштабованості.

Його архітектура враховує потреби як невеликих інді-проектів, так і AAA-продуктів, що потребують оптимізації під масовий мультиплеєр. Photon Fusion 2 підтримує кілька мережевих режимів роботи, які дозволяють адаптувати архітектуру гри під різні потреби:

Host Mode – це класичний режим із локальним хостом. Один із клієнтів (зазвичай той, хто створив гру) виконує роль хоста, на якому обробляється вся логіка гри. Інші клієнти підключаються до цього хоста.

Перевага цього режиму – мінімальна затримка для хостуючого гравця, простота налаштування та менше навантаження на серверну інфраструктуру. Однак недоліком є те, що в разі виходу хоста з гри сесія переривається або потребує перенесення стану гри.

Shared Mode – усі клієнти взаємодіють із сервером через загальний авторитетний механізм. У цьому режимі логіка гри централізовано обробляється, і всі клієнти є підлеглими сервера. Це надає високий рівень безпеки від шахрайства, точну синхронізацію між клієнтами, стабільність та можливість масштабування. Shared mode ідеально підходить для матчмейкінгу, ігор на великі групи гравців і випадків, коли важлива централізована перевірка станів (наприклад, в кіберспортивних дисциплінах). [8] Photon Fusion 2 також вирізняється низкою переваг, які роблять його привабливим рішенням для розробників:

По-перше, він підтримує детерміновану симуляцію – можливість досягати ідентичного результату виконання логіки гри на різних клієнтах, що дозволяє ефективно реалізовувати rollback-механізми та лаг-компенсацію, особливо в жанрах з високою динамікою (шутери, файтинги, аркади).

По-друге, Fusion має інтегрований підтримку інтерполяції та екстраполяції, що спрощує синхронізацію руху, навіть при нестабільному з’єднанні. Усе це реалізується з використанням буферів і таймшрифтів (time-shifting), що дозволяє згладжувати відображення позицій об’єктів і підтримувати відчуття безперервного руху.

Третім важливим аспектом є висока продуктивність – Photon Fusion оптимізований під роботу з тисячами гравців у сесії (наприклад, у battle royale або MMO-проектах). Завдяки асинхронному обміну, підтримці delta compression, агрегуванню пакетів та мультиканальній передачі, Fusion здатен ефективно використовувати пропускну здатність і мінімізувати мережеві витрати.

Ще однією перевагою є платформна незалежність: Fusion підтримує роботу на всіх основних платформах – від ПК до мобільних пристроїв, VR/AR та консолей. [6][9]

Це дозволяє створювати кросплатформні проекти без значних додаткових витрат на адаптацію мережевої архітектури. Також слід зазначити наявність інструментів моніторингу та діагностики, які дають

змогу аналізувати якість з'єднання, виявляти втрати пакетів, відставання кадрів, перевантаження мережі тощо – усе це критично важливо для підтримки ігрового досвіду в стабільному стані.

1.4.2 Mirror

Mirror – це відкритий і високорівневий мережевий фреймворк для Unity, що успадковує і розвиває досвід UNET, акцентуючи увагу на простоті використання та гнучкості. У його основі лежить чітке розділення авторитету над станом гри (Server Authority) та клієнтської логіки, що гарантує узгодженість даних і мінімізує ризики розсинхронізації.

Mirror стабільно працює з Unity версій 2019-2022 і сумісний із декількома транспортними шарами (Telepathy, kcp2k, WebSockets тощо), що дозволяє розробникам обирати оптимальний протокол залежно від умов мережі та вимог проекту. Висока активність у репозиторії (понад 5 600 комітів за останній рік) свідчить про постійний розвиток і підтримку спільнотою. Однією з ключових можливостей Mirror є SyncVars – змінні класів, що успадковують NetworkBehaviour, які автоматично синхронізуються з сервера на клієнти. При підключенні нового гравця їм надсилаються останні значення всіх SyncVars видимих об’єктів, що спрощує підтримку цілісності стану світу. Цей механізм добре працює для зберігання будь-яких властивостей об’єкта – від позицій до складних ігрових параметрів.

Для передачі команд від клієнта до сервера та розповсюдження подій від сервера до клієнтів Mirror використовує Remote Procedure Calls: атрибути [Command] призначені для виклику методів на серверній стороні, а [ClientRpc] – дозволяють серверу ініціювати виконання функцій на всіх клієнтах. Такий підхід зрозумілий розробнику й водночас гнучкий у налаштуванні бізнес-логіки. Ще одним важливим компонентом є Interest Management – система відбору об’єктів «за зоною інтересу». Вона дозволяє

клієнту отримувати оновлення лише про ті об’єкти, що знаходяться поруч або відповідають певним критеріям (радіусна фільтрація, кастомні алгоритми), що підвищує масштабованість і знижує навантаження на мережу та процесор.

Mirror підтримує пакетування повідомлень (batching) і бітпакінг булевих прапорів, що оптимізує використання пропускної здатності. Розробники можуть також симулювати затримки та втрати пакетів безпосередньо в середовищі тестування, щоб оцінити стійкість ігрової логіки в реальних мережевих умовах. Для великих проєктів Mirror надає можливість запуску headless-серверів (без графічного інтерфейсу) як на локальних машинах, так і в хмарі, що полегшує автоматизацію розгортання та масштабування. Additive Scenes і Network Proximity Checker допомагають динамічно підвантажувати й відвантажувати частини світу, зменшуючи використання пам’яті й процесорного часу при роботі з великими картами.

Mirror ліцензовано за MIT і не вимагає роялті, що робить його доступним для комерційних та інді-проектів. Велика кількість прикладів, шаблонів і демонстраційних репозиторіїв на GitHub дозволяє розробникам швидко розпочати роботу і адаптувати готові рішення під власні потреби. Незважаючи на численні переваги, у Mirror є свої виклики. У нестандартних сценаріях може знадобитися ручне налаштування авторитету над об’єктами, а відсутність вбудованого хмарного хостингу вимагає підключення зовнішніх сервісів для діагностики та масштабування. Проте гнучкість API й активна спільнота роблять Mirror одним із найпопулярніших рішень для реалізації мультиплеєра в Unity. [7][8][10]

1.4.3 Netcode for GameObjects

Netcode for GameObjects (NGO) – це офіційний високорівневий SDK від Unity, який додає готові мережеві можливості до звичних робочих процесів із GameObject і MonoBehaviour, спираючись на підлеглий транспортний шар.

NGO призначений для проектів будь-якого масштабу: від простих кооперативних ігор до мережевих шутерів і MMO. У серці NGO лежить ідея чіткого розділення відповідальності: Server Authority із перевіркою всіх клієнтських команд на сервері, або Host Authority, коли один із учасників одночасно виконує роль сервера. Для синхронізації стану є два основні механізми: NetworkVariable (заміна SyncVar із старих систем) для автоматичної реплікації змін стану та RPC (Remote Procedure Calls) для передачі подій між клієнтами й сервером. Ці інструменти дозволяють розробнику гнучко визначати, які об’єкти та події мають пріоритет і якою мірою частота оновлень повинна бути оптимізована. [11]

Реплікування об’єктів відбувається через NetworkObject і пов’язаний із ним список префабів – усі мережеві префаби мають бути попередньо зареєстровані, щоб NGO міг автоматично створювати, знищувати та синхронізувати їхній стан під час гри. Кожен NetworkBehaviour може визначати власні NetworkVariable і RPC-методи ([ServerRpc], [ClientRpc]), що дає змогу чітко розділяти логіку: клієнтський запит на дію обробляється на сервері, а результати обчислень повертаються назад у вигляді RPC. Пакет NGO сумісний із Unity Transport та іншими транспортними шарами (наприклад, MLAPI Transport, LiteNetLib), що дозволяє вибирати оптимальний протокол (UDP, WebSocket тощо) залежно від потреб проєкту. NGO також підтримує Custom Messaging System для надсилання довільних бінарних або JSON-повідомлень, якщо вбудовані RPC/NetworkVariable не покривають усіх випадків.

Для оптимізації мережевого трафіку NGO пропонує aggregated batching (автоматичне об’єднання кількох оновлень у один пакет) та delta compression для NetworkVariable, що знижує обсяг передаваних даних. Система NetworkTransform і NetworkAnimator спрощують синхронізацію руху об’єктів і анімацій, при цьому реалізуючи інтерполяцію та предикцію на стороні клієнта. Особливістю NGO є інтеграція з Unity Gaming Services: Relay для P2P-з’єднань через хмарні сервери, Lobby і Multiplay для організації

матчмейкінгу та хостингу. Це дає змогу швидко налаштувати серверну інфраструктуру без власного DevOps, використовуючи єдиний Unity API. Netcode for GameObjects розвивається відкрито на GitHub під ліцензією MIT, має понад 2 200 зірок і активну спільноту, що регулярно доповнює приклади та виправляє баги. Документація на сайті Unity детально описує версію 2.x із новими можливостями, такими як Distributed Authority Network Topology для розподіленого управління об’єктами без єдиного хоста.

Netcode for GameObjects є універсальним рішенням для Unity-розробників, які шукають інструменти «із коробки» для швидкого створення мережевих ігор із гнучкою авторитетною моделлю, оптимізацією трафіку та інструментами інтеграції з хмарними сервісами. [7][9]

1.5 Висновки та постановка задачі

У процесі глибинного аналізу сучасних патернів і мережевих фреймворків для онлайн-шутерів було виявлено два ключові чинники, що мають найбільший вплив на якість ігрового досвіду. По-перше, високий пінг у різних географічних регіонах спричиняє лаги, нестабільність позиційних оновлень і видиму “телепортацію” гравців, що призводить до втрати відчуття точності та справедливості в перестрілках. По-друге, складність масштабування традиційних Peer-to-Peer чи хостованих сесій обмежує можливість збільшення кількості учасників у матчі без істотного зростання інфраструктурних витрат або підвищення ризику єдиної точки відмови.

Аналіз функціональності Photon Fusion 2 показав, що його Shared Mode із розподіленим авторитетом над об’єктами (State Authority) і вбудованими механізмами узгодження стану через хмарну кімнату ідеально відповідає виявленим вимогам. Shared Mode автоматично розподіляє відповідальність за різні об’єкти між клієнтами, усуваючи залежність від єдиного хоста та забезпечуючи безперервність сесії навіть під час виходу окремих учасників. Крім того, Fusion 2 пропонує вбудований Interest Management для

вибіркового розсилання оновлень та delta-compression для мінімізації трафіку, що додатково знижує затримки й розвантажує мережу. Проаналізувавши сучасні архітектурні й мережеві патерни, необхідно реалізувати в Unity онлайн-шутер на Photon Fusion 2 у режимі Shared Mode з централізованим узгодженням стану (State Authority), інтегрованими механізмами лаг-компенсації (інтерполяція, екстраполяція та rollback), а також базовими ігровими компонентами (рух персонажа, стрільба, динамічний HUD, чат). Проект повинен забезпечувати середню затримку ≤ 100 мс для 95 % гравців, підтримувати не менше ніж 200 одночасних підключень без втрати якості синхронізації та бути готовим до подальшого масштабування в хмарі.

2 ОБРАНА ПЛАТФОРМА, ТЕХНОЛОГІЇ ТА ІГРОВІ СИСТЕМИ

2.1 Ігровий рушій Unity

Unity – це один із найпопулярніших ігрових рушіїв, який широко використовується як у розробці інді-ігор, так і в проєктах комерційного рівня. Його гнучка архітектура, розширювана система компонентів і потужний інструментарій роблять його ідеальним вибором для створення як однокористувацьких, так і багатокористувацьких ігор. Основу архітектури Unity становить модель GameObject-Component, яка принципово відрізняється від класичних об’єктно-орієнтованих підходів, що використовуються в інших рушіях. У Unity вся сцена складається з об’єктів (GameObject), які самі по собі є лише порожніми контейнерами. Вся поведінка, зовнішній вигляд, фізичні властивості, логіка керування – усе це визначається за допомогою доданих до GameObject компонентів. [12]

Кожен компонент виконує конкретну функцію: наприклад, Transform відповідає за положення, обертання і масштаб; MeshRenderer – за відображення графіки; Collider – за взаємодію з фізикою; MonoBehaviour – за виконання користувацької логіки. Такий підхід дозволяє створювати складні об'єкти шляхом комбінації простих, незалежних модулів, забезпечуючи високу гнучкість, повторне використання і масштабованість коду. Unity сприяє створенню модульної архітектури, де замість жорстко пов’язаних класів розробник оперує легко замінними блоками, що значно спрощує налагодження та розширення гри.

Цикл кадру в Unity є фундаментальним для розуміння внутрішньої роботи рушія. Він визначає послідовність оновлення об'єктів, виконання логіки, обчислення фізики, оновлення анімацій та рендерингу сцени. Основні етапи цього циклу включають кілька ключових фаз: Awake і Start для початкової ініціалізації об'єктів, Update для обробки логіки, що виконується

щокадрово, FixedUpdate – для обробки фізичних процесів з фіксованим кроком часу, LateUpdate – для логіки, яка повинна бути виконана після Update (наприклад, позиціювання камери). Це забезпечує детерміновану та передбачувану послідовність дій, що критично важливо для мережевих ігор, де навіть мілісекунди можуть мати значення.

Ще однією важливою концепцією Unity є сцени. Сцена – це логічна одиниця, яка містить набір об'єктів та конфігурацій, що використовуються для певної частини гри. Наприклад, окрема сцена може містити головне меню, інша – ігрову арену, третя – екран налаштувань. Unity підтримує одночасне завантаження кількох сцен у режимі Additive, що особливо корисно для великих ігрових світів або модульних рівнів, де певні ділянки простору або UI можуть довантажуватись окремо, не впливаючи на основну логіку. Завдяки цьому можна ефективно організувати клієнт-серверну модель у мультиплеєрі, де базова сцена виконує роль платформи для мережевої логіки, а решта сцен довантажуються за потреби.

Unity надає великий набір інструментів, які роблять його привабливим вибором для створення мережевих ігор. Насамперед, рушій підтримує інтеграцію з низкою мережевих фреймворків – як вбудованих (Netcode for GameObjects), так і сторонніх (Photon, Mirror, т.д.), що дозволяє розробнику обрати найкраще рішення відповідно до вимог проєкту. Багато з цих рішень легко інтегруються з існуючою архітектурою Unity, дозволяючи використовувати знайомі патерни та компоненти. Наприклад, замість написання низькорівневого коду для синхронізації позицій гравців, достатньо прикріпити до GameObject спеціалізований мережевий компонент, який автоматично передає оновлення по мережі.

Серед ключових переваг Unity варто виділити візуальний редактор, який дозволяє проєктувати сцени та логіку в реальному часі, підтримку широкого спектра платформ (від Windows, Android, iOS до WebGL, PlayStation, Xbox), а також багатий екосистемний простір: Asset Store, офіційна документація, спільнота, відкриті плагіни й шаблони.

Інструменти профілювання, включно з Network Profiler, надають змогу аналізувати трафік, затримки, обсяг переданих даних та виявляти вузькі місця у продуктивності.

Особливої уваги заслуговує той факт, що Unity підтримує паралельне виконання процесів за допомогою Job System, Burst Compiler та Data-Oriented Technology Stack (DOTS). Це відкриває нові горизонти в обробці великої кількості об’єктів одночасно – критично важливо для масових мультиплеєрних ігор. Також завдяки Addressable Assets можливо реалізувати динамічну доставку контенту гравцям через CDN-сервери, зменшуючи розмір основного білду і час першого завантаження. Це важливо в умовах, коли потрібно постійно оновлювати гру або додавати новий контент без порушення доступу до серверу.

Таким чином, Unity – це не просто ігровий рушій, а гнучка платформа, що дозволяє будувати мультиплеєрні ігри з високим рівнем масштабованості, безпеки та продуктивності. Його модульна архітектура, розвинений цикл оновлення, підтримка сучасних мережевих фреймворків, оптимізаційні інструменти й активна спільнота перетворюють його на ефективний інструмент для розробки онлайн-шутерів, симуляторів, RPG або навіть масштабних MMO-проектів.

2.2 Photon Fusion 2: теоретичний розбір

2.2.1 Структура мережевого стеку: ролі NetworkRunner, принципи авторитету

У Photon Fusion 2 структура мережевого стеку побудована на основі чітко визначених ролей та централізованого підходу до обробки ігрової логіки, що дозволяє досягти точності синхронізації та масштабованості мережевої взаємодії. Ключовим елементом цього стеку є об’єкт NetworkRunner – це серце мережевого середовища, через яке здійснюється

весь контроль над сесією, створенням, оновленням та синхронізацією ігрових об’єктів. NetworkRunner ініціює сесію у відповідному режимі (Shared, Host, Server, Client), визначає топологію взаємодії між клієнтами та сервером і відповідає за основний цикл симуляції (simulation loop), який виконується з фіксованим інтервалом (tick rate). [7][8]

NetworkRunner виступає центральним диспетчером: він зчитує ввід гравця, передає його до серверної логіки, запускає симуляцію на authoritative стороні, синхронізує об’єкти між усіма учасниками, виконує лаг-компенсацію та інтерполяцію. У режимі Shared, який застосовується в цьому проєкті, кожен клієнт бере на себе обробку логіки та симуляції тих об’єктів, за які він має авторитет, тоді як загальний стан гри зберігається в “хмарній кімнаті” Photon Cloud і синхронізується між учасниками з використанням моделі «Eventual Consistency». Ця модель означає, що об’єкти після свого створення належать певному клієнту – саме він виконує всі оновлення їхнього стану локально і передає винятково зміни, а інші клієнти в свою чергу витягують і застосовують ці «дельти» в міру їх надходження. Всі події в Shared Mode прив’язуються до дискретних кроків симуляції (ticks), які відбуваються з фіксованою частотою, але клієнти не гарантують ідеального вирівнювання цих кроків між собою, отже синхронізація відбувається на рівні окремих змін стану, а не через глобальні знімки. [6][13]

Принцип авторитету (authority) в архітектурі Fusion полягає в тому, що лише певна сторона (зазвичай сервер) має право змінювати стан об’єкта. Наприклад, гравець може на своєму клієнті натиснути кнопку стрільби, але фактичне рішення про створення снаряда, його напрямок, колізії та ушкодження ухвалюється лише сервером. Це виключає можливість шахрайства, пов’язаного з маніпуляцією станом на стороні клієнта, і дозволяє централізовано перевіряти усі дії в грі.

Fusion також дозволяє вручну задавати авторитет для кожного NetworkObject, залежно від контексту: в одних випадках сервер може повністю контролювати об’єкт (наприклад, NPC або спільний ресурс), в

інших – клієнт може мати обмежений авторитет лише на час вводу. Така гнучкість дозволяє балансувати між продуктивністю, безпекою та відчуттям миттєвого керування з боку гравця.

2.2.2 Основні механізми синхронізації станів і RPC

У Photon Fusion 2 основні механізми синхронізації станів і віддалених викликів (RPC) побудовані навколо централізованого керування NetworkObject, автоматичного обміну змінними через мережу та спеціалізованої подієвої системи. Архітектура Fusion дозволяє досягати високої точності в оновленні ігрових об’єктів між усіма клієнтами, зберігаючи при цьому контроль за авторитетністю дій та оптимізацію мережевого трафіку. [7][9]

Синхронізація станів у Fusion реалізується через систему NetworkBehaviour – це похідний клас, який дозволяє позначати змінні як Networked, що робить їх частиною серіалізованого стану об’єкта. Змінні, позначені як Networked, автоматично відстежуються та серіалізуються у вигляді «дельт» (тільки зміни з попереднього стану), які потім надсилаються через мережу. Це дозволяє уникнути зайвих витрат пропускної здатності і гарантує, що всі клієнти бачать однаковий стан об’єкта. Fusion самостійно обирає найефективніший спосіб інтерполяції/екстраполяції, залежно від налаштувань та типу змінної (вектор, булеве значення, тощо), забезпечуючи плавну зміну положення об’єктів, їх стану та поведінки навіть за нестабільного з’єднання. [13]

Інший важливий механізм – Input Authority, який дає можливість клієнту вводити команди (наприклад, рух або стрільбу), не змінюючи стан об’єкта напряму. Замість цього Fusion збирає ввід, надсилає його до того учасника, який має авторитет (State Authority), де вже виконується симуляція дії та обчислення результату, що потім синхронізується з іншими. Це дає змогу будувати передбачувану архітектуру, де результат гри визначається

авторитетною логікою, навіть якщо різні клієнти отримують злегка різний ввід. Для обміну подіями між клієнтами та сервером у Fusion використовується система RPC (Remote Procedure Call). Це спеціалізований механізм, який дозволяє викликати методи на віддалених учасниках сесії. RPC в Fusion може бути одностороннім або двостороннім – виклик від клієнта до сервера, від сервера до клієнтів або широкомовний (broadcast). Методи, які будуть використовуватися як RPC, позначаються атрибутами ([Rpc], [Rpc(RpcSources.InputAuthority, RpcTargets.All)] тощо), де явно вказується, хто має право викликати метод і хто повинен його виконати.

Це дає змогу реалізовувати миттєві дії, як-от відтворення звуку, запуск анімації, спавн об’єктів або передача повідомлень. Важливою особливістю є те, що RPC не є засобом синхронізації стану – він лише виконує дію у певний момент часу. Тому для тривалих станів (наприклад, положення гравця, стан здоров’я, інвентар) використовують саме Networked змінні, а RPC лише ініціює зміну або реакцію.

Також Fusion дозволяє комбінувати RPC і Networked синхронізацію для складніших сценаріїв.

Наприклад, при натисканні кнопки клієнт відправляє ввід, авторитетна сторона змінює Networked змінну (наприклад, isShooting = true) та викликає RPC для запуску анімації або візуального ефекту на всіх клієнтах. Усе це відбувається в рамках одного симуляційного кроку (tick), що забезпечує точну відповідність між логікою та візуалізацією.

2.2.3 Порівняння режимів Host vs Shared з точки зору затримки та надійності

Порівняння режимів Host і Shared у Photon Fusion 2 з погляду затримки та надійності дозволяє краще зрозуміти, який із них доцільно обрати залежно від типу гри, вимог до стабільності та досвіду користувача. У Host Mode один із клієнтів (зазвичай той, хто створює гру) виступає в ролі авторитетного

хоста, який виконує всю симуляцію гри, обробку вводу, перевірку колізій, логіку дій тощо. Інші клієнти підключаються до нього як до сервера. Основною перевагою такого підходу є мінімальна затримка для хоста та захист від шахрайства клієнтів, які підключені до хоста. Він працює на локальній машині – тобто, відсутня мережа між логікою та гравцем. Для інших учасників затримка залежить від фізичної відстані та якості підключення до хостуючого клієнта. Це робить Host Mode ефективним для PvP-шутерів, MMO та інших проектів. [8][14] Проте Host Mode має серйозні недоліки в надійності. Якщо хост закриє гру або втратить з'єднання, сесія завершується для всіх. Жодна синхронізація або збереження стану не допоможе без спеціальних механізмів перенесення хоста.

Shared Mode побудований на такій архітектурі, де симуляція гри виконується локально на кожному клієнті. Кожен клієнт володіє своїм об'єктом і може змінювати його стан, і синхронізувати. У плані надійності, Host Mode значно перевершує Shared Mode. Серверна частина може бути розміщена на хмарній інфраструктурі, мати резервування та бути незалежною від поведінки клієнтів. Навіть при виході одного або кількох гравців сесія продовжує працювати.

Однак, Host Mode зазвичай має вищу середню затримку, ніж Shared Mode для клієнта, оскільки всі дії повинні проходити через сервер. Це потребує реалізації інтерполяції, екстраполяції, відновлення запізнілих даних і додаткових оптимізацій. Але завдяки цим же механізмам і передбачуваній архітектурі він дозволяє будувати стійку до навантажень та обману мультиплеєрну систему.

2.3 Авторизація за допомогою Firebase Authentication

Інтеграція Firebase Authentication дозволяє не лише автентифікувати користувачів, а й створює фундамент для персоналізації, безпеки та подальшої роботи з профілями гравців. Цей механізм забезпечує єдину

унікальну ідентифікацію кожного гравця за допомогою UID, що генерується Firebase після входу, незалежно від того, який тип автентифікації був використаний – анонімний, через Google, Apple, Facebook чи email-пароль. Таким чином, UID гравця можна використовувати як основний ключ у базі даних для зберігання його досягнень, статистики, скіна, рівня, внутрішньоігрової валюти тощо. [15]

З точки зору архітектури проєкту, авторизація в Firebase відбувається ще до підключення до ігрової сесії Photon Fusion. Наприклад, на початковому екрані гри гравець може авторизуватися, і лише після цього – отримати доступ до гри, меню пошуку матчів або системи лобі. Такий підхід дозволяє фільтрувати гравців: неавторизовані користувачі не отримають доступ до основного функціоналу, що підвищує безпеку і контроль над доступом.

Отриманий токен автентифікації Firebase може бути використаний не лише на клієнті, але й переданий до бекенду або навіть до Photon Cloud у вигляді кастомного поля аутентифікації. У Fusion, під час ініціалізації NetworkRunner, можна вказати користувача, що вже має певний UID, і таким чином пов’язати цей UID із мережею Photon. Це дозволяє відслідковувати гравця впродовж кількох сесій або навіть на різних пристроях, а також застосовувати обмеження (наприклад, бан за порушення).

Інтеграція Firebase також дозволяє створити систему збереження прогресу, синхронізовану з Firebase Realtime Database або Firestore. Наприклад, після кожної гри статистика гравця – кількість убивств, перемог, очки досвіду – може оновлюватися в базі даних під UID цього користувача. Це дозволяє створити повноцінну систему лідерів, яка працює незалежно від конкретної сесії або пристрою.

Для реалізації цієї функціональності достатньо використати Firebase SDK for Unity, який включає підтримку авторизації та баз даних. Окрім стандартних методів авторизації, Firebase дозволяє застосовувати мультифакторну автентифікацію (MFA), email verification, password reset, а також обмеження доступу через правила безпеки (Firebase Security Rules), що

особливо корисно при роботі з конфіденційними даними або особистою інформацією гравців. Ще однією перевагою використання Firebase у зв’язці з Photon Fusion є можливість реалізації кросплатформеності. Авторизація Firebase працює однаково як на Android, iOS, WebGL, так і в десктопних збірках, а UID користувача залишається незмінним. Це дозволяє одному гравцеві мати єдиний акаунт на всіх пристроях, з єдиним прогресом і доступом до персоналізованих налаштувань.

У контексті онлайн-шутера, де важливо мати стабільну систему входу та унікальну ідентифікацію кожного гравця для збереження прогресу, статистики, античітінгу або персоналізації, Firebase Authentication є одним із найпростіших і найефективніших рішень. Він зменшує потребу у власній серверній авторизаційній інфраструктурі, має масштабовану хмарну архітектуру, тісно інтегрується з іншими сервісами Firebase і добре працює у зв’язці з Photon Fusion. Таким чином, у проекті авторизація через Firebase не лише полегшує управління гравцями, а й створює основу для реалізації більш складних функцій – як-то система друзів, інвентар, монетизація, аналітика або push-сповіщення.

2.4 Теоретичний опис ключових ігрових систем

2.4.1 Система руху

Система руху є не просто технічним елементом, а основою геймплейного досвіду, що прямо впливає на відчуття керованості, плавності, реалізму та чесності взаємодії між гравцями. Зважаючи на те, що шутери передбачають високу динаміку, точність та швидкий зворотній зв’язок, рух персонажа повинен базуватись на надійних фізичних принципах та бути стійким до мережевих проблем, таких як затримки, втрати пакетів або нестабільне з’єднання. Фізична модель руху ґрунтується на класичних законах механіки. Основним рушієм є рівняння другого закону Ньютона.

Змінні у цьому закону такі: F – сила, що прикладається до об’єкта (наприклад, через введення користувача), m – маса об’єкта, a – прискорення. Це означає, що будь-який рух – результат дії сили, а не миттєве переміщення, як у примітивних системах позиціонування.

Стан руху персонажа описується через швидкість і прискорення. Якщо взяти вектор швидкості v, то його зміна залежить від часу дії сили.

Формули реалізуються через фізичні симуляції Unity (наприклад, Rigidbody), або вручну – за допомогою скриптової реалізації власного фізичного рушія. Для гравця це означає, що персонаж рухається поступово, з інерцією, з урахуванням тертя, гравітації або навмисно змодельованих сил опору. Раптове зупинення або поворот у протилежний бік відбувається не одразу, а з певною затримкою, що надає руху більшої реалістичності.

Окрему роль у цьому процесі відіграє гравітація. У фізичному сенсі вона визначається як сила.

Там g – прискорення вільного падіння (в Unity за замовчуванням 9.81 м/с^2). Гравітація застосовується до персонажа автоматично, якщо використовується компонент Rigidbody з активованою опцією "Use Gravity". У результаті, коли гравець стрибає або спускається з уступу, його рух виглядає природно – він прискорюється вниз під дією гравітації, аж поки не зіткнеться з поверхнею. [16]

Однак у мультиплеєрі рух ускладнюється необхідністю передавати й синхронізувати стан об’єкта між усіма учасниками гри в умовах обмеженої пропускної здатності й змінної затримки. Photon Fusion використовує модель авторитету (authority), де клієнт надсилає серверу лише інформацію про ввід (натискання клавіш), а сервер сам симулює рух згідно фізичних рівнянь. У режимі Shared, як обрано в цьому проєкті, сервер не є окремим хостом – кожен клієнт має обмежену авторитетність над власним об’єктом, але фінальні дані синхронізуються централізовано через хмарну інфраструктуру, що дозволяє досягти стабільності, низького пінгу та уникнути проблем із хост-міграцією.

Щоб уникнути «телепортацій» або ривків, система включає інтерполяцію: позиції, що надходять з мережі, не встановлюються миттєво, а плавно поєднуються із попередніми станами. Наприклад, поточне положення обчислюється як лінійна або кубічна інтерполяція між двома останніми отриманими координатами. Це дозволяє обійти візуальні артефакти, спричинені нерівномірною доставкою пакетів. Далі йде важливий механізм – прогнозування (client-side prediction). Гравець одразу бачить результат свого вводу, не чекаючи відповіді від сервера. Локально обчислений рух тимчасово показується гравцеві як припущення.

Коли сервер надсилає свою "офіційну" позицію, клієнт порівнює її з локальною симуляцією: якщо розбіжність незначна, корекція не потрібна; якщо ж вона перевищує допустимий поріг – застосовується корекція або навіть rewind (відкат до правильного стану), після чого дані вводів програються повторно.

Для врахування ударів, перешкод та взаємодій між гравцями застосовується механізм rollback: сервер «відмотує» стан гри на час надходження вводу від клієнта, перевіряє наявність колізій або влучань, і тільки тоді оновлює глобальний стан. Це особливо важливо у стрілецьких іграх – якщо гравець бачить ворога на певній позиції й стріляє, то сервер має точно відтворити цей момент, навіть якщо позиція противника вже встигла змінитись.

Також варто згадати про «snap to grid» або «reconciliation» – техніку корекції положення, яка м’яко "підтягує" локального персонажа до положення, визнаного правильним сервером. Це відбувається поступово, щоб уникнути раптових стрибків на екрані.

2.4.2 Система стрільби та зброї

У багатокористувацькому шутері система стрільби та зброї відіграє ключову роль у геймплеї, оскільки від її якості залежить не лише комфорт

гравця, а й загальне сприйняття бою, чесність взаємодії між користувачами та технічна стабільність. Архітектура цієї системи повинна бути достатньо гнучкою, щоб підтримувати як різноманітні типи зброї, так і різні стилі стрільби, а також здатною до масштабування та підтримки у складних мережевих умовах.

Існують два основних підходи до реалізації стрільби – hitscan та projectile-модель. У випадку з hitscan, постріл моделюється у вигляді миттєвого променя, який визначає попадання одразу в момент натискання кнопки. Це зручно для стрілецької зброї ближнього бою або для ігор, де критично важлива швидка реакція. Натомість projectile-модель передбачає фізичне існування кулі або снаряда, який рухається з певною швидкістю, має масу, може піддаватись впливу зовнішніх сил, таких як гравітація, і досягати цілі через деякий час. Такий підхід дозволяє досягти більшої реалістичності, наприклад, у зброї далекого бою, гранатометах або артилерії.

Щоб забезпечити модульність і зручність у підтримці, доцільно побудувати систему зброї на основі об'єктно-орієнтованої архітектури з використанням базових класів і наслідування. Базовий клас описує загальні властивості та методи, притаманні всій зброї: час перезарядки, ємність магазину, тип боєприпасів, швидкість стрільби, тип стрільби (одиночні постріли, автоматична стрільба тощо). Конкретні реалізації зброї наслідують цей базовий клас і додають унікальну поведінку – наприклад, у снайперської гвинтівки може бути затримка перед пострілом, у дробовика – кілька трасуючих частинок одночасно. Після натискання гравцем кнопки вогню, система спочатку формує дані пострілу: напрямок, положення, тип зброї, а також часову мітку дії. Ця інформація відправляється на сервер, який виконує лаг-компенсацію, тобто аналізує, де саме перебували гравці в момент пострілу з урахуванням затримки мережі. Сервер визначає, чи було влучення, і надсилає результат назад клієнтам. Такий підхід дозволяє запобігти шахрайству та забезпечити однакове відображення результатів для всіх учасників сесії. Попадання та завдання шкоди також обробляються на

сервері. Коли сервер підтверджує влучення, він оновлює стан здоров’я влученого об’єкта, ініціює візуальні ефекти – спалахи, частинки, звук – і синхронізує ці зміни з усіма клієнтами. Щоб надати гравцеві відчуття чуйності, деякі ефекти (наприклад, спалах пострілу або зворотній удар) можуть бути відтворені одразу на клієнті, ще до підтвердження від сервера, але лише як локальна візуальна реакція.

З архітектурної точки зору доцільно використовувати шаблони проектування, такі як "Стратегія", для делегування логіки пострілу залежно від типу зброї, або "Стан", для зміни режиму стрільби. Це спрощує додавання нових типів зброї або поведінки без необхідності змінювати вже написаний код. Система повинна бути масштабованою та адаптованою до потреб PvP- або PvE-ігрового процесу.

2.4.3 Система стану гравця: здоров’я/броня, механізми відновлення й респавну, управління життєвими циклами

Система стану гравця є фундаментальним компонентом, що визначає життєздатність персонажа, його реакцію на пошкодження, механізми відновлення, а також правила смерті й відродження. Ця система має бути побудована з урахуванням не лише геймплейної логіки, а й мережевої синхронізації, щоб забезпечити чесну взаємодію між усіма учасниками гри.

Основними параметрами стану гравця є здоров’я (health) та, за потреби, броня (armor). Здоров’я представляє основний показник витривалості персонажа – якщо воно падає до нуля, гравець вважається "мертвим". Броня ж виконує роль додаткового шару захисту, що поглинає частину шкоди. В різних ігрових підходах броня може зменшувати шкоду повністю або частково, може бути пробивною, регенеративною чи одноразовою. Механізм отримання шкоди базується на взаємодії з системою стрільби. Коли сервер фіксує попадання, він розраховує отриману шкоду та оновлює параметри стану гравця. Це означає, що саме сервер, як авторитетний вузол, відповідає

за зниження здоров’я, аби уникнути маніпуляцій з боку клієнтів. Після зміни стану, нові значення синхронізуються через мережу й відображаються на всіх клієнтах. [17]

Важливим елементом є механізми відновлення. Відновлення здоров’я може бути пасивним (автоматично з часом), активним (через використання аптечок, спеціальних здібностей), або залежати від командної взаємодії (наприклад, лікування союзником). Усі ці дії мають бути точно синхронізовані, адже зміна здоров’я впливає на бойову ефективність персонажа.

Респавн (відродження) – ще один критичний компонент. Після смерті гравець або переходить у спостережний режим, або отримує таймер відродження. Сам процес респавну включає обрання точки відродження (рандомно або відповідно до логіки карти), скидання параметрів стану до початкових (повне здоров’я, початкова зброя), та запуск супутніх подій (анімації, відображення на HUD, інформування інших гравців). Сервер має ініціювати респавн після завершення затримки й передати відповідні події клієнтам.

Управління життєвим циклом гравця (життя - смерть - очікування - відродження) повинно бути чітко структурованим. Це часто реалізується через систему станів (наприклад, FSM – finite state machine), де кожен стан (живий, мертвий, у режимі відродження) має власну логіку оновлення й реагування на зовнішні події. З архітектурної точки зору, модуль стану гравця доцільно відокремити в самостійний компонент, який відповідає лише за облік здоров’я, броні та стану. Для збереження синхронності ці параметри мають бути мережевими змінними, які оновлюються централізовано й реплікуються усім гравцям.

2.4.4 UI/HUD: принципи реактивного оновлення інформації

Система UI/HUD (інтерфейсу користувача) виконує критичну роль у

забезпеченні гравця актуальною інформацією про стан персонажа, боєзапас, пінг, події у грі тощо. Щоб така система була ефективною в реальному часі, її реалізація повинна ґрунтуватися на принципах реактивного оновлення, оптимізації продуктивності та точній синхронізації з внутрішнім станом гри.

Принцип реактивного оновлення передбачає, що UI не оновлюється кожен кадр вручну, а реагує лише на зміну відповідних параметрів – наприклад, здоров'я, кількість патронів чи рівень затримки. Це досягається завдяки використанню властивостей (наприклад, BindableProperty, ReactiveProperty, ScriptableObject-ів або спеціальних UI-біндінгів), які автоматично викликають оновлення візуального елементу лише в момент зміни значення. Таким чином, коли здоров'я зменшується після отримання шкоди або змінюється боєзапас після пострілу, відповідний UI-елемент оновлюється без потреби в постійному моніторингу. [18]

Що стосується пінгу, цей показник зазвичай оновлюється з певною періодичністю (наприклад, раз на кілька секунд), і виводиться в окремий UI-елемент.

Пінг – це ключовий показник якості з'єднання гравця із сервером, і його відображення дозволяє користувачеві оперативно реагувати на можливі затримки або проблеми з мережею. У Fusion пінг можна отримати через відповідні API NetworkRunner, але важливо не оновлювати його щокадрово, щоб не створювати зайвого навантаження на UI-підсистему.

Одним із пріоритетів при побудові HUD є мінімізація навантаження на рендеринг. Це особливо актуально для мобільних пристроїв або платформ із обмеженими ресурсами. Щоб уникнути перевантаження GPU, рекомендується:

- використовувати Canvas, розділені за типами оновлення (наприклад, окремо для динамічних і статичних елементів);
- уникати частих викликів SetActive, перекомпіляцій layout-ів і змін RectTransform, які можуть запускати перерахунок всього UI;
- групувати оновлення значень – наприклад, об'єднувати

відображення боєзапасу та здоров’я в один блок, що оновлюється раз на декілька кадрів, а не миттєво;

- по можливості використовувати спрайти, замість текстових елементів, які потребують більше ресурсів на рендерінг.

З архітектурної точки зору, UI-система має бути відокремлена від внутрішньої логіки гравця. Це означає, що дані про здоров’я, боєприпаси чи стан гравця зберігаються на ігровому об’єкті або в менеджері, й передаються у вигляді подій чи через централізовані сервіси в UI. Таким чином, інтерфейс працює лише як візуальне відображення поточного стану й не виконує жодної логіки – що спрощує тестування, масштабування й обслуговування системи.

В онлайн-проектах особливо важливо, щоб UI не залежав від локальних розрахунків.

Наприклад, дані про залишок здоров’я чи кількість патронів мають надходити з авторитетного джерела (зазвичай – від локального NetworkObject, який є власником об’єкта гравця), і не можуть оновлюватися виключно на основі клієнтських припущень.

Це знижує ризик розсинхронізації й гарантує коректність відображення навіть при втраті пакетів або короткочасних перебоях з’єднання. UI/HUD-система в мультиплеєрному шутері має базуватися на реактивному підході, синхронізації зі станом гравця, оптимізації продуктивності й відокремленні візуальної частини від логіки гри.

Це дозволяє досягти високої стабільності, зрозумілості та ефективності в ігровому процесі незалежно від платформи чи якості мережевого з’єднання. Також завдяки чіткому поділу відповідальності в UI-структурі, розробнику легко розмежовувати логіку роботи інтерфейсу та його візуальне відображення, що значно спрощує супровід та масштабування проекту.

2.4.5 Матчмейкінг і лобі

Система матчмейкінгу та лобі виконує фундаментальну функцію – забезпечує організований пошук гравців, формування команд, налаштування параметрів гри й синхронний перехід у матч. Ця система є проміжним етапом між входом у гру й безпосереднім ігровим процесом, тому її коректна архітектура має ключове значення для зручності користувача й стабільності всього мультиплеєра. [19]

Матчмейкінг – це процес, під час якого гравець або створює нову ігрову сесію, або приєднується до вже існуючої. Зазвичай він реалізується на базі механізму пошуку або фільтрації доступних кімнат (сесій), або автоматично – за допомогою системи швидкого підбору гравців (quick match). У Fusion це реалізується через інтерфейс NetworkRunner.JoinSession та StartGame, де можна вказувати назву сесії, тип кімнати (публічна, приватна), максимальну кількість гравців тощо. У рамках проєкту може бути реалізована система фільтрації кімнат за регіоном, пінгом, режимом гри чи кількістю гравців, що вже приєдналися.

Коли гравці потрапляють у лобі (до початку самої гри), починається другий важливий етап – синхронізація налаштувань. У лобі можуть бути доступні такі елементи: вибір персонажа чи класу, налаштування спорядження, обрання карти, погодження правил гри (наприклад, кількість фрагів до перемоги або обмеження по часу).

Тут важливо, щоб усі гравці мали однакову інформацію та бачення поточних налаштувань. Це досягається або через централізовану логіку на стороні хоста (у Host Mode), або шляхом визначення "лідера" лобі, який має право вносити зміни, що потім реплікуються всім іншим клієнтам. Photon Fusion дозволяє ефективно реалізувати таку логіку за допомогою NetworkObject-ів, які мають Networked Properties, і через RPC – відправку команд від одного клієнта до інших. Наприклад, вибір гравцем зброї може надсилатись через RPC до лобі-менеджера, який зберігає всі вибори й

транслює їх назад для відображення в інтерфейсі. Такий підхід гарантує консистентність даних – усі бачать однакові налаштування перед стартом гри. У момент, коли всі гравці готові, ініціюється перехід із лобі в сам матч. Це супроводжується передачею інформації про склад учасників, їхні вибори, ігровий режим тощо. У Fusion ця логіка реалізується як перехід у нову сцену (SceneManager.LoadScene), який може бути ініційований хостом або централізованим контролером, після чого відбувається повторна ініціалізація NetworkRunner у режимі гри. Ключовим викликом у реалізації матчмейкінгу є забезпечення цілісності даних при приєднанні нових гравців. Система має бути здатною передати новоприбулому поточний стан усіх налаштувань, учасників, їхніх статусів (готовий/не готовий) тощо. Це часто досягається через State Authority або спеціальні компоненти, що відповідають за збереження глобального стану лобі та синхронізують його з новими клієнтами через Networked State або OnPlayerJoined callback.

2.4.6 Система чату

Реалізація власного чату на базі Photon Fusion є ключовим елементом, що значно покращує якість взаємодії між гравцями та загальний досвід гри. Чат виконує не тільки функцію простого обміну текстовими повідомленнями, а й сприяє формуванню спільноти, командній координації, а також підвищує залученість користувачів у процес гри. У сучасних онлайн-іграх комунікація – це не розкіш, а необхідність, тому важливо, щоб чат був надійним, швидким і безпечним. Для реалізації чату на Photon Fusion використовується мережевий стек, який забезпечує передачу даних між клієнтами та сервером або авторитетним вузлом. Кожне повідомлення представляє собою певний об’єкт, який передається через мережу, часто за допомогою Remote Procedure Calls (RPC) або Networked Properties. Ці механізми дозволяють не лише відправити повідомлення від одного гравця до інших, але й контролювати правильність та послідовність їх отримання. Надійність доставки

повідомлень – важливий фактор, оскільки в онлайн-іграх навіть невеликі затримки або втрата даних можуть знизити якість комунікації і викликати роздратування у користувачів. З технічної точки зору чат необхідно реалізувати як окремий сервіс, який взаємодіє з основною ігровою логікою, але при цьому працює автономно, щоб не перевантажувати мережу і не створювати додаткових затримок для геймплею. Важливо враховувати, що повідомлення від гравців можуть надходити з різною частотою, а у великих матчах кількість учасників може сягати десятків чи сотень, тому слід передбачити оптимальні механізми масштабування. Це може включати агрегування повідомлень, використання черг повідомлень, а також розумне розподілення трафіку через relay-сервери або інші сервіси Photon. [20]

Ще одним критичним аспектом є захист від шахрайства і зловживань. Оскільки чат є відкритою комунікаційною платформою, у ньому може з’являтися небажаний або образливий контент. Для запобігання цьому слід впроваджувати різні рівні фільтрації – від базових перевірок на нецензурну лексику до складних алгоритмів машинного навчання, що аналізують контекст повідомлень. Важливо також мати механізми для модерації, як автоматичної, так і ручної, а також можливість користувачам повідомляти про порушення. Крім того, слід передбачити обмеження на частоту відправки повідомлень, щоб уникнути спаму, а також впровадити блокування користувачів, які систематично порушують правила.

Інтерфейс користувача для чату повинен бути максимально зручним і ненав’язливим. Важливо, щоб інформація про нові повідомлення відображалась швидко і зрозуміло, але при цьому не заважала основному ігровому процесу. Це досягається шляхом використання реактивних UI-патернів, які оновлюють тільки ті частини інтерфейсу, де з’явилися нові повідомлення, мінімізуючи навантаження на рендеринг. Можна також впроваджувати функції сортування та фільтрації повідомлень, виділення важливих або приватних повідомлень, а також історію переписки, яка завантажується динамічно.

Важливо пам’ятати про безперервність зв’язку. При поганому інтернет-з’єднанні або тимчасових проблемах з мережею чат повинен коректно обробляти відсутність зв’язку, забезпечуючи повторну доставку повідомлень і оновлення стану після відновлення підключення. Для цього застосовуються буфери і черги, що зберігають повідомлення локально до моменту успішної відправки.

3 РОЗРОБКА ТА РЕАЛІЗАЦІЯ ПРОЕКТУ

3.1 Загальна архітектура реалізації

В основі архітектури реалізації мережевого шутера на Unity лежить прагнення до максимальної модульності та гнучкості, що дозволило організувати весь проект так, щоб кожен його аспект – від ініціалізації мережевої сесії до відображення фінальних кадрів на HUD – працював у єдиному, але легко масштабованому потоці. У кореневій директорії Unity-проекту всі скрипти згруповано за функціональними категоріями, що значно полегшує навігацію та підтримку коду (рисунок 3.1).

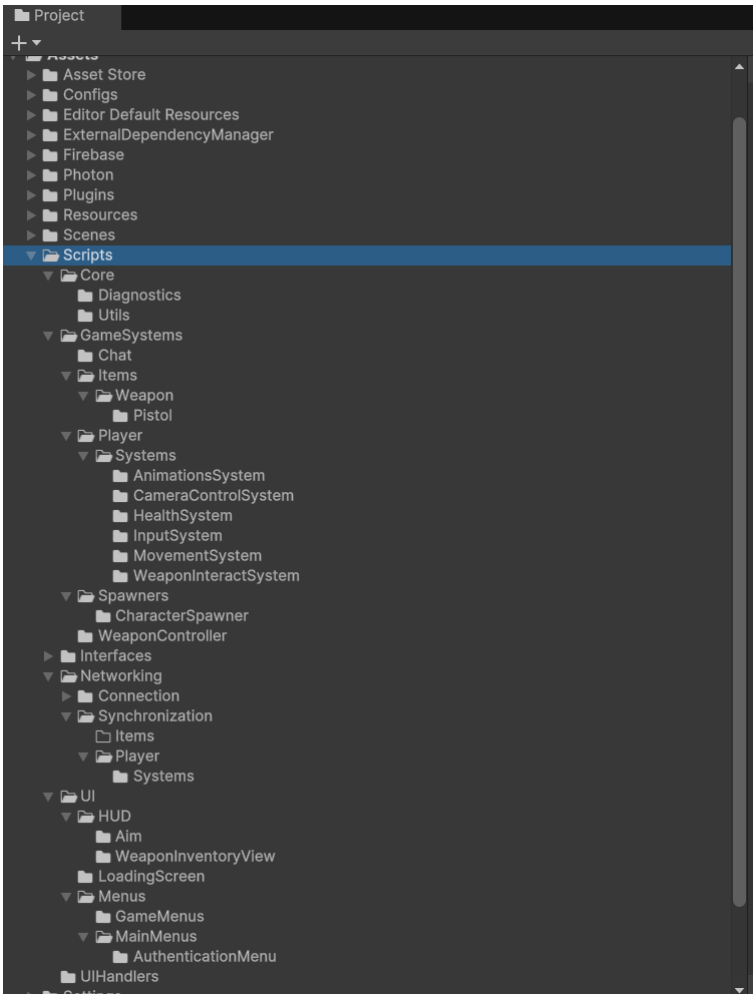

Рисунок 3.1 – Структура папок проекту

Запуск гри розпочинається з класу Bootstrap, який у Awake створює єдиний екземпляр програми, підвантажує налаштування PhotonAppSettings та SessionConnectionConfig і передає керування механізму Fusion Runner для роботи через UDP, управління RPC-викликами та синхронізації станів (рисунок 3.2).

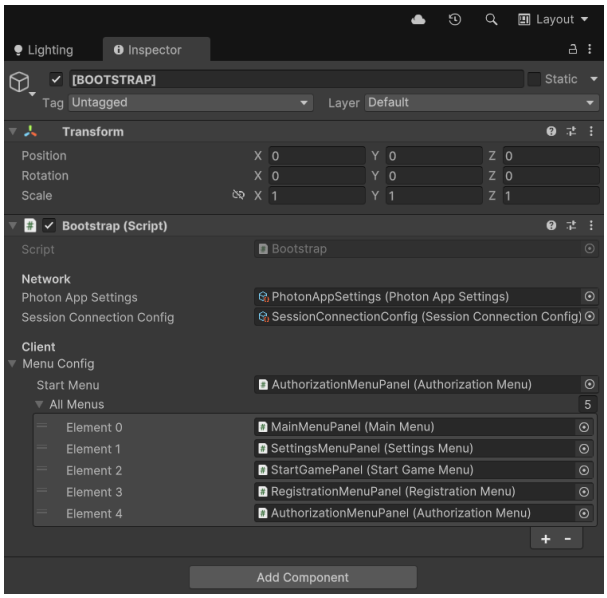

Рисунок 3.2 – Компонент Bootstrap, який ініціалізує глобальні сервіси та перехід в ігрову сцену

Одразу після цього відбувається вибір режиму авторитету – Host або Shared, встановлюються необхідні таймштампи та буфери для алгоритмів інтерполяції й екстраполяції. Після первинного встановлення зв’язку з Fusion, коли мережевий рівень повідомляє про готовність, керування передається скрипту EntryPoint, розташованому безпосередньо в сцені GameScene. Саме через компонент EntryPoint відбувається ініціалізація CharacterSpawnerConfig та GameMenuConfig, створюється гравець, налаштовується стартове меню й HUD-фабрика та підписуються обробники подій з NetworkCallbacksHandler (рисунок 3.3). Далі обробники SessionConnectionHandler і SessionDisconnectHandler перехоплюють події входу в лобі, підключення та відключення гравців і ретранслюють їх до внутрішніх систем гри, зберігаючи повну незалежність мережевої логіки від

ігрових механік. Паралельно модуль Firebase Authentication здійснює асинхронну реєстрацію та вхід користувача, валідує токени й повертає управління назад у EntryPoint для остаточного конфігурування Photon Fusion Runner та формування UI з урахуванням прав доступу гравця.

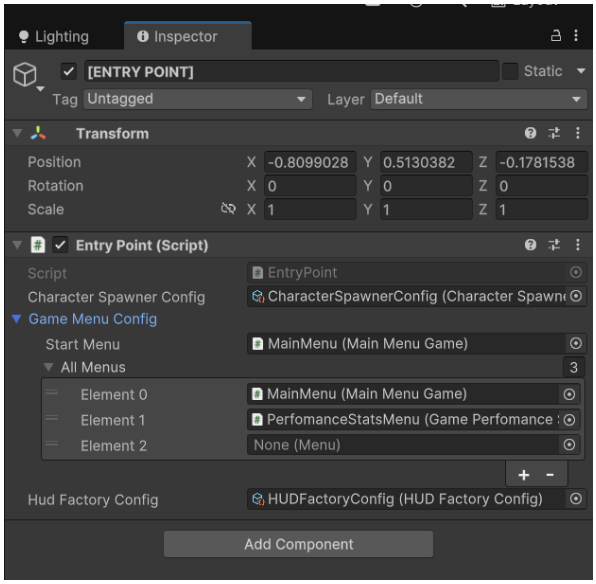

Рисунок 3.3 – Компонент EntryPoint, який ініціалізує ігрові системи та гравця

У папці GameSystems/Items/Weapon знаходиться базовий абстрактний клас Weapon, який успадковує NetworkBehaviour та реалізує інтерфейси IWeaponNotifiers і IStateAuthorityChanged. Саме він задає загальні властивості зброї – Damage, AttackRate, Range та звукові кліпи пострілу, а також ініціалізує в Awake компоненти Rigidbody і AudioSource. У похідних класах, наприклад Pistol, конкретизуються параметри типу зброї, такі як функція розкиду чи ємність магазину. Безпосередньо взаємодію гравця зі зброєю обробляє скрипт WeaponController: він реагує на ввід, викликає метод Attack() у активного Weapon через RPC-механізми Photon Fusion і координує відтворення візуальних і звукових ефектів, оновлює лічильники боєкомплекту та викликає подію OnAttack для UI. Такий поділ відповідальностей дозволяє легко додавати нові види зброї. Далі у папці GameSystems NetworkCharacterController разом із CharacterSpawner перетворюють Unity-ввід у рухи персонажів з інтерполяцією та компенсацією

пінгу, а NetworkCharacterHealth опікується життєвим циклом персонажа, обробкою ушкоджень та респавном. UI Manager, використовуючи патерн Observer, реагує на зміни станів і реактивно оновлює елементи HUD – від індикаторів здоров’я і боєкомплекту до лічильників часу й чат-вікна в лобі – без зайвих затримок. Поєднавши фасадний підхід для спрощення інтерфейсів взаємодії з мережевими сервісами та централізовану точку входу через EntryPoint, архітектура перетворилася на цілісну екосистему, готову до розширення новими режимами, картами та інтеграцією сторонніх сервісів. Таке рішення дозволяє будь-якому новому розробнику швидко зорієнтуватися в структурі папок та скриптів і продовжити роботу над оптимізацією продуктивності, покращенням мережевої стабільності чи доповненням функціоналу.

3.2 Розробка та інтеграція ігрових систем

3.2.1 Реалізація системи руху

У реалізації системи руху було важливо зберегти максимальну гнучкість і розділити відповідальність між обробкою вводу гравця та безпосереднім переміщенням персонажа у світі. Для цього в класі NetworkCharacterController було створено два окремі компоненти – CharacterMovementInput, що читає та нормалізує натискання клавіш, і CharacterMovement, яке переводить отримані вектори в рух через стандартний CharacterController Unity.

Першим кроком при завантаженні сцени в Awake() відбувається ініціалізація цих двох об’єктів із конфігураційними скрипт-об’єктами (лістинг 3.1), які задають такі параметри, як швидкість ходьби та бігу, прискорення, сила стрибка та параметри гравітації.

Наприклад, конфігурація руху передається в конструктор CharacterMovement у вигляді CharacterMovementConfig, а

CharacterMovementInputConfig слугує для налаштування клавіш бігу і стрибка.

Лістинг 3.1 – Ініціалізація компонентів руху та обробки вводу (клас NetworkCharacterController)

```
CharacterMovementInput = new(_characterInputConfig);
CharacterMovement = new(_characterMovementConfig,
CharacterController)
{
    RotateNotifier = CameraControl
};
```

Всередині CharacterMovementInput відбувається виклик стандартних методів Unity Input System (лістинг 3.2). При кожному виклику UpdateMovementInput() клас перевіряє, чи дозволений ввід (наприклад, меню може блокувати рух), і лише потім збирає значення осей:

Лістинг 3.2 – Обробка вводу руху (клас CharacterMovementInput)

```
float dirX = isSmoothing
    ? Input.GetAxis("Horizontal")
    : Input.GetAxisRaw("Horizontal");
float dirZ = isSmoothing
    ? Input.GetAxis("Vertical")
    : Input.GetAxisRaw("Vertical");
_direction = new Vector3(dirX, 0f, dirZ).normalized;
```

Така перевірка дозволяє легко переключати між плавним рухом і «жорстким» контролем на зразок снайперської прицільної стрільби.

Крім того, за допомогою властивості IsInputAllowed та підписки на події від IActiveMenuChangedNotifier можна автоматично відключати механіку руху, коли відкрито будь-яке вікно інтерфейсу або якщо персонаж помирає.Перехід від вектора напряму до реального переміщення здійснює CharacterMovement. Спершу він обчислює націлені швидкості по правій і передній осі персонажа, враховуючи, чи натиснута клавіша бігу (лістинг 3.3):

Лістинг 3.3 – Обробка швидкості руху на осях на основі вводу (клас CharacterMovement)

```
float currentMaxSpeed = isRun ? _config.RunSpeed :
_config.WalkSpeed;

float targetSpeedX = direction.x * currentMaxSpeed *
_config.HorizontalSpeedModifier;

float targetSpeedZ = direction.z * currentMaxSpeed;
```

Далі за допомогою Mathf.MoveTowards поступово змінюється поточна швидкість, що створює відчуття прискорення і гальмування. Особливу увагу було приділено руху назад: якщо гравець рухається у зворотному напрямку, швидкість знижується, щоб уникнути неприродного «бігу задом». Потім, обчисливши вектор з урахуванням глобальних осей персонажа, система викликає метод CharacterController.Move() у стандартного компонента Unity, який автоматично оброблює колізії та переміщає на цей вектор, а через подію OnMovement передає отриману швидкість іншим підсистемам – наприклад, анімаційному контролеру, який дає команду на перехід між ходьбою та бігом.

Стрибок і гравітацію було винесено в окремі методи CalculateJump() і ApplyGravity().

Під час натискання кнопки стрибка, якщо персонаж стоїть на землі, обчислюється необхідний імпульс за формулою кінематики, а у гравітації реалізовано простий драг–модель із обмеженням максимальної швидкості падіння, що запобігає надмірному прискоренню при тривалому польоті (лістинг 3.4).

Лістинг 3.4 – Метод, який оброблює стрибок за формулою кінематики (клас CharacterMovement)

```
public void CalculateJump(bool jumpBtnPressed)
    {
        if (jumpBtnPressed && _characterController.isGrounded)
        {
            _gravityVelocity.y = Mathf.Sqrt(_config.JumpForce *
-2f * _config.Gravity);
            OnJump?.Invoke();
        }
    }
```

У самому NetworkCharacterController інтегруються ці дві частини в єдиний цикл: у методі Update() тільки гравець з авторитетом (StateAuthority) обробляє ввід і наказує руху та стрибку, а в FixedUpdate() відбувається застосування розрахованих переміщень і гравітації з фіксованою дельтою часу (лістинг 3.5).

Лістинг 3.5 – Метод, який оброблює гравітацію (клас CharacterMovement)

```
public void ApplyGravity(float timeDelta)
    {
        if (!_characterController.isGrounded)
        {
            float force = _config.Gravity * timeDelta;

            GravityForce += force;

            if (Mathf.Abs(_gravityVelocity.y) >
_maximumFallingSpeed)
                GravityForce = -Mathf.Abs(_maximumFallingSpeed);

        }
        else
        {
            if (_gravityVelocity.y < 0)
                GravityForce = -2f;

        }
        _characterController?.Move(_gravityVelocity *
timeDelta); }
```

Синхронізація руху персонажа по мережі в Photon Fusion побудована на чіткому розділенні виконання логіки на «власника» (State Authority) та «спостерігачів», яких називають поділом на клієнтів із правом обчислювати стан та усіх інших, хто отримує цей стан у вигляді реплік. У нашому випадку повний цикл обробки вводу й переміщення виконується лише на тому клієнті, в якого є авторитет над об’єктом персонажа (лістинг 3.6).

Лістинг 3.6 – Розділення виконання логіки руху «власника» поточного об’єкту (клас NetworkCharacterController)

```
private void Update()
{
    if (Object.HasStateAuthority) // Якщо наш клієнт є власником цього компонента, то оброблюємо стан цього об’єкту
    {
        // Збір вводу та обчислення руху
        CharacterMovementInput.UpdateMovementInput(false);

CharacterMovement.CalculateJump(CharacterMovementInput.IsJumpInput);
    }
}

public void FixedUpdate()
{
    if (Object.HasStateAuthority)
    {
        // Застосування руху та гравітації
        CharacterMovement.CalculateMove(
            CharacterMovementInput.Direction,
            CharacterMovementInput.IsRunInput,
            Time.fixedDeltaTime);
        CharacterMovement.ApplyGravity(Time.fixedDeltaTime);
    }
}
```

Після того, як у FixedUpdate змінюється Transform через CharacterController.Move(), Photon Fusion автоматично пакує нові координати позиції та кути повороту об’єкта в щокадровий мережевий «снапшот» і розсилає їх іншим клієнтам у вигляді дельта-оновлень. На боці приймача Fusion застосовує власні алгоритми інтерполяції та екстраполяції, що дозволяє згладити рух і приховати мережеву затримку. Наявність буферів таймштампів, встановлених під час конфігурації Fusion Runner, гарантує, що кожен клієнт отримує дані з невеликим відтермінуванням, достатнім для коректного проміжного згладження, але без сильного «відставання» стану.

У коді ми не використовуємо додаткові RPC для самої трансформації персонажа – вона відбувається автоматично – але об’єкт все одно наслідує NetworkBehaviour, а Runner керує його життєвим циклом і виконує серіалізацію. При цьому власник об’єкта регулярно викликає Move() і позначає об’єкт як «dirty», тож Fusion знає, що потрібно передати нові координати. Відновлення стану на клієнтах здійснюється через вбудований у

Fusion компонент синхронізації трансформації об’єкту, який за замовчуванням використовує UDP, Reliability Channel для гарантованої доставки ключових оновлень та Unreliable Channel для високочастотних змін, таких як позиція і ротація.

Така архітектура забезпечує плавний контроль, передбачувану фізику руху та ідеальну зв’язку з мережею – адже кожен рух генерує прогнозовані вектори, які легко синхронізувати через Fusion Runner, не витрачаючи зайвих ресурсів на передачу зайвих даних. Завдяки чіткому поділу на CharacterMovementInput і CharacterMovement, а також їхню інтеграцію через NetworkCharacterController, система руху залишається максимально читабельною, розширюваною та готовою до будь-яких майбутніх змін: від підтримки нових режимів керування до адаптації під різні платформи або додавання складної фізики.

3.7 Стрільба та зброя (RPC та ефекти)

У цьому проекті механіка стрільби та управління зброєю побудована на базі абстрактного класу Weapon та його наслідувачів, а також на взаємодії між кількома допоміжними сервісами, які разом формують повноцінний ігровий цикл «підбери – вибери – стріляй – перезаряджай». У класі Weapon, що наслідує NetworkBehaviour, концентруються загальні властивості будь-якої зброї – сила ушкодження (Damage), швидкість атаки (AttackRate) та дальність (Range) – а також базова перевірка можливості атаки (лістинг 3.8).

Лістинг 3.8 – Метод перевірки можливості атаки (клас Weapon)

```
protected virtual bool CanInteract()
{
    return _lastTimeAttack + AttackRate <= Time.time;
}
public void TryAttack(Transform attackPoint)
{
    if (CanInteract())
    {
        AttackPoint = attackPoint;
```

```
            Attack();
            RPC_PlayAttackSound();
            _lastTimeAttack = Time.time;
            OnAttack?.Invoke(this);
        }
}
```

Сам метод Attack() оголошено абстрактним і реалізується у конкретних підкласах, але звук атаки в усіх випадках відтворюється через RPC-виклик (лістинг 3.9).

Лістинг 3.9 – RPC-Метод програвання звуку атаки (клас Weapon)

```
[Rpc]
private void RPC_PlayAttackSound()
{
    AudioSource.PlayOneShot(AttackSound);
}
```

Цей виклик гарантує, що звук почує кожен клієнт одразу після факту атаки, незалежно від того, чи саме на ньому відпрацював TryAttack. Потужнішим доповненням до базової логіки є клас WeaponRanged, який додає до зброї властивості магазинів, перезарядки та різні режими стрільби – «Raycast» або «Physics» – а також одноразові та автоматичні режими (ShootMode). Кількість патронів у магазині та загальний запас зберігаються як мережеві поля з атрибутом [Networked] і автоматично реплікуються на всіх клієнтах, викликаючи методи оновлення через OnChangedRender (лістинг 3.10).

Лістинг 3.10 – Мережеве поле з атрибутом OnChangedRender для реагування на зміни значень (клас WeaponRanged)

```
[Networked,
OnChangedRender(nameof(NetworkCurrentAmmoCountInMagazineChanged)
)]
public int CurrentAmmoCountInMagazine { get; private set; }
```

Коли ж настає подія стрільби, у зафіксованому методі Attack() зменшується лічильник патронів і, якщо обрано режим Raycast, виконується звичайний рейкаст із джіттером (лістинг 3.11).

Лістинг 3.11 – Перевизначений метод Attack(), який застосовує конкретний тип стрільби та віднімає кількість патронів зброї (клас WeaponRanged)

```
protected sealed override void Attack()
{
    CurrentAmmoCountInMagazine--;
    RPC_PlayMuzzleFlash();
    switch (ShootType)
    {
        case ShootType.Raycast:
            RaycastShoot();
            break;
        case ShootType.Physics:
            PhysicsShoot();
            break;
    }
}
```

У RaycastShoot() через звичайний Physics.Raycast визначається попадання, далі викликається метод HitTarget, який також використовує RPC для відтворення ефекту попадання (MuzzleFlash та HitEffect) та наносить реальну шкоду (лістинг 3.12).

Лістинг 3.12 – Метод HitTarget(), який викликається при попаданні у ціль (клас WeaponRanged)

```
protected virtual void HitTarget(RaycastHit hitInfo)
{
    RPC_PlayHitEffect(hitInfo.point, hitInfo.normal);
    if (hitInfo.collider.TryGetComponent<IDamagebleProvider>(out
var dmg))
        dmg.Damageble.TakeDamage((int)Damage);
}
```

У цьому ланцюжку всі візуальні й аудіо ефекти виконуються синхронно та гарантовано на всіх клієнтах, тому гравці бачать і чують один одного одночасно, навіть якщо їх мережеві затримки відрізняються. Безперервну взаємодію гравця зі зброєю бере на себе клас CharacterWeaponActionHandler. Він слухає команди клавіш «вогонь» і

«перезарядка» та, визначивши поточний режим стрільби (Single або Auto), викликає у активної зброї TryAttack(AttackPoint) або Reload() (лістинг 3.13).

Лістинг 3.13 – Взаємодія гравця зі зброєю (клас CharacterWeaponActionHandler)

```
private void UpdateAttackInput()
{
    if (_currentWeapon == null) return;
    bool attackInput = ShootMode.Single
        ? Input.GetKeyDown(_config.DefaultAttackKey)
        : Input.GetKey(_config.DefaultAttackKey);

    if (attackInput)
        _currentWeapon.TryAttack(AttackPoint);}
```

Після перезарядки клас WeaponRanged запускає корутину, де через RPC також програється звук перезарядження, і аж після затримки часу заповнюється магазин (лістинг 3.14).

Лістинг 3.14 – Обробка перезарядки зброї (клас WeaponRanged)

```
[Rpc]
private void RPC_PlayReloadSound()
{
    AudioSource.PlayOneShot(ReloadSound);
}
private IEnumerator ReloadRoutine()
    {
        IsReloading = true;
        RPC_PlayReloadSound();

        yield return new WaitForSeconds(ReloadTime);

        int requiredCountAmmoInMagazine = AmmoCountInMagazine -
CurrentAmmoCountInMagazine;

        CurrentMaxAmmoCount -= requiredCountAmmoInMagazine;

        if (CurrentMaxAmmoCount < 0)
        {
            CurrentMaxAmmoCount = 0;
            CurrentAmmoCountInMagazine =
requiredCountAmmoInMagazine;
        }
        else
            CurrentAmmoCountInMagazine = AmmoCountInMagazine;
```

```
IsReloading = false;}
```

Щоб гравець міг змінювати зброю та підбирати її на карті, були реалізовані WeaponInventory та WeaponPickupSystem. Перший зберігає зброю в слотах і повідомляє про зміну активної зброї через події, а другий через фішку запиту авторитету (RequestStateAuthority) підключає право керування фізикою об’єкта та переміщує його в руку гравця. Встановлені через RPC звуки й ефекти при підборі та скиданні зброї гарантують уніфіковану взаємодію всіх клієнтів із єдиним об’єктом зброї.

Завдяки такій архітектурі, де кожен етап стрільби – від натискання клавіші до нанесення шкоди та відтворення ефектів – проходить крізь перевірені RPC-методи та мережеві поля Fusion, система стрільби працює узгоджено, плавно й безпомилково у будь-яких мережевих умовах. Нові типи зброї або зміни логіки можна вводити, просто наслідуючи базові класи й додаючи потрібні конфігурації, не модифікуючи основні механізми обміну даними між клієнтами.

3.15 Система здоров’я та відродження

Система здоров’я та відродження побудована на поєднанні локальної логіки обробки ушкоджень і глобальної мережевої синхронізації. Базовий клас CharacterHealth інкапсулює всі правила, пов’язані із критичним станом персонажа: він зберігає поточне значення здоров’я, обробляє виклики TakeDamage() та TakeHeal(), автоматично викликає подію OnHealthChanged при зміні значення і запускає OnDie, коли здоров’є опускається до нуля (лістинг 3.16).

Лістинг 3.16 – Метод обробки ушкоджень (клас CharacterHealth)

```
public void TakeDamage(int amount)
{
    int validAmount = Mathf.Abs(amount);

    if (!CanTakeDamage) return;

    if (CurrentHealth - validAmount > 0)
    {
        CurrentHealth -= validAmount;
    }

    else
    {
        CurrentHealth = 0;
        OnDie?.Invoke();
    }
}
```

Однак у мультиплеєрній грі простого локального класу недостатньо – потрібно гарантувати, що кожен клієнт і сервер мають синхронізоване значення здоров’я. Для цього створено NetworkCharacterHealth, який успадковує NetworkBehaviour з Fusion і проксує виклики базового класу у мережу. Після спавну об’єкта на вузлі з авторитетом (StateAuthority) цей компонент створює екземпляр CharacterHealth, підписується на його події та ініціалізує мережеве поле NetworkCurrentHealth (лістинг 3.17).

Лістинг 3.17 – Ініціалізація CharacterHealth та підписка на події (клас NetworkCharacterHealth)

```
public override void Spawned()
{
    if (Object.HasStateAuthority)
    {
        _characterHealth = new CharacterHealth(_config);
        _characterHealth.OnHealthChanged +=
HandleLogicHealthChanged;
        _characterHealth.OnDie += OnDieChange;
        NetworkCurrentHealth = _characterHealth.CurrentHealth;
    }
}

private void HandleLogicHealthChanged(int newHealth)
{
    NetworkCurrentHealth = newHealth;}
```

Атрибут [Networked, OnChangedRender] гарантує, що при зміні NetworkCurrentHealth на вузлах без авторитету автоматично викликається локальний метод OnNetworkHealthChanged(), який у свою чергу викликає подію OnHealthChanged для UI-менеджера та інших систем (лістинг 3.18).

Лістинг 3.18 – Реагування на зміну NetworkCurrentHealth (класс NetworkCharacterHealth)

```
[Networked, OnChangedRender(nameof(OnNetworkHealthChanged))]
public int NetworkCurrentHealth { get; private set; }

private void OnNetworkHealthChanged()
{
    OnHealthChanged?.Invoke(CurrentHealth);
}
```

Запит на збільшення або зменшення здоров’я від гравця теж обробляється через RPC: виклик TakeDamage(int amount) спрямовується на вузол-авторитет (лістинг 3.19).

Лістинг 3.19 – RPC виклики зміни здоров’я (клас NetworkCharacterHealth)

```
public void TakeDamage(int amount)
{
    Rpc_TakeDamage(amount);
}

[Rpc(sources: RpcSources.All, targets:
RpcTargets.StateAuthority)]

private void Rpc_TakeDamage(int amount, RpcInfo info = default)
{
     if (!Object.HasStateAuthority)
     {
          return;
     }

    _characterHealth.TakeDamage(amount);
}
```

Це забезпечує єдину точку внесення змін у здоров'я, виключає гонки даних і гарантує коректну реплікацію у всіх клієнтів.

Відродження гравця реалізовано у службі CharacterSpawner, яка спостерігає за подією смерті через підписку на OnDie у NetworkCharacterHealth.

Коли здоров'я доходить до нуля, викликається метод Respawn(), що стартує асинхронний таймаут відродження, відключаючи ввід руху й стрільби на час кулдауну (лістинг 3.20), після чого персонаж відроджується та вмикається рух та стрільба.

Лістинг 3.20 – Обробка відродження гравця (клас CharacterSpawner)

```
_character.Damageble.OnDie += Respawn;
private async Task ApplyRespawnCooldown()
{
    _character.CharacterMovementInput.IsInputAllowed = false;
    _character.CharacterWeaponActionHandler.IsInputAllowed =
false;
    await
Task.Delay(TimeSpan.FromSeconds(_config.RespawnCooldown));
    _character.CharacterMovementInput.IsInputAllowed = true;
    _character.CharacterWeaponActionHandler.IsInputAllowed =
true;
    _character.CharacterController.Move(respawnPosition);
    _character.transform.localRotation = respawnRotation;
}
```

Метод SpawnNetwork() створює новий екземпляр NetworkCharacterController, встановлює початкові позицію та обертання, підключає повідомники введення і станів, а також повертає створений об'єкт у гру (лістинг 3.21).

Лістинг 3.21 – Метод створення персонажа (клас CharacterSpawner)

```
public Game.Character.NetworkCharacterController
SpawnNetwork(NetworkRunner runner, PlayerRef player)
{
    var character = runner.Spawn(_config.Prefab, spawnPosition,
spawnRotation, player);
    character.CharacterMovementInput.ActiveMenuChangedNotifier =
_dependencies.ActiveMenuChangedNotifier;
```

```
    character.Damageble.OnDie += Respawn;
    OnSpawned?.Invoke(character);
    return character;
}
```

3.22 Створення лобі та матчмейкінг

У проекті підготовка лобі, створення та приєднання до матчів реалізована через взаємодію трьох основних компонентів: SessionConnectionHandler, NetworkCallbacksHandler та PlayerJoinHandler.

По-перше, на початку гри користувач вводить назву сесії (лобі) і натискає кнопку «Створити» або «Приєднатися». Цю подію обробляє SessionConnectionHandler, який викликає NetworkRunner.StartGame(...) із налаштованими параметрами – режимом гри, максимальною кількістю гравців та ім’ям лобі. Після успішного створення або підключення до сесії, якщо цей клієнт має права авторитету над сценою (IsSceneAuthority), він завантажує ігрову сцену (лістинг 3.23).

Лістинг 3.23 – Метод-обробник підключення до сесії гри (клас SessionConnectionHandler)

```
public async Task<StartGameResult> StartGameAsync(string
sessionName, DisplayTextHandler displayTextHandler)
{
    StartGameArgs startGameArgs = new ()
    {
        GameMode          = _config.GameMode,
        SessionName       = sessionName,
        SceneManager      = _sceneManager,
        CustomLobbyName   = _config.LobbyNameDefault,
        PlayerCount       = _config.MaxCountPlayers
    };

    var result = await _runner.StartGame(startGameArgs);
    if (result.Ok && _runner.IsSceneAuthority)
    {
        await _runner.LoadScene(SceneRef.FromIndex(1));
    }
    return result;
}
```

По-друге, всі мережеві події підключення й відключення гравців централізовано передаються через NetworkCallbacksHandler, який реалізує інтерфейс INetworkRunnerCallbacks. Ці події відправляє Fusion за допомогою інтерфейсу INetworkRunnerCallbacks. Потім цей об’єкт треба зареєструвати у колбеки Fusion, щоб він зміг знайти цей об’єкт та визивати потрібні методи. Зокрема, коли новий гравець приходить у сесію або йде з неї, викликаються відповідні мережеві колбеки (лістинг 3.24).

Лістинг 3.24 – Обробник мережевих подій (клас NetworkCallbacksHandler)

```
public void OnPlayerJoined(NetworkRunner runner, PlayerRef
player)
{
    OnPlayerJoinedEvent?.Invoke(runner, player);
}

public void OnPlayerLeft(NetworkRunner runner, PlayerRef player)
{
    OnPlayerLeftEvent?.Invoke(runner, player);
}
```

Ці події підписуються на обробники в PlayerJoinHandler. Саме тут вирішується, коли дійсно слід створювати нового персонажа або, навпаки, прибирати його із гри (лістинг 3.25).

Лістинг 3.25 – Підписка на обробники в PlayerJoinHandler (клас PlayerJoinHandler)

```
public PlayerJoinHandler(CharacterSpawner characterSpawner)
{
    _characterSpawner = characterSpawner;
}

public IPlayerConnectNotifiers PlayerConnectNotifier
{
    set
    {
        _playerConnectNotifier.OnPlayerJoinedEvent +=
OnPlayerJoined;
        _playerConnectNotifier.OnPlayerLeftEvent   +=
OnPlayerLeft;}}
```

У методі OnPlayerJoined перевіряється, чи це наш локальний гравець (runner.LocalPlayer).

Якщо так, викликається спавн через CharacterSpawner, і збережений об’єкт додається до словника _joinedPlayers (лістинг 3.26).

Лістинг 3.26 – Обробка підключення гравця (клас PlayerJoinHandler)

```
private void OnPlayerJoined(NetworkRunner runner, PlayerRef
player)
{
    if (runner.LocalPlayer != player) return;

    var character = _characterSpawner.SpawnNetwork(runner,
player);
    _joinedPlayers.Add(player,
character.GetComponent<NetworkObject>());
}
```

Коли ж гравець виходить із сесії, OnPlayerLeft перевіряє наявність цього гравця в _joinedPlayers і викликає runner.Despawn(...), видаляючи його об’єкт зі сцени (лістинг 3.27).

Лістинг 3.27 – Обробка відключення гравця (клас PlayerJoinHandler)

```
private void OnPlayerLeft(NetworkRunner runner, PlayerRef
player)
{
    if (runner.LocalPlayer != player) return;

    if (_joinedPlayers.TryGetValue(player, out var netObj))
    {
        runner.Despawn(netObj);
        _joinedPlayers.Remove(player);
    }
}
```

3.2.5 Реалізація чату

Чат у грі реалізовано через компонент ChatController, який наслідує NetworkBehaviour Fusion і обробляє відкриття/закриття вікна чату, введення повідомлення та його мережеву відправку. При натисканні клавіші (за замовчуванням T) вмикається інтерфейс (CanvasGroup), курсор звільняється, а TMP_InputField фокусується для введення тексту. Після натискання клавіші відправки (Return) викликається метод (лістинг 3.28).

Лістинг 3.28 – Обробка відправлення повідомлень (клас ChatController)

```
private void SendNetworkMessage(string sender, string message)
{
    if (Runner == null) return;
    RPC_ReceiveMessage(sender, message);
}

[Rpc(sources: RpcSources.All, targets: RpcTargets.All)]
private void RPC_ReceiveMessage(string sender, string message,
RpcInfo info = default)
{
    DisplayMessage(sender, message);
}
```

Цей RPC гарантує, що кожен клієнт отримає повідомлення одразу після його відправки. У методі DisplayMessage створюється новий екземпляр префабу ChatMessage у вмістовному контейнері, відображаючи ім’я гравця та текст. Одночасно підтримується черга обмеженої довжини – старі повідомлення автоматично видаляються, коли їх більше за налаштовану maxMessages. Такий підхід забезпечує простий і надійний мультиплеєрний чат без додаткових серверних комунікацій.

3.3 Взаємодія з Firebase (аутентифікація)

Ініціалізація зв’язку з Firebase відбувається одразу при створенні об’єкта FirebaseConnector, котрий у конструкторі запускає асинхронну

перевірку залежностей через FirebaseApp.CheckAndFixDependenciesAsync(). Успішне повернення статусу DependencyStatus.Available встановлює внутрішній прапор IsConnected = true та викликає подію OnTryConnected, дозволяючи всім підписаним компонентам дізнатися про готовність бекенду до роботи (лістинг 3.29). Цей підхід забезпечує негайну реакцію системи на зміну статусу підключення, що є критично важливим для своєчасного старту взаємодії з хмарним сервісом.

Лістинг 3.29 – Підключення до Firebase (клас FirebaseConnector)

```
private async Task ConnectToFireBaseAsync()
{
    var status = await
FirebaseApp.CheckAndFixDependenciesAsync();
    if (status == DependencyStatus.Available)
        IsConnected = true;
    OnTryConnected?.Invoke(status);
}
```

Після успішного підключення AuthenticationHandler бере на себе логіку реєстрації та входу користувача. У конструкторі цього класу відбувається перевірка наявного збереженого сеансу через FirebaseAuth.DefaultInstance. CurrentUser, а також підписка на подію OnTryConnected для отримання остаточного статусу. Реєстрація і вхід обробляються одним приватним методом AuthenticateAsync, який залежно від типу (Registration або Authorization) викликає відповідний асинхронний API Firebase (лістинг 3.30).

Лістинг 3.30 – Обробка реєстрації та авторизації користувача (клас AuthenticationHandler)

```
Task<AuthResult> task = authenticationType ==
AuthenticationType.Registration
    ?
FirebaseAuth.DefaultInstance.CreateUserWithEmailAndPasswordAsync
(email, password)
    :
FirebaseAuth.DefaultInstance.SignInWithEmailAndPasswordAsync(ema
il, password);

var result = await task;
```

```
if (task.IsCompletedSuccessfully && authenticationType ==
AuthenticationType.Registration)
{
    _currentUser = result.User;
    await _currentUser.UpdateUserProfileAsync(new UserProfile {
DisplayName = username });
    OnRegistrationComlpete?.Invoke(result);
}
else if (task.IsCompletedSuccessfully)
{
    _currentUser = result.User;
    OnAuthorizationComlpete?.Invoke(result);
}
OnAuthenticationComplete?.Invoke(result);
SaveRememberMe(remember);
```

У разі реєстрації після створення облікового запису виконується ще один виклик UpdateUserProfileAsync, щоб встановити псевдонім користувача. Після успішної аутентифікації AuthenticationHandler викликає OnAuthenticationComplete, що дозволяє UI-панелям оновити інтерфейс (наприклад, перейти від екрану входу до головного меню). Опція «Remember Me» зберігається через PlayerPrefs і дозволяє автоматично виявляти минулі сеанси під час наступних запусків.

3.4 Побудова UI/HUD

Інтерфейс гравця в мережевому шутері складається з двох головних частин: безпосереднього HUD у грі, який відображає стан здоров’я, патронів та слоти, і системи меню, яка керує діалоговими вікнами (лобі, пауза тощо). Відповідальність за створення й оновлення HUD покладена на класи HUDController і HUDFactory. При створенні персонажа через CharacterSpawner подія OnSpawned передається до HUDFactory, який динамічно інстанціює префаб контролера HUD (лістинг 3.31).

Лістинг 3.31 – Створення HUD гравця (клас HUDFactory)

```
private void CreateHUD(NetworkCharacterController
networkCharacterController)
{
```

```
    var hud = Instantiate(_config.HUDControllerPrefab);
    hud.HealthNotifiers = networkCharacterController.Damageble;
    hud.WeaponInventoryView.WeaponInventoryNotifiers =
networkCharacterController.WeaponInventory;
    hud.WeaponPickupSystemNotifiers =
networkCharacterController.WeaponPickupSystem;
    hud.AimHandler.WeaponPickupSystemNotifiers =
networkCharacterController.WeaponPickupSystem;
}
```

HUDController підписується на події зміни здоров'я та боєзапасу. При зміні здоров'я він отримує нове значення через інтерфейс IHealthNotifiers і перераховує заповнення Slider у методі HealthBarUpdate (лістинг 3.32).

Лістинг 3.32 – Оновлення здоров'я на HUD (клас HUDController)

```
private void HealthBarUpdate(int amount)
{
    float ratio = (float)_healthNotifiers.CurrentHealth /
_healthNotifiers.Config.MaxHealth;
    _healthBar.value = ratio;
}
```

Для патронів він використовує інтерфейс IRangedWeaponNotifiers, підписуючись на події CurrentMaxAmmoCountChanged та CurrentAmmoCountInMagazineChanged, щоби одразу оновити відповідні поля TMP_Text (лістинг 3.33).

Лістинг 3.33 – Підписка на події зміни кількості патронів зброї для відображення на HUD (клас HUDController)

```
private void SubscribeToWeaponEvent(IWeaponPickable
weaponPickable)
{
    if (weaponPickable is IRangedWeaponNotifiers rangedWeapon)
    {
        rangedWeapon.CurrentMaxAmmoCountChanged +=
CurrentMaxAmmoCountUpdate;
        rangedWeapon.CurrentAmmoCountInMagazineChanged +=
CurrentAmmoCountInMagazineUpdate;

CurrentMaxAmmoCountUpdate(rangedWeapon.CurrentMaxAmmoCount);

CurrentAmmoCountInMagazineUpdate(rangedWeapon.CurrentAmmoCountIn
```

```
Magazine);
    }
}
```

Поряд із польовим відображенням інформації HUD у грі реалізована цілісна система меню, яка відповідає за всі екранні вікна: лобі, налаштувань, паузи, авторизації тощо. В її основі лежить абстрактний клас Menu, що вже містить базову логіку показу й приховування, а також подію OnChangeActive, яка повідомляє підписані компоненти (наприклад, класи вводу руху й стрільби) про блокування або розблокування користувацького інтерфейсу (лістинг 3.34).

Лістинг 3.34 – Базова логіка для усіх меню (клас Menu)

```
public abstract class Menu : MonoBehaviour,
IActiveMenuChangedNotifier
{
    public event Action<bool> OnChangeActive;
    private bool _isActive;
    public bool IsActive
    {
        get => _isActive;
        private set
        {
            _isActive = value;
            OnChangeActive?.Invoke(_isActive);
        }
    }

    public virtual void Show() => IsActive = true;
    public virtual void Hide() => IsActive = false;
}
```

Усі конкретні вікна (наприклад, LoginMenu, SettingsMenu, PauseMenu) наслідують цей базовий клас і лише задають власні UI-елементи та обробники кнопок, не турбуючись про загальну поведінку відкриття/закриття.

Менеджер MenuSwitcher отримує список усіх цих меню при ініціалізації, ховає їх і відслідковує поточне активне вікно. Виклик OpenMenu(someMenu) автоматично приховує попереднє меню та показує

вибране (лістинг 3.35). Це треба для зручної навігацію між екранами та та підтримкою єдиного активного інтерфейсу.

Лістинг 3.35 – Менеджер MenuSwitcher (класс MenuSwitcher)

```
public sealed class MenuSwitcher
{
    private readonly List<Menu> _menus;
    private Menu _currentMenu;

    public MenuSwitcher(List<Menu> menus)
    {
        _menus = menus;
        foreach (var menu in _menus)
        {
            menu.Hide();
            menu.Initialize(this);
        }
    }

    public void OpenMenu(Menu menu)
    {
        _currentMenu?.Hide();
        menu.Show();
        _currentMenu = menu;
    }
}
```

4 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1 Системні вимоги

Для комфортного й безперебійного геймплею цей мережевий шутер на Unity із Photon Fusion та Firebase вимагає певного рівня апаратного забезпечення і якості мережевого з’єднання. На десктопах оптимальним середовищем є сучасна 64-бітна ОС Windows (10 чи 11) з мінімум чотирма гігабайтами оперативної пам’яті та двоядерним процесором класу Intel Core i3 або AMD Ryzen 3. Хоча інтегровані графічні чіпи можуть забезпечити базовий рівень зображення, для більш плавного відтворення візуальних ефектів та стабільного фреймрейту рекомендується дискретна відеокарта з окремими чотирма гігабайтами відеопам’яті. Для встановлення гри достатньо декілька сотень мегабайт на диску, однак рекомендується вільний простір не менше пів гігабайта, щоб врахувати оновлення та тимчасові файли.

Ключовим фактором у мультиплеєрі є якість інтернет-з’єднання: навіть найпотужніший комп’ютер не зможе приховати велику затримку або нестабільність пакету. Для мінімальної роботи досить пропускної здатності близько 5 мегабіт на секунду й пінгу до 200 мілісекунд, але для ігрового комфорту рекомендується широка смуга від 20 мегабіт і нижчий ping (близько 50–100 мс), особливо в швидкоплинних перестрілках. У мобільній версії проект підтримує Android 8.0 і вище з двома гігабайтами оперативної пам’яті і базовим графічним API OpenGL ES 3.0.

Мобільні пристрої можуть використовувати як домашню Wi-Fi мережу, так і мобільний інтернет 4G/LTE, однак у другому випадку обов’язково звернути увагу на стабільність сигналу і відсутність значних пікових втрат пакетів. Дотримання цих вимог забезпечує плавну роботу інтерполяції руху, швидку реакцію на дії гравців та надійну синхронізацію усіх ключових механік гри.

4.2 Встановлення та запуск гри

Для запуску гри спочатку необхідно завантажити відповідний інсталяційний пакет – на ПК це звичайний .exe інсталятор або .zip архів із готовою грою, на Android – .apk-файл. Після розпакування на стаціонарному комп’ютері (рисунок 4.1) достатньо запустити головний виконуваний файл, який автоматично створить підпапку з налаштуваннями користувача й під’єднається до серверів Photon Fusion. Перший запуск може зайняти кілька секунд, оскільки система перевіряє конфігурацію мережевого клієнта та інсталює необхідні залежності Firebase.

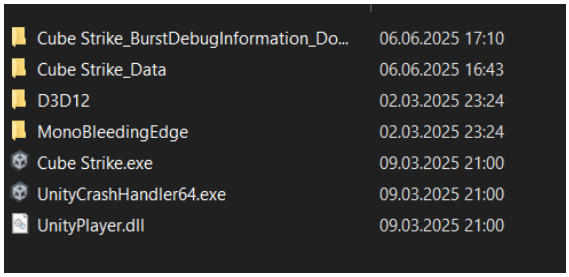

Рисунок 4.1 – Папки і файли гри

На мобільних пристроях потрібно дозволити встановлення з невідомих джерел (якщо ви встановлюєте через .apk), після чого відкрити файл інсталятора через будь-який файловий менеджер. Після завершення інсталяції з’явиться ярлик гри в меню додатків. При першому запуску програма запитає дозвіл на доступ до інтернету та, за необхідності, відкриє екран аутентифікації через Firebase. Лише після успішного входу або створення акаунта користувач потрапляє в головне меню, де можна створити або приєднатися до мережевого лобі. Якщо під час встановлення виявляються відсутні компоненти (наприклад, старіша версія .NET або відсутня підтримка TLS), програма виведе повідомлення з інструкцією, що саме слід оновити. Після успішного встановлення та аутентифікації ваша копія гри повністю готова до гри у мультиплеєрному режимі.

4.3 Авторизація гравця

При запуску гри на екрані відображається форма авторизації, яка містить два поля для введення – електронної пошти та пароля – а також прапорець «Запам’ятати мене» і кнопки «Продовжити» та переходу на форму реєстрації (рисунок 4.2). Після введення цих даних натискання кнопки «Продовжити» передає логін і пароль до AuthenticationHandler, який через Firebase Auth виконує асинхронний вхід. Якщо прапорець «Запам’ятати мене» було активовано, результат успішної аутентифікації зберігається у PlayerPrefs, і при наступному запуску програма автоматично відновить сесію без повторного введення облікових даних.

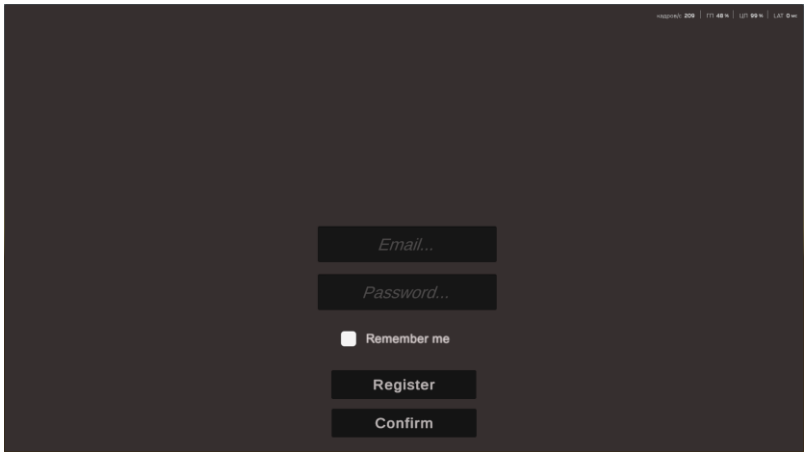

Рисунок 4.2 – Меню авторизації

Якщо користувача ще немає в системі, він може натиснути кнопку «Реєстрація», що відкриває окрему форму (рисунок 4.3). У ній розташовано чотири поля: ім’я користувача (псевдонім), пароль, підтвердження пароля та електронна пошта, а також кнопка «Відмінити» (вихід), яка повертає назад до вікна входу. Під час реєстрації AuthenticationHandler спочатку створює обліковий запис через CreateUserWithEmailAndPasswordAsync, а потім виконує UpdateUserProfileAsync для встановлення псевдоніма в полі DisplayName.

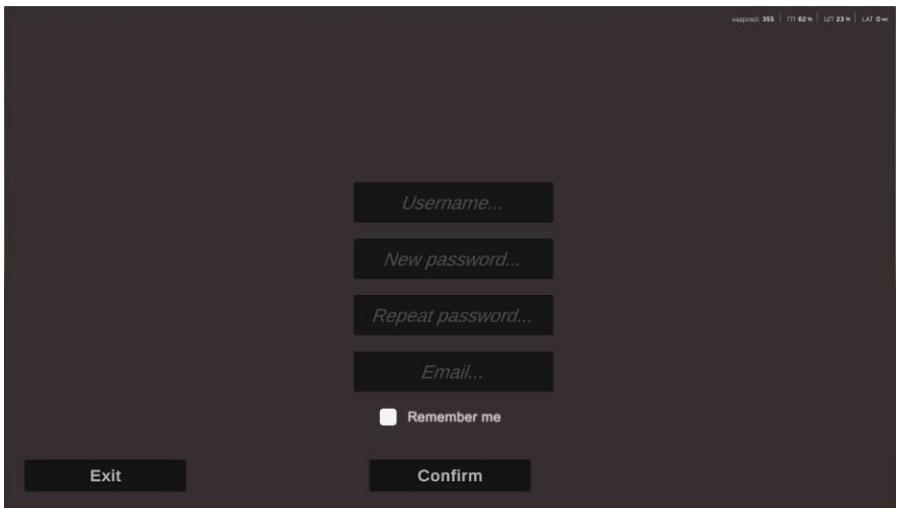

Рисунок 4.3 – Меню реєстрації

У разі будь-яких помилок (невірний формат email, занадто короткий пароль або невідповідність полів пароля) на відповідній формі з’являється повідомлення про причину відхилення. Після успішної реєстрації або входу користувач одразу потрапляє в головне меню гри. Усі облікові записи та токени сесії зберігаються в хмарному бекенді Firebase Authentication, а локально – лише налаштування «Запам’ятати мене» в PlayerPrefs і поточно активний токен, що дозволяє безпечний та прозорий для гравця процес повторного входу.

4.4 Підключення до матчу

Після авторизації гравець переходить до екрану «Start Game», де в текстове поле вводить назву кімнати (рисунок 4.4). Під полем розташовано дві кнопки: «Створити кімнату» та «Підключитися».

Натискання «Створити кімнату» викликає SessionConnectionHandler.StartGameAsync, який асинхронно створює нову сесію на сервері Photon Fusion із заданою назвою та параметрами (максимальна кількість гравців, режим гри). Якщо сесія створена успішно і клієнт має права авторитету над завантаженням сцени, Fusion автоматично завантажує ігровий рівень для всіх учасників. Натискання «Підключитися»

використовує той самий StartGameAsync, але клієнт приєднується до вже існуючої кімнати. Після встановлення з’єднання вікно «Start Game» закривається, і за допомогою вбудованого в Photon SceneManager відбувається синхронне завантаження ігрової сцени. Кожен гравець, незалежно від того, створив він кімнату чи приєднався до неї, потрапляє в ігрову сцену одночасно з іншими учасниками, що забезпечує узгоджений початок матчу.

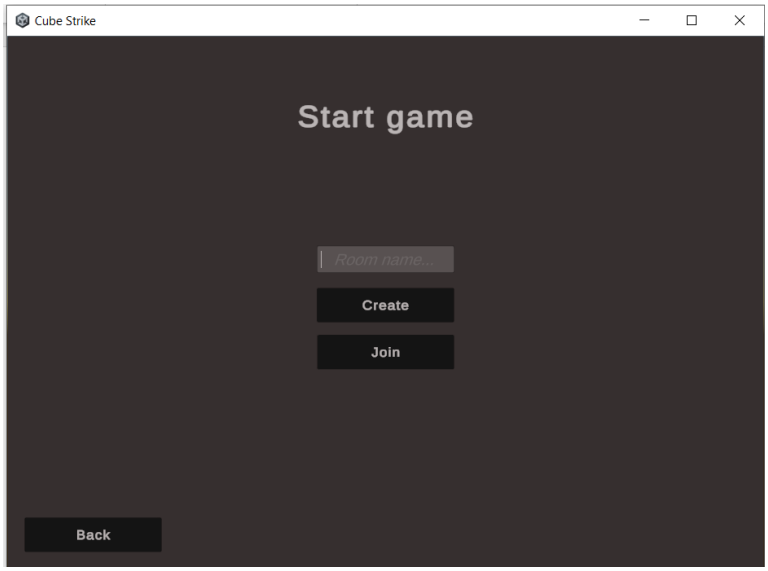

Рисунок 4.4 – Меню підключення до кімнат

4.5 Ігрове управління: HUD та ігровий чат

Після успішного підключення до матчу гравець потрапляє в ігрову зону (рисунок 4.5), де його чекає інтуїтивна система керування та повноцінний інтерфейс відображення стану персонажа й комунікації з іншими учасниками. Для пересування використовуються стандартні клавіші WASD (або аналогічні осі на ґеймпаді), утримання клавіші Shift активує біг, натискання пробілу – стрибок. Огляд навколишнього простору здійснюється рухом миші: зміщення курсору вліво/вправо повертає камеру по горизонталі, вгору/вниз – по вертикалі. Прицілювання плавно змінює швидкості обробки вводу миші, що дозволяє легко перейти від швидкого огляду до точного

цілепокладання.

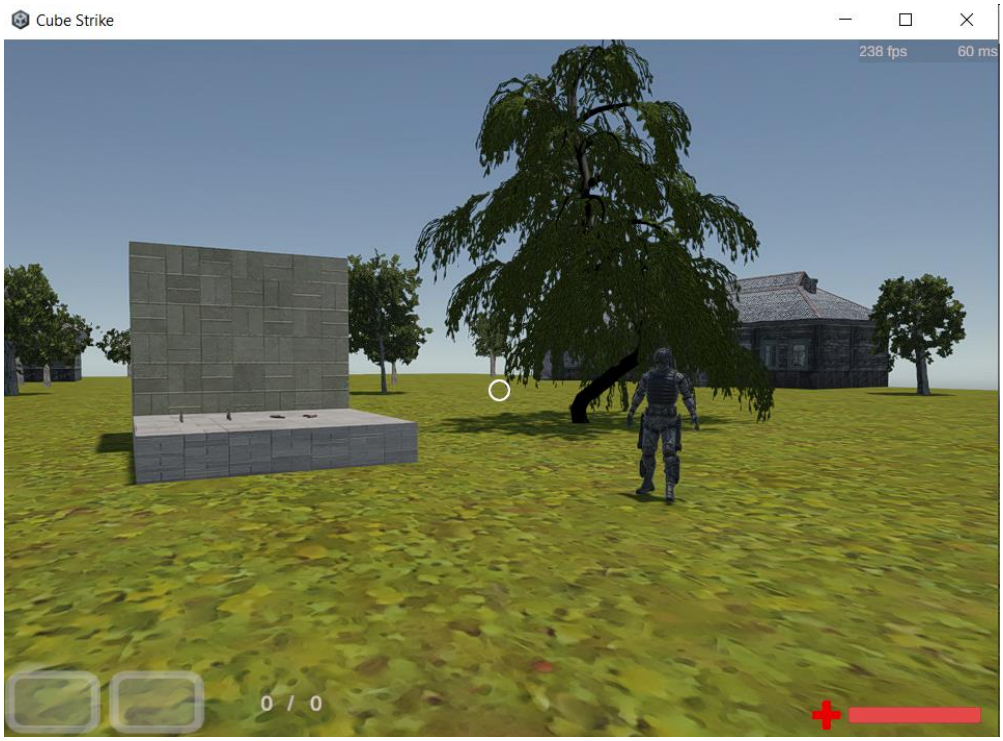

Рисунок 4.5 – Ігрова зона

Над грою висвічується HUD, до складу якого входить індикатор здоров’я (рисунок 4.6) у вигляді горизонтальної шкали, оновлення якої відбувається в реальному часі при отриманні ушкоджень чи лікуванні.

Рисунок 4.6 – Індикатор здоров’я

Поруч із ним розташовані дві цифрові панелі (рисунок 4.7), що показують залишок патронів у магазині й загальний запас. Будь-яка дія зі

зброєю одразу відображається на екрані.

Рисунок 4.7 – Цифрові панелі залишка патронів

Приціл завжди знаходиться в центрі екрану й лише підсвічується ефектом, коли гравець піднімає або змінює зброю.

Для текстової комунікації між гравцями реалізовано вбудований мультиплеєрний чат. Натискання клавіші T відкриває інтерфейс введення повідомлень (рисунок 4.8). Гравець може ввести текст і надіслати його, натиснувши Enter – повідомлення буде миттєво передано всім гравцям, які перебувають у тій самій кімнаті.

Всі повідомлення з’являються у вигляді вертикального списку в лівій частині екрану, автоматично прокручуються донизу та зберігають лише останні десять реплік, що запобігає перевантаженню інтерфейсу.

Для повернення в активний ігровий режим чат можна закрити, натиснувши клавішу F1. Це приховує інтерфейс чату, блокує курсор і дозволяє знову повноцінно керувати персонажем без відволікань.

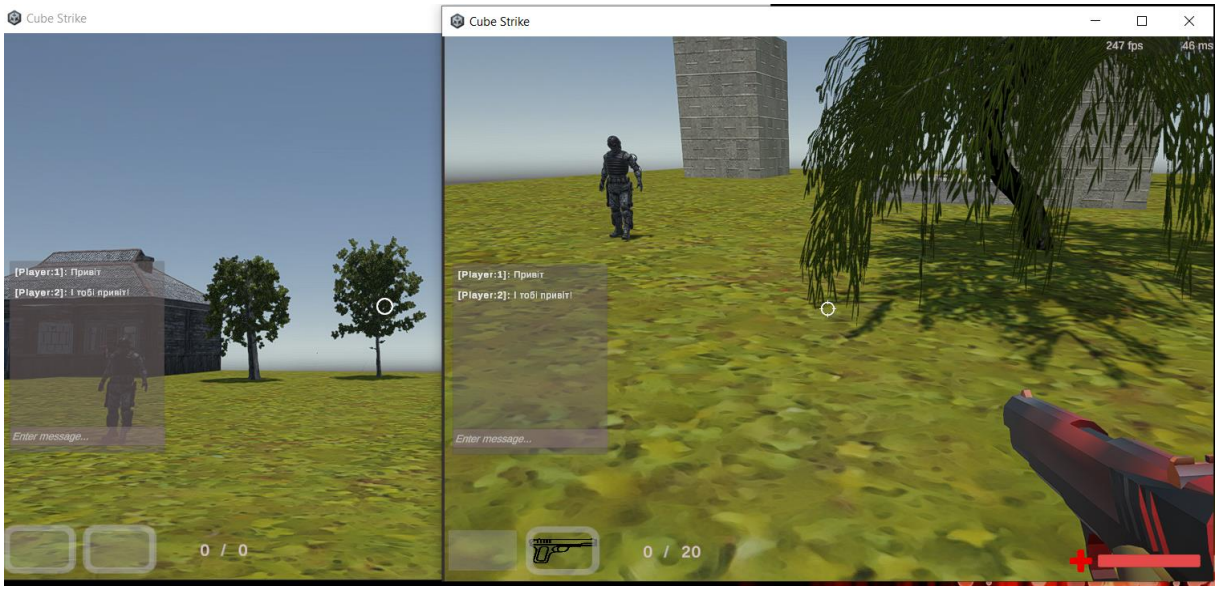

Рисунок 4.8 – Онлайн чат

Під час гри гравець може викликати внутрішньо ігрове меню натисканням клавіші Escape (рисунок 4.9). У цьому меню відображається кнопка «Leave», яка дозволяє залишити поточний матч і повернутися в головне меню гри.

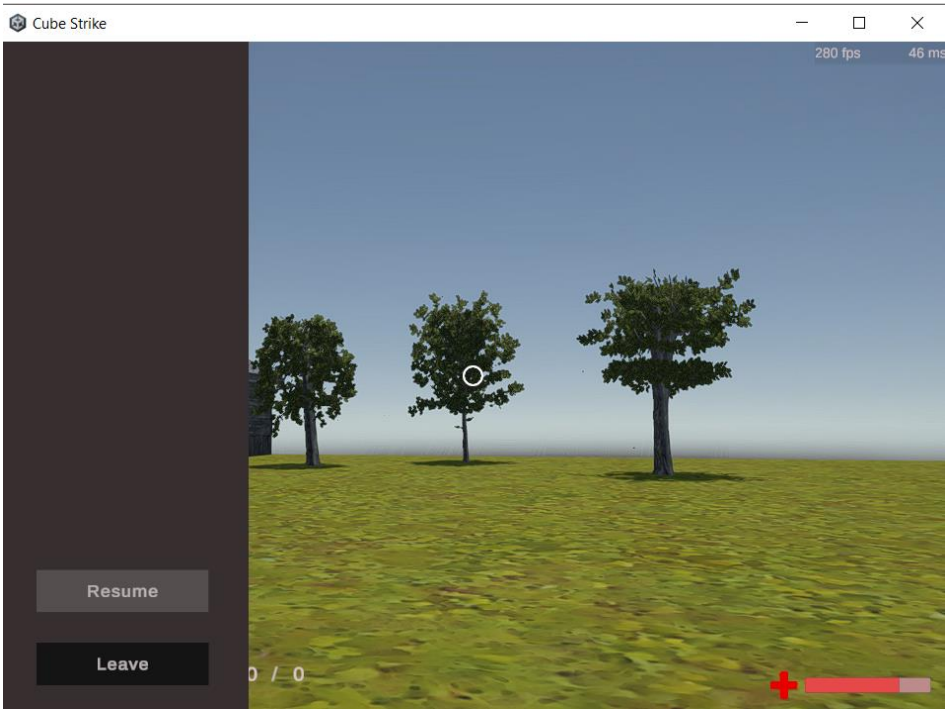

Рисунок 4.9 – Головне ігрове меню

Після натискання кнопки відбувається роз'єднання з кімнатою, очищення сесії гравця та завантаження стартового інтерфейсу. Меню створено з урахуванням зручності – воно накладається поверх гри та не вимагає повного виходу з програми для перезапуску матчу або зміни параметрів.

ВИСНОВКИ

У процесі дослідження було проаналізовано ключові технічні аспекти створення багатокористувацьких шутерів у середовищі Unity із застосуванням сучасних мережевих технологій. Основну увагу було зосереджено на вирішенні трьох основних викликів, притаманних онлайн-іграм: забезпеченні захисту від нечесної гри, мінімізації затримок під час взаємодії між клієнтами та забезпеченні стабільної масштабованості.

Запропонована архітектура, хоча й не передбачає повноцінної серверної обробки ігрової логіки, дозволяє досягти базового рівня верифікації дій клієнтів шляхом перевірки коректності критичних операцій з боку системи синхронізації станів. Це рішення дає змогу зменшити навантаження на інфраструктуру, зберігаючи при цьому цілісність ігрового процесу. Особлива увага приділялася методам компенсації затримок, зокрема механізмам клієнтського передбачення, інтерполяції та екстраполяції, що забезпечують плавну та реалістичну взаємодію між гравцями у реальному часі.

Використання Unity та Photon Fusion 2 продемонструвало ефективність поєднання гнучкої клієнтської логіки з потужними можливостями сучасної мережевої синхронізації. Результатом стало створення фундаменту для реалізації функціонального багатокористувацького шутера з повноцінним матчмейкінгом, інтерактивним інтерфейсом, базовими ігровими системами та інтеграцією з хмарними сервісами, такими як Firebase, для збереження користувацьких даних.

Загалом отримані результати підтверджують доцільність і практичну ефективність обраного підходу, який дозволяє розробляти конкурентоспроможні онлайн-ігри навіть без наявності повноцінного серверного бекенду. Це відкриває нові можливості для інді-розробників і команд із обмеженими ресурсами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Меррілл, Б. А. “Первісні мультиплеєрні шутери: від ‘Maze War’ до сучасності” // First-Person Shooter Games: A Brief History and Taxonomy, вип. 56, № 2, 2023. – С. 1–24. – DOI: https://doi.org/10.1007/978-3-031-23161-2_473.

2. Кулкарні, Н. Д., & Бансал, С. "Використання патерну Facade у практичному сценарії телефонного застосунку" // Journal of Artificial Intelligence & Cloud Computing, т. 1, № 1, 2022. – С. 2–5. – DOI: https://doi.org/10.47363/JAICC/2022(1)180.

3. Наумович, Г. "Використання патерну Observer для реалізації аналізу потоків даних" // ACM SIGSOFT Software Engineering Notes, т. 27, № 6, 2002. – DOI: https://doi.org/10.1145/634636.586107.

4. Ван, А. І., Ву, Б. "Використання розробки ігор для викладання архітектури програмного забезпечення" // International Journal of Computer Games Technology, 2011. – DOI: https://doi.org/10.1155/2011/920873.

5. Merrill, B. A. “First-Person Shooter Games: A Brief History and Taxonomy” // First-Person Shooter Games: A Brief History and Taxonomy, vol. 56, no. 2, 2023. – pp. 1–24. – DOI: https://doi.org/10.1007/978-3-031-23161-2_473.

6. Bui, V.-P., Pandey, S. R., Casparsen, A., Chiariotti, F., & Popovski, P. (2023). The Role of Game Networking in the Fusion of Physical and Digital Worlds through 6G Wireless Networks. arXiv. https://arxiv.org/abs/2302.01672.

7. Smith, J., & Lee, H. (2009). A Hybrid Architecture for Massively Multiplayer Online Games. ACM SIGCOMM Computer Communication Review, 39(4), 41–46. https://doi.org/10.1145/1517494.1517507.

8. Colwell, C., & Davis, T. (2015). Machine Learning and Semantic Analysis for Cyberbullying Detection in Online Gaming Chats. Proceedings of the ACM Symposium on Applied Computing, 123–130. https://doi.org/10.1145/1230040.1230058.

9. Wang, Y., & Zhao, Q. (2010). Adaptive Client-Server Architecture for Mobile Multiplayer Games. Proceedings of SIMUTOOLS, Article 87. https://doi.org/10.4108/ICST.SIMUTOOLS2010.8704.

10. Mirror Networking Documentation: https://mirror-networking.gitbook.io/docs/manual/guides/synchronization/syncvars.

11. Unity Technologies. “Netcode for GameObjects: High-Level Networking Library for Unity” // Unity Multiplayer Documentation, 2025. – URL: https://docs-multiplayer.unity3d.com/netcode/current/about/

12. Lee, N. “Unity, A 2D and 3D Game Engine” // Encyclopedia of Computer Graphics and Games, 2023. – DOI: https://doi.org/10.1007/978-3-319-08234-9_536-1.

13. Moll, P., Isak, S., Hellwagner, H., & Burke, J. (2020). A Quadtree-based synchronization protocol for inter-server game state synchronization. Computer Networks, 185, 107723. https://doi.org/10.1016/j.comnet.2020.107723.

14. Dantas, A., & Baquero, C. (2025). CRDT-Based Game State Synchronization in Peer-to-Peer VR. Proceedings of the 12th Workshop on Principles and Practice of Consistency for Distributed Data, Rotterdam, Netherlands. https://doi.org/10.48550/arXiv.2503.17826.

15. Milojković, K., Živković, M., & Bačanin, N. (2024). Agile Multi-user Android Application Development With Firebase: Authentication, Authorization, and Profile Management. In Sinteza 2024 (pp. 405–412). DOI: https://doi.org/10.15308/Sinteza-2024-405-412.

16. Booth, J., & Ivanov, V. (2020). Realistic Physics Based Character Controller. arXiv. https://doi.org/10.48550/arXiv.2006.07508.

17. Colwell, A. M., & Glavin, F. G. (2018). Colwell's Castle Defence: A Custom Game Using Dynamic Difficulty Adjustment to Increase Player Enjoyment. arXiv. https://arxiv.org/abs/1806.04471.

18. Fagerholt, E., & Lorentzon, M. (2009). Beyond the HUD - User Interfaces for Increased Player Immersion in FPS Games. Retrieved from https://odr.chalmers.se/items/d5fe6889-4cc6-49c2-ba56-0d759e2f37eb.

19. Boroń, M., Brzeziński, J., & Kobusińska, A. (2020). P2P matchmaking solution for online games. Peer-to-Peer Networking and Applications, 13(1), 137–150. https://doi.org/10.1007/s12083-019-00725-3.

20. Bauer, D., Iliadis, I., Rooney, S., & Scotton, P. (2004). Communication Architectures for Massive Multi-Player Games. Multimedia Tools. https://doi.org/10.1023/B%3AMTAP.0000026841.97579.1f.

21. Проект та білд гри: https://drive.google.com/drive/folders/1Zy4Q6HbI4jLtQxoqI0kICjYNAKJGsfjO?usp=sharing.