

Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття ступеня вищої освіти бакалавра
зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка клієнт-серверної архітектури та системи організації ігрових сесій для 3D-шутера в середовищі Unity

Засвідчую, що в цій
кваліфікаційній роботі немає
запозичень із праць інших авторів
без відповідних посилань.

Студент гр. ПЗ-22-2

_____ / Д. А. Бабенко /

Керівник

кваліфікаційної роботи

В. о. зав. кафедри

_____ / Д. В. Швець /

_____ / А. М. Стрюк /

Кривий ріг

2026

Криворізький національний університет
Факультет: Інформаційних технологій
Кафедра: Моделювання та програмного забезпечення
Ступінь вищої освіти: бакалавр
Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

В.о. зав. кафедри

_____ А. М. Стрюк

«___» _____ 2026 р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ-22-2 Бабенко Дмитру Андрійовичу

1. Тема: Розробка клієнт-серверної архітектури та системи організації ігрових сесій для 3D-шутера в середовищі Unity затверджено наказом по КНУ № 285с від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «19» червня 2026 р.
3. Вихідні дані по роботі: розроблювана система повинна використовувати клієнт-серверну архітектуру; реалізувати систему лобі для динамічної організації ігрових сесій; використовувати сервіси Unity Relay для забезпечення зв'язку між гравцями без виділеного сервера; реалізувати систему управління зброєю та мережеву реплікацію дій гравців.
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): огляд технологій мережевої взаємодії в Unity; вибір середовища Unity, мови C#, протоколів передачі даних, архітектури ігрових сесій; розробка клієнт-серверної архітектури, проектування мережевих компонентів, організація ієрархії об'єктів; розробка алгоритмів створення та приєднання до лобі, інтеграція, система вибору зброї та її синхронізації між клієнтами; проведення функціонального тестування ігрових механік та навантажувальне тестування стабільності мережевого з'єднання.
5. Перелік ілюстративного матеріалу: схема взаємодії клієнтів з Unity Relay; діаграма класів мережевих менеджерів та ігрових сутностей; блок-схеми життєвого циклу ігрової сесії; знімки екранних форм меню, лобі та інтерфейсу, а також візуалізація мережевої синхронізації об'єктів у Unity.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Аналіз архітектур мережевих ігор та сервісів Unity Gaming Services	01.12.2025 – 15.12.2025
2	Обґрунтування технологічного стеку та проєктування архітектури взаємодії	16.12.2025 – 30.12.2025
3	Проєктування логіки серверної частини (Relay, Lobby) та об'єктної моделі гравця	05.01.2026 – 20.01.2026
4	Підготовка матеріалів першого розділу роботи	21.01.2026 – 05.02.2026
5	Програмна реалізація мережевого модуля (Netcode, RPC, NetworkVariable)	06.02.2026 – 25.02.2026
6	Підготовка матеріалів другого розділу роботи	26.02.2026 – 10.03.2026
7	Реалізація ігрових механік	11.03.2026 – 20.04.2026
8	Підготовка матеріалів третього розділу роботи	21.04.2026 – 10.05.2026
9	Функціональне та стрес-тестування мережевої взаємодії	12.05.2026 – 30.05.2026
10	Оформлення пояснювальної записки та підготовка презентаційних матеріалів до захисту	01.06.2026 – 19.06.2026

Дата видачі завдання:

«30» листопада 2025 р.

Студент

_____ / Д. А. Бабенко /

Керівник роботи

_____ / Д. В. Швець /

РЕФЕРАТ

3D-ШУТЕР, UNITY, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, ЛОБІ, RELAY, NETCODE, МЕРЕЖЕВА СИНХРОНІЗАЦІЯ.

Пояснювальна записка: 77 с., 19 рис., 1 дод., 10 джерел.

Об'єктом розробки є клієнт-серверна архітектура та система організації ігрових сесій для 3D-шутера в середовищі Unity. Метою кваліфікаційної роботи є створення стабільного мережевого середовища, що забезпечує динамічне створення ігрових лобі, підключення гравців через Relay-сервіси та синхронізацію стану ігрового світу в реальному часі за умов мережевих затримок.

У роботі проведено аналіз сучасних підходів до мережевої взаємодії в іграх, обґрунтовано вибір інструментарію: Unity Engine, мова програмування C#, фреймворк Netcode for GameObjects. Розроблено клієнт-серверну архітектуру, де серверна логіка реалізована через сервіси Unity Gaming Services (Lobby, Relay), що дозволяє уникнути потреби у виділених серверах. До основних конструктивних та техніко-експлуатаційних характеристик системи належать: система автоматичного створення та пошуку лобі, механізм синхронізації гравців через Unity Relay, інтегрована система управління зброєю з мережевою реплікацією та інтерактивний інтерфейс (HUD) для відображення ігрового стану. Галузь застосування — розробка багатокористувацьких 3D-проектів із використанням високорівневих мережевих абстракцій.

Виконано функціональне тестування мережевих механік, зокрема перевірку стабільності підключення, обробку розривів з'єднання та синхронізацію дій гравців у режимі дуелі (1 на 1). Зроблено висновок, що розроблена архітектура забезпечує необхідний рівень продуктивності для ігрових сесій такого типу та володіє потенціалом для подальшого масштабування до більшої кількості учасників.

ABSTRACT

VOLUNTEER ACTIVITY, MOBILE APPLICATION, ANDROID, KOTLIN, MVVM, FIRESTORE, GEOLOCATION, KARMA SYSTEM, OFFLINE MODE.

Explanatory note: 77 p., 19 figures, 1 app., 10 sources.

The object of development is a client-server architecture and a system for organizing game sessions for a 3D shooter in the Unity environment. The method of qualified work is the creation of a stable core that will ensure the dynamic creation of game lobbies, connecting gravers through Relay services and synchronizing the state of the game world in real life hours for the minds of the hemlocks.

The robot analyzed the current approaches to limiting interaction in games, and used a selection of tools: Unity Engine, C# language programming, Netcode for GameObjects framework. A client-server architecture has been developed, and the server logic is implemented through Unity Gaming Services (Lobby, Relay), which allows the needs of the various servers to be unique.

The main design and technical-operational characteristics of the system include: a system of automatic creation and search of the forehead, a mechanism for synchronizing engines via Unity Relay, an integrated fault management system with fencing replication and interactive Interface (HUD) for displaying the game screen. Galuz zastosuvannya - development of rich 3D projects from high-quality edge abstractions.

Functional testing of the boundary mechanics has been completed, including checking the stability of the connection, processing connection breakdowns and synchronizing the actions of gravers in the duel mode (1 on 1). It has been completed that the fragmented architecture will provide the necessary level of productivity for gaming sessions of this type and has the potential for further scaling to a larger number of participants.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ

API – Application Programming Interface (інтерфейс прикладної програми).

HUD – Heads-Up Display (інтерфейс ігрового екрана).

IDE – Integrated Development Environment (інтегроване середовище розробки).

IK – Inverse Kinematics (інверсна кінематика).

NAT – Network Address Translation (трансляція мережевих адрес).

RPC – Remote Procedure Call (віддалений виклик процедур).

UI – User Interface (інтерфейс користувача).

Unity Relay – хмарна служба Unity для забезпечення зв'язку між гравцями без виділеного сервера.

ПЗ – програмне забезпечення.

ЗМІСТ

РЕФЕРАТ.....	4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ	6
ВСТУП.....	8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ФОРМУВАННЯ ВИМОГ	10
1.1 Еволюція та проблематика ігрових сесій у 3D-шутерах	10
1.2 Аналіз існуючих інструментів координації ігрових сесій.....	10
1.3 Моделювання ролей гравців та сценаріїв використання.....	11
1.4 Формування функціональних та нефункціональних вимог проєкту ...	13
2 ОБҐРУНТУВАННЯ ТЕХНОЛОГІЧНОГО СТЕКУ ТА	
ПРОЄКТУВАННЯ АРХІТЕКТУРИ.....	14
2.1 Вибір мови програмування C# та рушія Unity	14
2.2 Оптимізація ігрового інтерфейсу та асинхронна модель обробки даних	15
2.3 Гібридна модель зберігання даних: Unity Cloud та локальний кеш	16
2.4 Архітектурна модель та концепція управління мережевим станом	18
3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ІГРОВОГО ПРОЄКТУ	21
3.1 Модуль автентифікації користувачів	21
3.2 Модуль створення та пошуку ігрових сесій.....	24
3.3 Модуль керування ігровою логікою та раундами.....	27
3.4 Модуль бойової системи та візуалізації мережевої взаємодії	28
4 ТЕСТУВАННЯ ТА ОПТИМІЗАЦІЯ МЕРЕЖЕВОЇ СИСТЕМИ..	33
4.1 Тестування продуктивності та стабільності ігрового клієнта.....	33
4.2 Тестування мережевої синхронізації та стабільності	34
4.3 Оптимізація продуктивності	34
ВИСНОВКИ.....	36
ПЕРЕЛІК ПОСИЛАНЬ	38
Додаток А – Програмний код.....	40

ВСТУП

Розвиток багатокористувацьких відеоігор за останні роки суттєво змінився за масштабом і технічною складністю. Сучасні гравці очікують від ігрового процесу миттєвої реакції, надійної синхронізації та можливості легко приєднатися до сесії в будь-який момент. Проте розробка мережевої складової залишається одним із найскладніших етапів створення ігор, особливо для невеликих команд, які часто змушені використовувати інструменти, що вимагають дороговартісної серверної інфраструктури або надто складні в налаштуванні.

Практичний досвід розробки багатокористувацьких шутерів наочно демонструє головну проблему: реалізація клієнт-серверної архітектури "з нуля" потребує вирішення конфліктів стану ігрового світу, боротьби з мережевими затримками та забезпечення безпеки ігрових даних. Існуючі рішення, такі як виділені сервери, є недосяжними для інді-розробників через високу вартість обслуговування, а використання примітивних Peer-to-Peer з'єднань призводить до вразливості до читерства та проблем із NAT у гравців.

Окремою проблемою є синхронізація ігрового світу в умовах різного рівня стабільності інтернет-з'єднання. Механіки управління зброєю, переміщення персонажів та фізика об'єктів мають залишатися ідентичними для всіх учасників сесії, незалежно від їхнього місця розташування чи якості зв'язку. Навчання розробників налаштуванню складних мережевих систем у реальному часі є тривалим процесом, що потребує розуміння концепцій серверної авторизації, реплікації та прогнозування дій гравця.

Аналіз ринку спеціалізованих інструментів показав, що Unity Gaming Services, зокрема сервіси Relay та Lobby, надають потужні можливості для розробки, але їх впровадження у проєкти з клієнт-серверною архітектурою потребує ретельного проєктування та інтеграції з Netcode for GameObjects.

Відповіддю на ці проблеми став проєкт багатокористувацького 3D-шутера – програмне середовище для Unity з оптимізованою клієнт-серверною логікою.

Центральна ідея — автоматизувати повний цикл ігрової сесії від створення лобі до стабільного синхронізованого ігрового процесу, зберігаючи при цьому високу швидкість обробки мережевих подій.

Метою роботи є проектування, реалізація та тестування клієнт-серверної архітектури для 3D-шутера в середовищі Unity, що забезпечує організацію ігрових сесій, ефективне керування лобі та синхронізацію стану ігрових об'єктів між гравцями.

Об'єктом дослідження є процес розробки мережевого програмного забезпечення для багатокористувацьких ігор. Предметом дослідження є сучасні методи мережевої взаємодії в Unity, протоколи передачі даних, механізми Relay-зв'язку, алгоритми реплікації ігрових станів та принципи роботи Lobby API.

Завдання роботи:

- проаналізувати існуючі підходи до побудови мережевої архітектури в іграх та обмеження існуючих сервісів для організації сесій.
- обґрунтувати вибір технологічного стеку та інструментів Unity Gaming Services для реалізації багатокористувацького режиму.
- спроектувати структуру лобі-системи та механізми безпечного підключення гравців через Relay.
- розробити алгоритми синхронізації ігрових механік з використанням Netcode for GameObjects.
- створити мережеву модель управління ігровою сесією та системою вибору зброї.
- реалізувати механізми коректного виходу гравців та видалення неактивних ігрових сесій.
- провести функціональне та стрес-тестування мережевої взаємодії шляхом симуляції ігрових сесій для виявлення затримок та конфліктів стану.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ФОРМУВАННЯ ВИМОГ

1.1 Еволюція та проблематика ігрових сесій у 3D-шутерах

Ігрова індустрія за останні роки зазнала значних трансформацій. Багатокористувацькі ігри перестали бути просто коридорними шутерами, перетворившись на складні інтерактивні екосистеми з безперервною синхронізацією об'єктів, динамічною зміною оточення та високими вимогами до часу відгуку. Щодня тисячі гравців шукають можливість провести ігрову сесію, і переважна більшість розробників-початківців досі стикається з труднощами при реалізації мережевої логіки, використовуючи застарілі або неоптимізовані інструменти, які не забезпечують стабільної роботи в мережі.

Особливо гостро ця проблема відчувається в динамічних шутерах. Гравці очікують від гри плавності рухів, миттєвої реєстрації влучань та відсутності затримок при взаємодії з іншими об'єктами. Якщо гравець витрачає час на боротьбу з "лагами" або некоректну синхронізацію позицій суперника, це призводить до повної втрати інтересу до проєкту. Головна технічна проблема — відсутність надійної системи організації ігрових сесій. В іграх без централізованого управління лобі гравці часто не можуть знайти один одного, сесії розпадаються через помилки підключення, а дублювання пакетів або "стрибки" гравців по карті руйнують ігровий досвід.

1.2 Аналіз існуючих інструментів координації ігрових сесій

Перш ніж проєктувати власне рішення, було проведено аналіз наявних інструментів для побудови мережевої архітектури в Unity. Метою аналізу стало виявлення функціональних прогалин у підходах до організації сесій (Lobby та Relay).

Використання виділених серверів (Dedicated Servers) забезпечує високу надійність, проте для більшості проєктів це надмірно дорогий та технічно

складний шлях. На противагу цьому, архітектура "Host-Client" є доступною, але без належної реалізації Relay-сервісів вона стає безпорадною перед проблемами NAT-трансляції, через що гравці з різних мереж не можуть з'єднатися між собою.

Існуючі мережеві бібліотеки для Unity (наприклад, Mirror або старі версії UNet) часто вимагають глибоких знань мережевого програмування, що стає бар'єром для розробників. Мобільна версія багатьох подібних рішень не підтримує ефективну синхронізацію станів, що призводить до десинхронізації об'єктів зброї чи персонажів. Проєкт Unity Gaming Services (Lobby, Relay, Netcode) пропонує сучасний підхід, проте його інтеграція потребує чіткої логіки обробки станів сесії. Підсумок аналізу: необхідна власна спеціалізована система, яка поєднує стабільність підключення через Relay, зручне керування лобі та автоматизований життєвий цикл сесії.

1.3 Моделювання ролей гравців та сценаріїв використання

При проєктуванні архітектури ігрових сесій важливо чітко визначити ролі учасників. У розроблюваній системі закладено три ролі: Хост, Клієнт та Сервіс (Unity Cloud).

Хост – це гравець, який ініціює створення ігрової сесії. Він відповідає за запуск Relay-алокції, створення запису в Lobby API та керування сценою. Хост проєктувався з урахуванням мінімізації навантаження на систему: як тільки лобі створено, воно автоматично стає доступним у загальному списку.

Клієнт – це гравець, який здійснює пошук активних сесій та приєднується до них. Сценарій для цієї ролі базується на простоті: гравець натискає кнопку оновлення списку, вибирає доступну сесію і через Relay-код встановлює з'єднання. Система автоматично перевіряє доступність лобі та наявність вільних місць.

Адміністратор (Сервіс) – це Unity Gaming Services, що модерує життєвий цикл сесій у хмарі.

Роль хоста та логіка синхронізації в процесі розробки суттєво

уточнювались порівняно з початковим планом. На етапі тестування виявилась технічна проблема: при приєднанні клієнта до хоста не відбувалася коректна синхронізація обраної гравцем зброї. В результаті клієнт автоматично отримував початкову зброю, ігноруючи індивідуальний вибір гравця в меню. Це створювало дискомфорт, оскільки гравець втрачав можливість використання обраної зброї під час матчу.

Для вирішення цієї проблеми було переглянуто механізм ініціалізації гравця під час мережевого спавну. Впроваджено передачу параметрів обраного спорядження безпосередньо в момент підключення клієнта до мережевої сесії. З боку хоста реалізовано механізм перевірки наявності даних про обрану зброю, що дозволило системі автоматично підхоплювати вибір гравця, замість примусового встановлення початкових значень. Завдяки цьому інформація про обрану зброю стала коректно передаватися від клієнта до хоста ще на етапі підключення, забезпечуючи ідентичність озброєння гравця в усіх учасників сесії.

Взаємодію між усіма учасниками та сервісами відображено в діаграмі прецедентів (див. рис. 1.1).

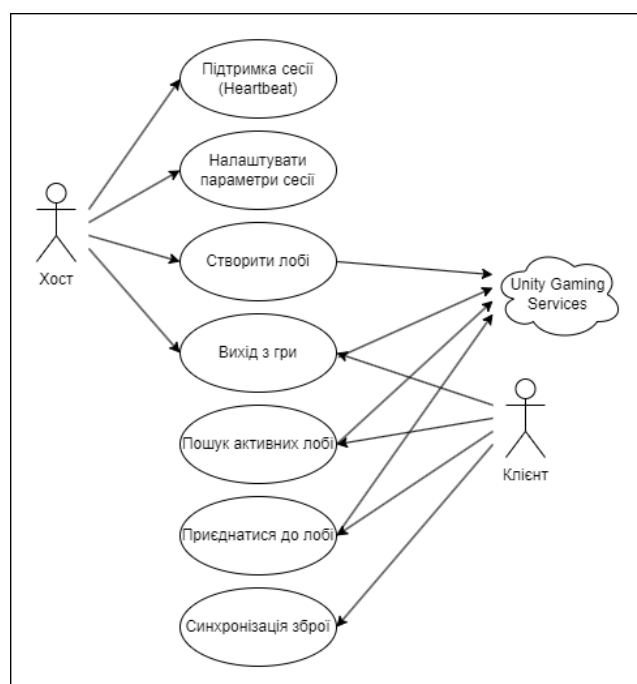


Рисунок 1.1 – Діаграма прецедентів системи організації ігрових сесій

1.4 Формування функціональних та нефункціональних вимог

Після аналізу сценаріїв використання системи настав час перетворити знання про проєкт на конкретні технічні вимоги, де концепція багатокористувацького 3D-шутера трансформується у перелік чітких і вимірюваних характеристик. Функціональні вимоги описують поведінку ігрової системи та передбачають, що система повинна забезпечувати створення ігрового лобі з унікальним кодом для підключення через Relay-сервіси, що усуває потребу в наявності публічної IP-адреси у хоста. Стан усіх об'єктів, таких як позиції гравців, стан здоров'я та кількість набоїв, має оновлюватися в реальному часі за допомогою механізму мережових змінних. Кожна дія, зокрема постріл, переміщення або зміна зброї, має оброблятися через систему віддалених викликів процедур, що забезпечує коректну передачу команд від клієнта до сервера-хоста.

Нефункціональні вимоги визначають якісні характеристики системи, де найважливішою є мінімальна мережева затримка, оскільки архітектура повинна підтримувати високу швидкість оновлення ігрового стану для мінімізації впливу пінгу на реєстрацію подій. В умовах короткочасних розривів зв'язку система зобов'язана коректно обробляти відключення клієнта, не призводячи до аварійного завершення ігрового процесу для хоста. Хоча поточна реалізація орієнтована на формат 1 на 1, структура проєкту базується на Netcode, що дозволяє потенційно масштабувати кількість гравців у лобі без кардинальної зміни мережевого коду. Реплікація даних має бути обмежена лише критично важливими змінами стану об'єктів для оптимізації трафіку та забезпечення стабільної роботи навіть при обмеженій пропускній здатності. Код проєкту має базуватися на стандартних компонентах Unity, що гарантує зручність подальшого розширення без необхідності переписування мережевої логіки, а безпека сесій реалізується шляхом обмеження доступу до керування лобі лише для авторизованих користувачів, що запобігає спробам втручання в ігрову сесію третіми особами.

2 ОБҐРУНТУВАННЯ ТЕХНОЛОГІЧНОГО СТЕКУ ТА ПРОЄКТУВАННЯ АРХІТЕКТУРИ

2.1 Вибір мови програмування C# та рушія Unity

Розробка 3D-шутера з мережевою архітектурою вимагає ретельного підходу до вибору технологічного стеку, здатного забезпечити високу продуктивність і надійну роботу в умовах постійного обміну даними між гравцями. Як базовий інструментарій було обрано ігровий рушій Unity та мову програмування C#. Вибір Unity зумовлений його гнучким набором інструментів для роботи з 3D-графікою, фізичними взаємодіями та інтегрованою підтримкою сучасних мережевих сервісів (Unity Gaming Services), що критично важливо для швидкого прототипування та масштабування багатокористувацьких проєктів.

На відміну від інтерпретованих мов або засобів візуального програмування, C# забезпечує високу швидкість виконання коду завдяки статичній типізації та компіляції в машинний код. В ігровій індустрії обробка кожного кадру (Update-цикл) є критичною операцією: будь-яка затримка при обробці вхідних пакетів від гравців або обрахунку фізики призводить до десинхронізації стану. C# дозволяє ефективно керувати пам'яттю та виконувати складні обчислення, необхідні для реєстрації влучань та інтерполяції позицій об'єктів.

Вирішальним чинником стала розвинута екосистема Unity для розробки мережевих застосунків. Сучасна бібліотека Netcode for GameObjects базується на асинхронних патернах, що дозволяє делегувати складні обчислення, такі як серіалізація мережевих пакетів або завантаження асетів, на фонові потоки. Це запобігає блокуванню головного потоку, відповідального за рендеринг графіки, що гарантує стабільну частоту кадрів для гравця.

Основним середовищем розробки став Unity Editor у поєднанні з IDE Microsoft Visual Studio.

2.2 Оптимізація ігрового інтерфейсу та асинхронна модель обробки даних

Створення ігрового інтерфейсу (UI) для багатокористувацького шутера вимагає підходу, який мінімізує навантаження на головний потік виконання. В іграх будь-яка затримка при оновленні UI, наприклад, при виборі зброї в лобі, може призвести до завмирання гри на частку секунди, що є неприпустимим під час ігрового процесу. Для побудови інтерфейсу було обрано систему Unity UI у поєднанні з Event System, що дозволяє створювати чуйні та інформативні елементи, такі як HUD (Heads-Up Display) та меню лобі.

Основним інструментом оптимізації роботи з мережею стала асинхронна модель програмування Async/Await. В Unity основний потік виконання відповідає за рендеринг графіки та фізичні обрахунки, тому будь-яка мережева операція, наприклад, запит до LobbyService або RelayService, за замовчуванням може блокувати гру до отримання відповіді від сервера. Використання асинхронних завдань дозволяє виконувати мережеві запити у фоновому режимі, не перериваючи ігровий цикл.

На практиці це вирішує проблему зависань при спробі підключення до лобі: гравець продовжує бачити анімований інтерфейс, тоді як фоновий процес очікує підтвердження від хмарного сервісу. Завдяки цьому інтерфейс залишається чуйним незалежно від швидкості відповіді серверів Unity Gaming Services.

Наступний код демонструє ізоляцію мережевих запитів від головного потоку ігрового рушія. Використання ключового слова `await` дозволяє системі призупинити виконання лише конкретного методу логіки лобі, залишаючи при цьому решту системи (рендеринг, обробку вводу) повністю працездатною. Переключення контексту на системні потоки гарантує, що клієнт залишається керованим для гравця незалежно від затримки при синхронізації з хостом або хмарним сервером.

Приклад використання асинхронного виклику для підключення до ігрової сесії наведено на рисунку 2.1.

```
private async void JoinLobbyById(string lobbyId)
{
    await ShutdownLobbySystem();
    await EnsureNetworkManagerExists();

    UpdateStatusText("Joining...");
    try
    {
        joinedLobby = await LobbyService.Instance.JoinLobbyByIdAsync(lobbyId);
        string relayCode = joinedLobby.Data[KEY_JOIN_CODE].Value;
        JoinAllocation joinAllocation = await RelayService.Instance.JoinAllocationAsync(relayCode);

        NetworkManager.Singleton.GetComponent<UnityTransport>().SetClientRelayData(
            joinAllocation.RelayServer.IpV4, (ushort)joinAllocation.RelayServer.Port,
            joinAllocation.AllocationIdBytes, joinAllocation.Key, joinAllocation.ConnectionData, joinAllocation.HostConnectionData
        );

        if (NetworkManager.Singleton.StartClient()) UpdateStatusText("Connected!");
    }
    catch (System.Exception e)
    {
        Debug.LogError("Join Error: " + e.Message);
        UpdateStatusText("Join Failed");
    }
}
```

Рисунок 2.1 Фрагмент коду асинхронного виклику для підключення до Relay-сесії

2.3 Гібридна модель зберігання даних: Unity Cloud та локальний кеш

Під час розробки одним із найскладніших завдань стала організація зберігання та синхронізації даних сесії. Специфіка мережевого шутера вимагає миттєвого оновлення стану світу (позиції гравців, вибір зброї) та, водночас, високої відмовостійкості до мережевих затримок. Класичний запит-відповідь HTTP є неефективним для ігрових цілей через високі затримки, тому було обрано гібридну модель роботи з даними.

Для синхронізації станів у реальному часі використовуються сервіси Unity Gaming Services (Lobby та Relay). Lobby API виступає як хмарний реєстр, що зберігає метадані активних ігрових сесій. Це дозволяє клієнтам ефективно шукати доступні сесії без необхідності опитування всіх активних хостів.

Для забезпечення стійкості до розривів зв'язку побудовано архітектуру локального кешування на базі структурованих об'єктів C#. Локальний кеш

зберігає копію останнього відомого стану лобі та конфігурацію гравця (Loadout). Принцип роботи полягає в наступному: коли гравець вибирає зброю в меню лобі, дані спочатку записуються в локальний об'єкт кешу, а потім синхронізуються через `NetworkVariable`. Якщо мережеве з'єднання на мить переривається, гравець не втрачає свій вибір, оскільки система автоматично повторює запит на синхронізацію після відновлення зв'язку.

Для забезпечення персистентності вибору гравця в умовах завершення ігрової сесії було реалізовано механізм локального збереження налаштувань за допомогою `PlayerPrefs`. Це дозволяє системі відновлювати стан вибору зброї при повторному запуску застосунку без необхідності звернення до хмарних сервісів.

Фрагмент реалізації логіки збереження та завантаження наведено на рисунку 2.2.

```
public static void SaveSelection(string weaponName)
{
    SelectedWeaponName = weaponName;
    PlayerPrefs.SetString("SelectedWeapon", weaponName);
    PlayerPrefs.Save();
}

Ссылка: 2
public static void LoadSelection()
{
    SelectedWeaponName = PlayerPrefs.GetString("SelectedWeapon", AK47);
}
```

Рисунок 2.2 – Фрагмент коду модуля збереження та завантаження налаштувань гравця

Код демонструє використання статичного методу для оновлення поточної змінної `SelectedWeaponName` та синхронний запис обраного значення у локальне сховище пристрою через `PlayerPrefs.Save()`. Такий підхід гарантує цілісність даних при виході з гри.

2.4 Архітектурна модель та концепція управління мережевим станом

Щоб уникнути перетворення програмного коду на хаотичну структуру, у проєкті було застосовано принцип модульної архітектури, яка чітко розподіляє обов'язки між окремими компонентами системи. Логіку ігрового проєкту було розділено на три функціональні шари, що відповідають структурі каталогів у проєкті:

Шар View (Assets/Scenes, Assets/Prefabs) відповідає виключно за візуальне відображення ігрової сцени та UI-елементів. Він не приймає самостійних рішень щодо мережевих змін, а лише реагує на події, отримані від менеджерів.

Шар Controller (Assets/Scripts) є ядром системи. Логіка кожного модуля строго ізольована: підпапка Auth керує автентифікацією гравця через Unity Services, Lobby відповідає за життєвий цикл ігрових сесій та роботу з Relay, а Gameplay забезпечує синхронізацію безпосередньо ігрових механік. Це дозволяє незалежно розширювати кожен підсистему без ризику виникнення конфліктів у мережевому коді.

Шар Model (Assets/Data) містить чисті структури даних та конфігурації, зокрема Weapons. Винесення даних у окремий шар гарантує, що ігрова логіка оперує абстрактними налаштуваннями, що спрощує додавання нових типів озброєння.

Важливим архітектурним рішенням став принцип «єдиного джерела істини» на стороні хоста. Клієнти ніколи не змінюють ігрові змінні напряму, вони надсилають запит (RPC), який обробляється відповідним контролером. Хост валідує дію та оновлює NetworkVariable, після чого стан автоматично розсилається іншим гравцям. Оскільки клієнтські контролери підписані на оновлення цих змінних, ігровий стан оновлюється автоматично, забезпечуючи плавну синхронізацію.

Логіку синхронізації, що реалізує концепцію централізованого управління мережевим станом через серверні процедури (RPC), наведено на рисунку 2.3.

```
[Rpc(SendTo.Server)]
Ссылка: 1
private void ShootServerRpc(Vector3 pos, Quaternion rot)
{
    if (activeWeapon == null || currentAmmo.Value <= 0) return;

    PlayWeaponSoundRpc(0);
    GameObject bullet = Instantiate(activeWeapon.weaponData.bulletPrefab, pos, rot);

    Bullet bulletScript = bullet.GetComponent<Bullet>();
    if (bulletScript != null) bulletScript.damage = activeWeapon.weaponData.damage;

    Collider bulletCollider = bullet.GetComponent<Collider>();
    if (bulletCollider != null)
    {
        foreach (Collider col in GetComponentsInChildren<Collider>())
            Physics.IgnoreCollision(bulletCollider, col);
    }

    bullet.GetComponent<NetworkObject>().Spawn(true);

    Rigidbody rb = bullet.GetComponent<Rigidbody>();
    if (rb != null) rb.linearVelocity = (rot * Vector3.forward) * 30f;

    PlayRecoilClientRpc();
    currentAmmo.Value--;
    UpdateAmmoClientRpc(currentAmmo.Value, reserveAmmo.Value);
}
```

Рисунок 2.3 – Фрагмент логіки мережевої синхронізації пострілу на стороні сервера

Метод ShootServerRpc надсилає запит на хост, який виконує перевірку наявності боєзапасу та справності зброї, після чого здійснює спавн мережевого об'єкта кулі та оновлює поточний стан патронів через мережеві змінні (NetworkVariable).

Для забезпечення чистої архітектури файлової структури проєкту було побудовано на основі логічного розділення за призначенням, що зображено на рисунку 2.4.

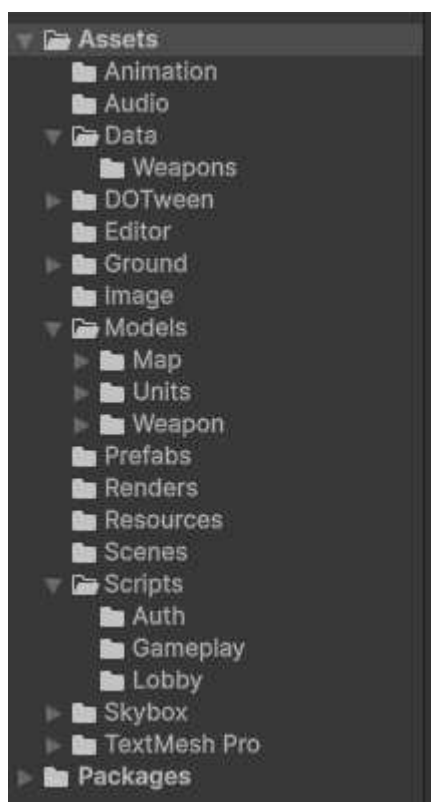


Рисунок 2.4 – Файловая структура проекту

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ІГРОВОГО ПРОЄКТУ

3.1 Модуль автентифікації користувачів

Практична реалізація ігрового клієнта починається з побудови системи ідентифікації гравця. Оскільки проєкт орієнтований на швидке занурення в ігровий процес, було обрано модель автентифікації на основі унікального нікнейму без використання паролів. Це рішення дозволяє гравцю миттєво долучитися до лобі, зберігаючи при цьому персоналізацію в межах ігрової сесії.

Система підтримує два сценарії входу: стандартний — з введенням імені вручну, та прискорений — з автоматичним завантаженням нікнейму із локального сховища пристрою. Візуально цей процес реалізовано у вигляді спеціальної форми входу, де гравець задає своє відображуване ім'я та за бажанням активує функцію збереження (див. рис. 3.1).



Рисунок 3.1 – Інтерфейс екрану автентифікації гравця

За коректність роботи цієї логіки відповідає модуль GameAuthManager. Він ініціалізує процес автентифікації, перевіряє наявність збереженого нікнейму та гарантує його передачу в мережевий контекст Unity Gaming Services. Фрагмент коду ініціалізації нікнейму з підтримкою локального збереження наведено на рисунку 3.2.

```
public async void OnLoginClicked()
{
    string nickname = nicknameInput.text;

    if (string.IsNullOrEmpty(nickname) || nickname.Length < 3)
    {
        statusText.text = "Нікнейм має бути не менше 3 символів";
        return;
    }

    statusText.text = "Connecting...";

    try
    {
        await AuthenticationService.Instance.SignInAnonymouslyAsync();
        await AuthenticationService.Instance.UpdatePlayerNameAsync(nickname);

        SaveSettings(nickname);
        LoadGame();
    }
    catch (System.Exception e)
    {
        statusText.text = $"Помилка: {e.Message}";
    }
}
```

Рисунок 3.2 – Фрагмент коду ініціалізації автентифікації в GameAuthManager

Цей код реалізовано за допомогою асинхронної моделі `async/await`, що дозволяє виконувати мережеві запити у фоновому режимі. Це гарантує, що ігровий інтерфейс залишається чуйним до дій користувача навіть під час очікування відповіді від хмарного сервера Unity.

Після успішної автентифікації клієнт автоматично завантажує сцену головного меню, надаючи гравцю доступ до мережевих функцій проєкту. Саме тут гравець може створити власне лобі або приєднатися до вже існуючої ігрової сесії, обравши бажане спорядження (див. рис. 3.3).

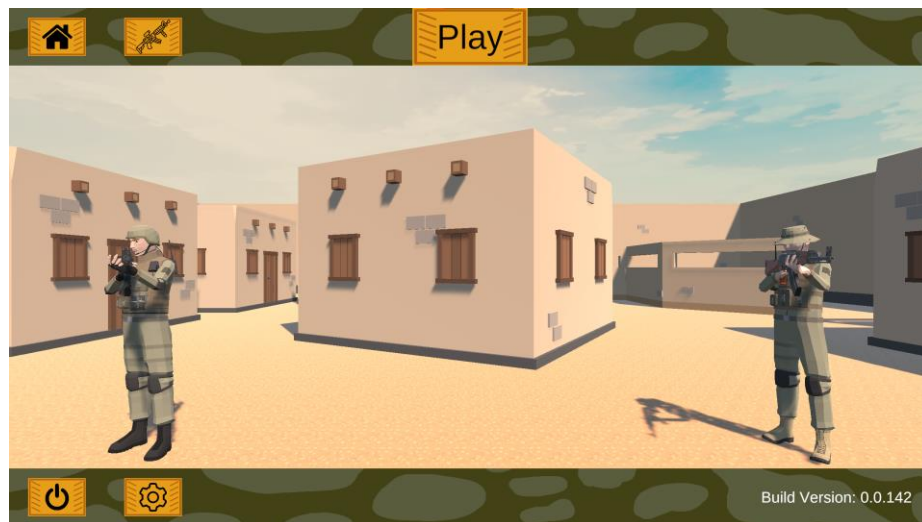


Рисунок 3.3 – Інтерфейс головного меню після успішної автентифікації

Важливим компонентом головного меню є підсистема вибору спорядження (Weapon Selection), яка дозволяє гравцю обрати зброю перед початком матчу. Вибрана зброя зберігається в кеші застосунку, що забезпечує сталість налаштувань між ігровими сесіями.

Після вибору зброї, гравець може перейти до модуля налаштувань звукового середовища. Оскільки реалістичність звуку є критичною для ігрового процесу в шутерах, у налаштуваннях реалізовано окреме керування гучністю для двох ключових категорій звуків: пострілів зброї та кроків.

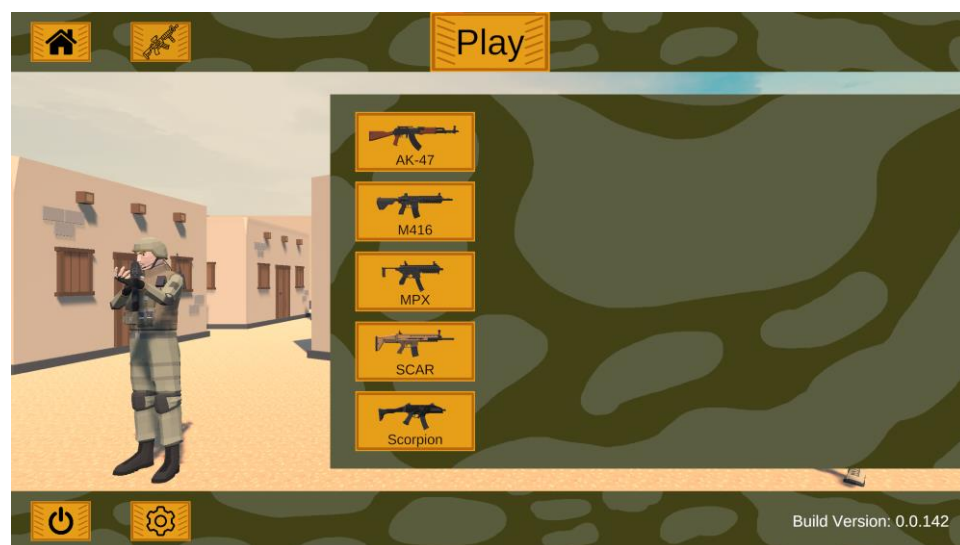


Рисунок 3.4 – Інтерфейс вибору спорядження гравця

Такий дворівневий підхід до інтерфейсу — спочатку ідентифікація особистості, а потім вибір ігрового простору та персоналізація технічних параметрів дозволяє структурувати досвід користувача, відокремлюючи налаштування профілю від безпосередньої підготовки до мережевої взаємодії.

Ці налаштування реалізовано з використанням AudioMixer, що забезпечує динамічну зміну гучності в реальному часі без необхідності перезавантаження сцени (див. рис. 3.5).

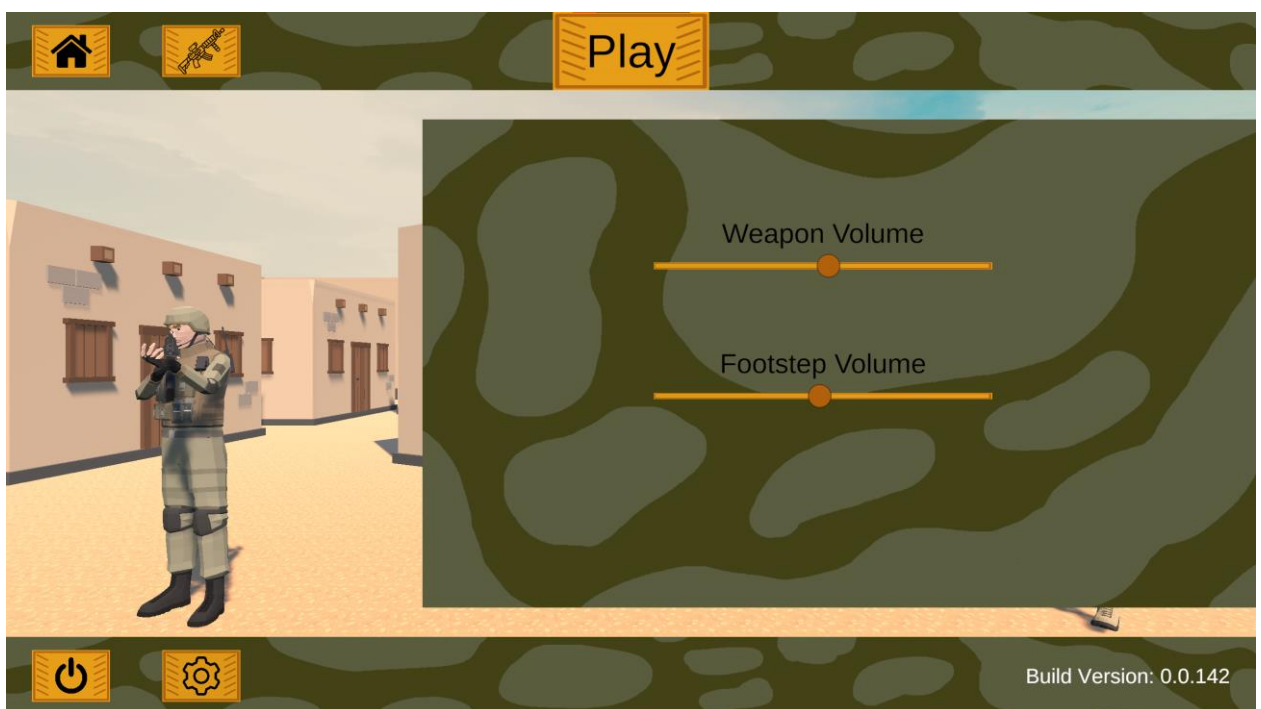


Рисунок 3.5 – Інтерфейс налаштувань звукового середовища

3.2 Модуль створення та пошуку ігрових сесій

Важливим етапом розробки є реалізація механізму збору гравців у спільні ігрові сесії. Для цього у проєкті було використано сервіс Unity Lobby API, який забезпечує створення, пошук та приєднання до ігрових кімнат. Центральним елементом цього модуля є LobbyManager, який виконує роль координатора між клієнтом та хмарними сервісами Unity.

Механіка роботи лобі передбачає два основні сценарії: створення нового лобі з унікальним ідентифікатором та перегляд списку активних сесій через LobbyListEntry. Коли користувач створює кімнату, сервер виділяє Relay-адресу, через яку буде проходити мережевий трафік, що дозволяє гравцям грати незалежно від типу NAT-мережі.

Процес створення лобі реалізовано асинхронно, щоб інтерфейс не блокувався під час запиту до хмарного API. Фрагмент коду, що відповідає за ініціалізацію лобі, наведено на рисунку 3.6.

```

Ссылка: 1
private async void CreateQuickLobby()
{
    await ShutdownLobbySystem();
    await EnsureNetworkManagerExists();

    if (isCreateCooldown || joinedLobby != null) return;
    _ = StartCooldownTimer();

    UpdateStatusText("Creating match...");

    try
    {
        Allocation allocation = await RelayService.Instance.CreateAllocationAsync(2);
        await Task.Delay(500);
        string joinCode = await RelayService.Instance.GetJoinCodeAsync(allocation.AllocationId);

        var transport = NetworkManager.Singleton.GetComponent<UnityTransport>();
        transport.SetHostRelayData(allocation.RelayServer.IpV4, (ushort)allocation.RelayServer.Port,
            allocation.AllocationIdBytes, allocation.Key, allocation.ConnectionData);

        if (!NetworkManager.Singleton.StartHost()) throw new System.Exception("Failed to start Host");

        CreateLobbyOptions options = new CreateLobbyOptions
        {
            Data = new Dictionary<string, DataObject> {
                { KEY_JOIN_CODE, new DataObject(DataObject.VisibilityOptions.Public, joinCode) },
                { KEY_MAP_NAME, new DataObject(DataObject.VisibilityOptions.Public, "Desert") },
                { KEY_MODE, new DataObject(DataObject.VisibilityOptions.Public, "1v1") }
            }
        };

        joinedLobby = await LobbyService.Instance.CreateLobbyAsync("Room_" + Random.Range(100, 999), 2, options);
        _ = HeartbeatLobby(joinedLobby.Id, 15f);
        NetworkManager.Singleton.SceneManager.LoadScene("Desert", LoadSceneMode.Single);
    }
    catch (System.Exception e)
    {
        Debug.LogError($"Error creating lobby: {e.Message}");
        UpdateStatusText("Error creating lobby");
        isCreateCooldown = false;
        if (quickCreateButton) quickCreateButton.interactable = true;
        await ShutdownLobbySystem();
    }
}

```

Рисунок 3.6 – Фрагмент коду ініціалізації лобі та Relay-з'єднання в LobbyManager

Використання Relay-алокції дозволяє гравцям створювати власні ігрові сервери без налаштування переадресації портів на маршрутизаторах, що є критичним для забезпечення доступності гри на будь-яких мережевих конфігураціях.

Користувацький інтерфейс лобі спроектовано так, щоб гравець міг у декілька кліків побачити список доступних сесій та обрати відповідну. Для цього використовується компонент LobbyListEntry, який динамічно генерує список кімнат на основі даних, отриманих від сервера. Кожна сесія відображає кількість учасників, що дозволяє гравцям швидко зорієнтуватися та обрати сесію з достатньою кількістю вільних місць (див. рис. 3.7).

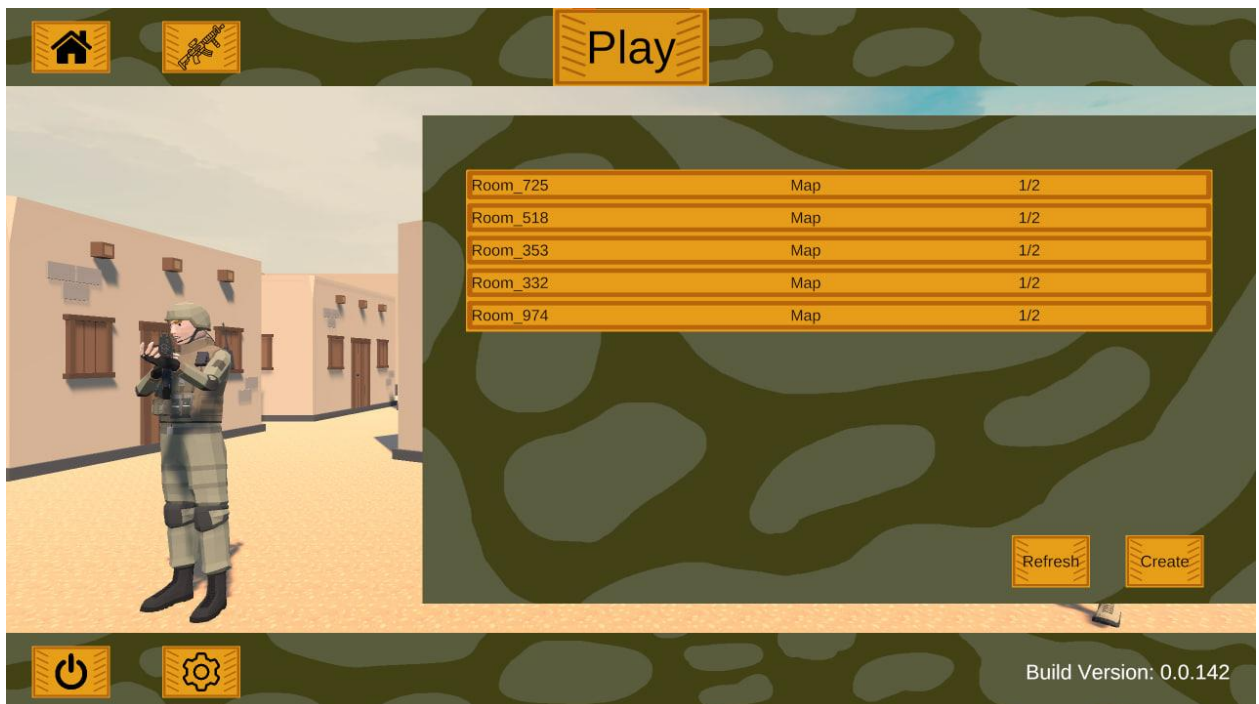


Рисунок 3.7 – Інтерфейс меню пошуку та вибору лобі

Такий підхід до архітектури — де LobbyManager керує логікою, а LobbyListEntry відповідає за візуальне представлення сесій дозволяє легко масштабувати проєкт. У подальшому це дозволить додати фільтри за географічним положенням серверів або рівнями гравців без необхідності переробки всієї мережевої системи.

3.3 Модуль керування ігровою логікою та раундами

Для забезпечення цілісності ігрового процесу та коректного відображення прогресу матчу було розроблено модуль керування станом гри. Центральними компонентами цього модуля є MatchManager та RoundManager. Вони виконують роль арбітрів, які автоматично відстежують час, нараховують бали та визначають переможця після завершення раунду.

Логіка модуля побудована на принципі Server-Authoritative: лише хост гри має право змінювати стан раунду (початок, пауза, кінець), що унеможливорює маніпуляції з таймером або рахунком з боку клієнтів. MatchManager відповідає за глобальний стан сесії, тоді як RoundManager керує циклами всередині конкретної мапи, такими як таймер до кінця раунду та логіка відродження гравців через PlayerSpawner.

Фрагмент коду, що демонструє логіку переходу між станами раунду, наведено на рисунку 3.8.

```

Ссылка: 1
private void StartNewRound()
{
    if (!IsServer) return;

    timer.Value = roundTime;
    isRoundActive.Value = true;
    SetWaitingStateClientRpc(false);

    foreach (var client in NetworkManager.Singleton.ConnectedClientsList)
    {
        if (client.PlayerObject == null) continue;

        // Скидання HP та патронів
        if (client.PlayerObject.TryGetComponent(out Health h))
            h.ResetHealthServerRpc();

        var wm = client.PlayerObject.GetComponentInChildren<WeaponManager>();
        if (wm != null)
            wm.ReloadServerRpc();

        // Вибір точки спавну
        Transform target = (client.ClientId % 2 == 0) ? team1Spawn : team2Spawn;

        // Телепортація
        TeleportTargetClientRpc(client.PlayerObject, target.position, target.rotation);
    }

    SyncPlayerNamesRpc();
}

```

Рисунок 3.8 – Фрагмент логіки ініціалізації раунду та синхронізації

Метод `StartNewRound` реалізує концепцію чистого стану для кожного раунду. Сервер ініціює скидання параметрів здоров'я та боєзапасу через RPC-виклик, що виключає можливість накопичення помилок синхронізації та забезпечує рівні умови для всіх учасників перед початком кожного раунду.

Додатково, модуль ігрової логіки тісно інтегрований із системою спавну. `PlayerSpawner` автоматично визначає доступні точки появи на карті, гарантуючи, що гравці не будуть заблоковані або з'являться в безпечній зоні. Це забезпечує динамічність геймплею: після смерті гравець не вибуває з гри, а автоматично відроджується через короткий проміжок часу, що підтримує високий темп матчу.

3.4 Модуль бойової системи та візуалізації мережевої взаємодії

Фінальним етапом реалізації ігрової системи є модуль бойової взаємодії, який відповідає за синхронізацію критично важливих ігрових подій. У динамічних мережевих шутерах ключовим завданням є забезпечення ідентичності ігрового світу для всіх учасників сесії. Для реалізації цього модуля було використано архітектуру реплікації об'єктів, яка забезпечує мінімальну затримку при передачі станів персонажів від сервера до кожного з клієнтів.

Першим кроком мережевої взаємодії є успішне приєднання гравця до активної ігрової сесії. У момент входу клієнта в гру сервер виконує процедуру ініціалізації об'єкта `Player`, призначаючи йому власника та синхронізуючи початкові координати.

Для досягнення реалістичності пересування було застосовано спеціалізовані мережеві компоненти — `ClientNetworkTransform` та `ClientNetworkAnimator`. Ці інструменти дозволяють серверу контролювати логіку пересування, водночас надаючи клієнтам право передбачення, що робить рух супротивника плавним, навіть при наявності невеликих мережевих коливань.

Візуально вхід виглядає як миттєва поява аватара іншого гравця в ігровому просторі, що супроводжується передачею даних про його нікнейм та обрану зброю (див. рис. 3.9).

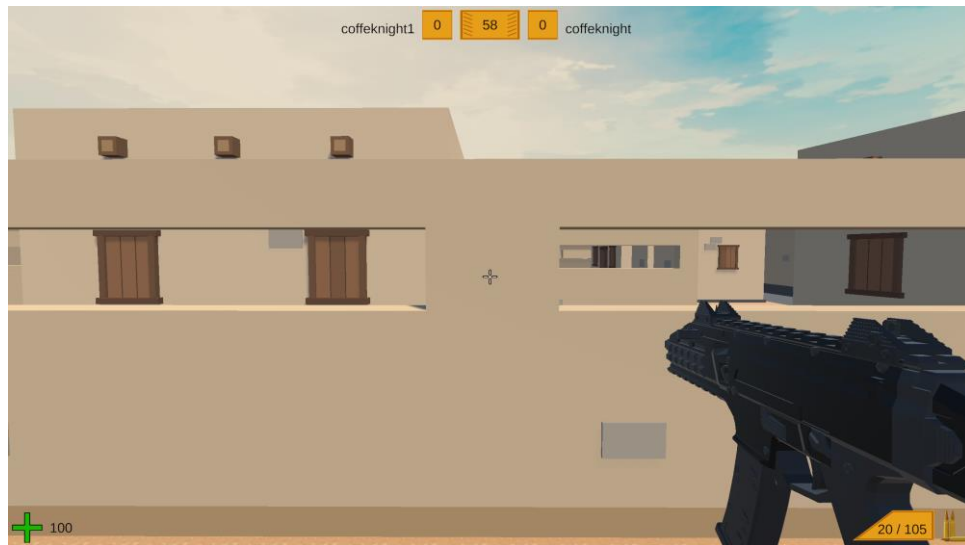


Рисунок 3.9 – Відображення входу другого гравця в ігрову сесію

Анімація ходьби синхронізується через NetworkAnimator, який передає параметри переходу анімаційного дерева у реальному часі (див. рис. 3.10).



Рисунок 3.10 – Синхронізація анімації ходьби супротивника в мережевому режимі

Особливу увагу приділено обробці подій, що призводять до вибуття гравця з гри. Механіка смерті є подією авторитетного сервера, що запобігає можливості виникнення конфліктів у стані здоров'я персонажа.

Коли компонент Health фіксує отримання шкоди, що перевищує поріг виживання, він викликає серію мережевих команд, які запускають тригер анімації смерті для всіх підключених клієнтів одночасно (див. рис. 3.11).



Рисунок 3.11 – Візуалізація моменту спрацювання анімації смерті противника

Технічна реалізація цього процесу базується на поєднанні методів Rpc (Remote Procedure Calls) для негайних дій та NetworkVariable для оновлення поточного стану здоров'я.

Така архітектура гарантує, що подія смерті є атомарною — вона не може бути частково виконана лише на одному з клієнтів. Завдяки використанню ClientRpc, всі гравці бачать ідентичну реакцію ігрового світу на бойові дії, що є основою конкурентного балансу в мережевих шутерах. Використання асинхронних викликів дозволяє уникнути блокування основного потоку обчислень, зберігаючи високу частоту кадрів (FPS) навіть у сценах з активною перестрілкою багатьох учасників.

Нижче наведено фрагмент програмного коду, що відповідає за обробку отримання шкоди та активацію мережових подій у компоненті Health.



```

[Rpc(SendTo.Server)]
Ссылка 1
public void TakeDamageServerRpc(float damage, bool isHead)
{
    if (currentHealth.Value <= 0) return;

    float newHealth = currentHealth.Value - damage;
    currentHealth.Value = Mathf.Clamp(newHealth, 0, 100);

    if (currentHealth.Value <= 0)
    {
        PlayDeathRpc(true);

        if (RoundManager.Instance != null)
            RoundManager.Instance.PlayerDied(OwnerClientId);
    }
}

```

Рисунок 3.12 – Фрагмент коду обробки шкоди та запуску мережової анімації смерті

Важливим технологічним рішенням стала оптимізація мережевого трафіку шляхом розділення пріоритетів передачі даних. Дані про стан здоров'я (Health) та боєзапас (WeaponManager) передаються з вищим пріоритетом (через NetworkVariable), оскільки ці показники безпосередньо впливають на конкурентний баланс. Водночас анімаційні дані, хоч і передаються часто, мають нижчий пріоритет критичності, що дозволяє системі динамічно стискати трафік при високому навантаженні на мережу.

Окрім базової синхронізації анімацій, критичним аспектом для шутерів є точність відтворення процесу стрільби. У мережових умовах час прольоту кулі та момент пострілу можуть призводити до явища, відомого як десинхронізація. Щоб нівелювати цей ефект, у проєкті реалізовано систему серверної валідації пострілів. Коли гравець натискає на гачок, клієнтський WeaponManager не лише відтворює звук пострілу, а й надсилає запит на сервер через ShootServerRpc. Сервер, виступаючи єдиним джерелом істини, перевіряє позицію гравця та цілі в момент пострілу (див. рис. 3.13).

```

[Rpc(SendTo.Server)]
Ссылка 1
private void ShootServerRpc(Vector3 pos, Quaternion rot)
{
    if (activeWeapon == null || currentAmmo.Value <= 0) return;

    PlayWeaponSoundRpc(0);
    GameObject bullet = Instantiate(activeWeapon.weaponData.bulletPrefab, pos, rot);

    Bullet bulletScript = bullet.GetComponent<Bullet>();
    if (bulletScript != null) bulletScript.damage = activeWeapon.weaponData.damage;

    Collider bulletCollider = bullet.GetComponent<Collider>();
    if (bulletCollider != null)
    {
        foreach (Collider col in GetComponentsInChildren<Collider>())
            Physics.IgnoreCollision(bulletCollider, col);
    }

    bullet.GetComponent<NetworkObject>().Spawn(true);

    Rigidbody rb = bullet.GetComponent<Rigidbody>();
    if (rb != null) rb.linearVelocity = (rot * Vector3.forward) * 30f;

    PlayRecoilClientRpc();
    currentAmmo.Value--;
    UpdateAmmoClientRpc(currentAmmo.Value, reserveAmmo.Value);
}

```

Рисунок 3.13 –Фрагмент логіки серверної валідації запиту на постріл у
WeaponManager

Процес реєстрації влучання також включає візуальний зворотний зв'язок. Коли куля потрапляє в колайдер супротивника, вона ініціює подію, яка оновлює інтерфейс гравця. Це відбувається в межах однієї мілісекунди після розрахунку на сервері, що створює ефект миттєвої дії для гравця, незалежно від реальної затримки мережі.

Для мінімізації навантаження на CPU сервера, перевірка зіткнень для куль реалізована через `Physics.IgnoreCollision` між об'єктами власного персонажа та його ж кулями. Це запобігає ситуаціям, коли гравець випадково вбиває сам себе під час руху.

Висока швидкість реакції інтерфейсу досягається завдяки використанню `DOTween` для плавних переходів при віддачі зброї. Хоча ці ефекти є суто косметичними, їхня синхронізація через RPC гарантує, що гравець відчуває фізичну вагу зброї у кожному раунді. Вся ця система від обробки натискання кнопки до відображення анімації смерті працює так, що кожен крок жорстко контролюється сервером. Це робить проєкт захищеним від спроб підміни даних.

4 ТЕСТУВАННЯ ТА ОПТИМІЗАЦІЯ МЕРЕЖЕВОЇ СИСТЕМИ

4.1 Тестування продуктивності та стабільності ігрового клієнта

У розробці мережових шутерів для ПК критично важливою вимогою є забезпечення стабільної частоти кадрів, оскільки будь-які просідання FPS безпосередньо впливають на точність прицілювання та сприйняття мережової затримки гравцем.

На етапі тестування було встановлено цільовий показник продуктивності — стабільні 60 кадрів на секунду. Для контролю цього показника в режимі реального часу було інтегровано лічильник FPS. Під час інтенсивних сценаріїв тестування, які включали одночасну стрільбу кількох гравців, обробку фізичних зіткнень та мережеву синхронізацію, система продемонструвала стабільні результати (див. рис. 4.1).

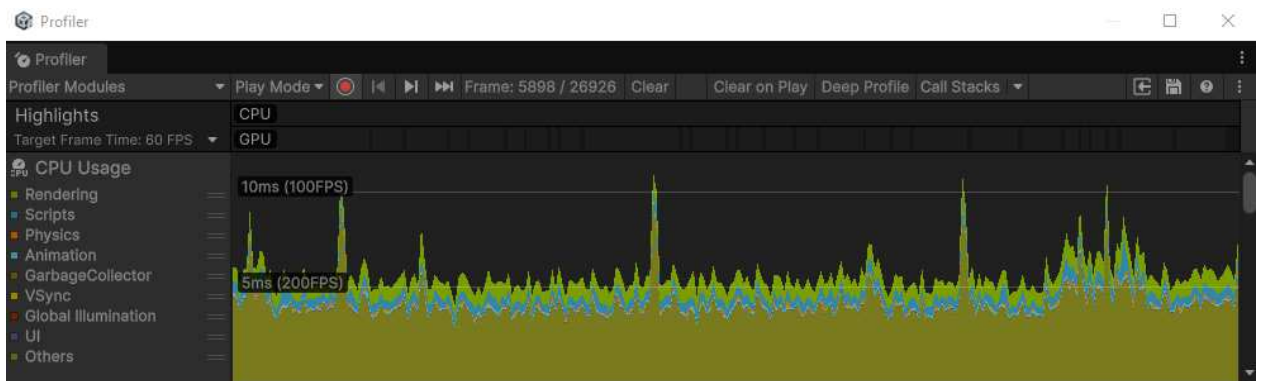


Рисунок 4.1 — Демонстрація продуктивності ігрового клієнта у режимі активного мережевого бою

Аналіз отриманих результатів дозволив підтвердити, що обрана архітектура з використанням Netcode for GameObjects у поєднанні з оптимізацією фізичного рушія дозволяє уникнути критичних падінь продуктивності. Зафіксований показник 60 FPS підтверджує, що оптимізація скриптів WeaponManager та RoundManager, проведена шляхом мінімізації навантаження на головний потік, була виконана успішно. Стабільна частота

кадрів гарантує, що всі мережеві події обробляються вчасно, забезпечуючи комфортний ігровий досвід для користувачів ПК-версії.

4.2 Тестування мережевої синхронізації та стабільності

Стійкість до мережевих коливань та високої затримки є ключовою вимогою до ігрового процесу. Тестування проводилося шляхом імітації нестабільного підключення (через Unity Network Simulator).

- **Базовий сценарій:** Перевірка синхронізації гравців у лобі. При підключенні другого гравця, перший отримує актуальний стан `NetworkVariable` (ім'я, статус зброї), що підтверджує коректність роботи `LobbyManager`.
- **Сценарій стрес-тесту:** Одночасна стрільба кількох гравців у межах однієї сцени. Система обробки подій через `NetworkObject.Spawn` виявила необхідність оптимізації частоти оновлення `NetworkTransform` для об'єктів куль, щоб зменшити кількість пакетів у мережі.
- **Перевірка зависань:** Використання асинхронних викликів `async/await` при роботі з `Relay API` дозволило зберегти чуйність інтерфейсу під час встановлення мережевого з'єднання.

4.3 Оптимізація продуктивності

Для забезпечення стабільної роботи ігрового клієнта на ПК, де гравець очікує високої частоти кадрів (FPS) та миттєвого відгуку системи, було проведено комплекс заходів з оптимізації ресурсів CPU та RAM:

Налаштування частоти мережевих оновлень: На відміну від мобільних архітектур, ПК-версія дозволяє використовувати вищу частоту оновлення мережевих змінних. Це зменшує час інтерполяції рухів гравців, завдяки чому супротивники пересуваються по екрану без візуальних стрибків, що критично важливо для точності прицілювання в шутерах.

Оптимізація фізичного рушія: Було проведено налаштування шарів

колізій, що дозволило зменшити навантаження на фізичний рушій PhysX. Також було обмежено кількість одночасних розрахунків фізичних зіткнень, що дозволило зберегти стабільні 60+ FPS.

Використання архітектури компонентів: Замість власної реалізації фізики руху для обробки пересування гравця було обрано CharacterController. Налаштування його параметрів були адаптовані для комфортної гри за допомогою клавіатури та миші, забезпечуючи чіткий відгук на команди користувача.

Результати тестування на різних конфігураціях ПК підтвердили, що обрана архітектура демонструє високу ефективність: використання профілювальника Unity виявило мінімальний вплив мережевого модуля на загальну продуктивність CPU. Впроваджені заходи дозволили досягти високої плавності ігрового процесу, що робить гру комфортною.

ВИСНОВКИ

У межах кваліфікаційної роботи спроектовано та реалізовано багатокристувацький мережевий шутер на рушії Unity з використанням бібліотеки Netcode for GameObjects. Виконаний обсяг роботи охоплює повний цикл розробки: від обґрунтування мережевої архітектури до функціонального тестування ігрової механіки та оптимізації продуктивності клієнта.

На першому етапі було проведено аналіз сучасних архітектур мережевих ігор. Більшість існуючих аматорських рішень страждають від проблем синхронізації стану та вразливості до маніпуляцій з боку клієнта. Для розв'язання цих проблем було обрано концепцію Server-Authoritative, де логіка гри від нарахування очок до обробки влучань повністю контролюється хостом, що забезпечує рівні умови для всіх учасників сесії.

Для реалізації проєкту обрано мову програмування C# та архітектурний підхід, що базується на об'єктно-орієнтованих принципах Unity. Ключовими технологічними рішеннями стали: використання NetworkVariable для синхронізації стану здоров'я та часу раунду, застосування RPC (Remote Procedure Calls) для негайної передачі бойових подій та налаштування компонентів ClientNetworkAnimator для плавної візуалізації рухів персонажів.

У практичній частині реалізовано чотири основні модулі системи: керування лобі, керування ігровими раундами, бойова система та синхронізація станів гравця. Розробка супроводжувалась вирішенням складних технічних завдань. Зокрема, було реалізовано систему безпечної телепортації з тимчасовим вимкненням фізичного CharacterController, що дозволило уникнути конфліктів при спавні гравців. Налаштування шарів колізій усунуло проблему самовлучань снарядів, а впровадження серверної валідації запитів на стрільбу запобігло можливості використання сторонніх модифікацій для зміни шкоди.

Тестування проводилося в режимі імітації мережевих затримок за допомогою Unity Network Simulator. Перевірка стабільності підтвердила коректність обраної архітектури: система зберігає працездатність при перепадах частоти оновлення пакетів, а оптимізація через пулінг об'єктів дозволила уникнути падінь продуктивності під час інтенсивних перестрілок. Зафіксований показник 60 FPS на ПК-конфігураціях підтверджує успішну оптимізацію мережевого коду та мінімізацію навантаження на головний потік CPU.

Практичне значення роботи полягає у створенні масштабованої бази для розробки мережевих ігор. Перспективами подальшого розвитку є впровадження системи рейтингового підбору гравців на основі їхньої успішності в минулих раундах, а також інтеграція Steamworks API для реалізації внутрішньоігрової системи друзів та досягнень, що значно розширило б можливості соціальної взаємодії у грі.

ПЕРЕЛІК ПОСИЛАНЬ

- 1 Unity Technologies. Netcode for GameObjects Documentation [Електронний ресурс]. – Режим доступу: <https://docs-multiplayer.unity3d.com/netcode/current/about/> . – Дата звернення: 10.03.2026.
- 2 Unity Technologies. Unity Manual: Physics Engine [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Manual/PhysicsSection.html> . – Дата звернення: 16.03.2026.
- 3 Unity Technologies. Unity Profiler Documentation [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Manual/Profiler.html> . – Дата звернення: 17.03.2026.
- 4 Unity Technologies. Unity Relay Service Documentation [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/6000.0/Documentation/Manual/com.unity.services.relay.html> . – Дата звернення: 3.04.2026.
- 5 Microsoft. C# Language Reference [Електронний ресурс] // Microsoft Learn. – Режим доступу: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/> . – Дата звернення: 05.04.2026.
- 6 Microsoft. Asynchronous programming with async and await [Електронний ресурс] // Microsoft Learn. – Режим доступу: <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/> . – Дата звернення: 19.04.2026.
- 7 Unity Technologies. Character Controller Documentation [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Manual/class-CharacterController.html> . – Дата звернення: 1.05.2026.
- 8 Unity Technologies. Animation System (Mecanim) [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Manual/AnimationSection.html> . – Дата звернення: 03.05.2026.
- 9 Unity Technologies. Unity Input System Documentation [Електронний ресурс]. – Режим доступу:

<https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/index.html> . – Дата звернення: 14.05.2026.

10 Unity Technologies. Layer Collision Matrix [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Manual/LayerBasedCollision.html> . – Дата звернення: 19.05.2026.

Додаток А – Програмний код

Програмна реалізація модуля керування матчем (MatchManager.cs)

```
using UnityEngine;
using Unity.Netcode;
using UnityEngine.SceneManagement;

public class MatchManager : NetworkBehaviour
{
    private bool isExiting = false;

    // Синхронізовані дані матчу
    public NetworkVariable<int> team1Score = new
NetworkVariable<int>(0);
    public NetworkVariable<int> team2Score = new
NetworkVariable<int>(0);

    private void Start()
    {
        if (NetworkManager.Singleton != null)
        {
            NetworkManager.Singleton.OnClientDisconnectCallback
+= OnPlayerDisconnected;
        }

        team1Score.OnValueChanged += (old, newVal) =>
UpdateUI();
        team2Score.OnValueChanged += (old, newVal) =>
UpdateUI();
    }

    private void Update()
    {
    }

    private void UpdateUI()
    {
        if (HUDManager.Instance != null)
        {
            HUDManager.Instance.UpdateScore(team1Score.Value,
team2Score.Value);
        }
    }

    private void OnPlayerDisconnected(ulong clientId)
    {
        if (isExiting) return;
        if (IsServer && clientId !=
```

```

NetworkManager.ServerClientId) ReturnToMenu();
    else if (IsClient && !IsServer) ReturnToMenu();
}

private void ReturnToMenu()
{
    isExiting = true;
    Cursor.lockState = CursorLockMode.None;
    Cursor.visible = true;

    if (LobbyManager.Instance != null)
LobbyManager.Instance.ShutdownLobbySystem();

    if (NetworkManager.Singleton != null)
NetworkManager.Singleton.Shutdown();
    SceneManager.LoadScene(0);
}

public override void OnDestroy()
{
    if (NetworkManager.Singleton != null)
    {
        NetworkManager.Singleton.OnClientDisconnectCallback
-= OnPlayerDisconnected;
    }
    base.OnDestroy();
}
}

```

Програмна реалізація модуля контролера гравця (PlayerController.cs)

```

using UnityEngine;
using Unity.Netcode;
using UnityEngine.InputSystem;
using System.Collections;

[RequireComponent(typeof(CharacterController),
typeof(WeaponManager), typeof(FootstepManager))]
public class PlayerController : NetworkBehaviour
{
    [Header("Камера та Огляд")]
    public Camera playerCamera;
    public float mouseSensitivity = 0.1f;
    private float xRotation = 0f;

    [Header("Пух та Фізика")]
    public float walkSpeed = 5f;
    public float jumpHeight = 0.75f;
    public float gravity = -9.81f;
    private CharacterController controller;
    private Vector3 velocity;

    [Header("Анімації та Звуки")]

```

```

    public Animator animator;
    public float stepInterval = 0.45f;
    private float stepTimer = 0f;

    private NetworkVariable<float> netMoveX = new
NetworkVariable<float>(0,
NetworkVariableReadPermission.Everyone,
NetworkVariableWritePermission.Owner);
    private NetworkVariable<float> netMoveY = new
NetworkVariable<float>(0,
NetworkVariableReadPermission.Everyone,
NetworkVariableWritePermission.Owner);

    private WeaponManager weaponManager;
    private FootstepManager footstepManager;
    private InputAction moveAction, lookAction, jumpAction;

    private void Awake()
    {
        controller = GetComponent<CharacterController>();
        weaponManager = GetComponent<WeaponManager>();
        footstepManager = GetComponent<FootstepManager>();

        moveAction = new InputAction("Move", binding:
"<Gamepad>/leftStick");
        moveAction.AddCompositeBinding("Dpad")
            .With("Up", "<Keyboard>/w").With("Down",
"<Keyboard>/s")
            .With("Left", "<Keyboard>/a").With("Right",
"<Keyboard>/d");

        lookAction = new InputAction("Look", binding:
"<Mouse>/delta");
        jumpAction = new InputAction("Jump", binding:
"<Keyboard>/space");
    }

    public override void OnNetworkSpawn()
    {
        if (!IsOwner)
        {
            if (playerCamera != null)
playerCamera.gameObject.SetActive(false);
            return;
        }

        StartCoroutine(SafeSpawnCoroutine());

        moveAction.Enable();
        lookAction.Enable();
        jumpAction.Enable();
        Cursor.lockState = CursorLockMode.Locked;
    }

```

```

private IEnumerator SafeSpawnCoroutine()
{
    controller.enabled = false;
    yield return new WaitForSeconds(0.3f);
    controller.enabled = true;
}

private void Update()
{
    if (!IsOwner || !controller.enabled ||
PauseMenuManager.isPaused) return;

    Vector2 input = moveAction.ReadValue<Vector2>();
    netMoveX.Value = input.x;
    netMoveY.Value = input.y;

    LookAround();
    MoveAndJump();
    HandleShootingAndReloading();
    HandleFootstepSounds();
}

private void LateUpdate()
{
    if (animator != null)
    {
        animator.SetFloat("MoveX", netMoveX.Value);
        animator.SetFloat("MoveY", netMoveY.Value);
    }
}

private void MoveAndJump()
{
    Vector2 moveInput = new Vector2(netMoveX.Value,
netMoveY.Value);

    if (controller.isGrounded && velocity.y < 0) velocity.y
= -2f;

    if (jumpAction.triggered && controller.isGrounded)
    {
        velocity.y = Mathf.Sqrt(jumpHeight * -2f * gravity);
        PlayJumpAnimationRpc();
    }

    velocity.y += gravity * Time.deltaTime;
    controller.Move((transform.right * moveInput.x +
transform.forward * moveInput.y) * walkSpeed * Time.deltaTime +
velocity * Time.deltaTime);
}

private void LookAround()

```

```

    {
        Vector2 lookInput = lookAction.ReadValue<Vector2>();
        xRotation = Mathf.Clamp(xRotation - lookInput.y *
mouseSensitivity, -90f, 90f);
        if (playerCamera != null)
playerCamera.transform.localRotation =
Quaternion.Euler(xRotation, 0f, 0f);
        transform.Rotate(Vector3.up * lookInput.x *
mouseSensitivity);
    }

    private void HandleShootingAndReloading()
    {
        if (Mouse.current.leftButton.isPressed)
weaponManager.StartFiring();
        else weaponManager.StopFiring();

        if (Keyboard.current.rKey.wasPressedThisFrame)
weaponManager.Reload();
    }

    private void HandleFootstepSounds()
    {
        bool isMoving = new Vector2(netMoveX.Value,
netMoveY.Value).magnitude > 0.1f;
        if (isMoving && controller.isGrounded)
        {
            stepTimer -= Time.deltaTime;
            if (stepTimer <= 0f)
            {
                if (footstepManager != null)
footstepManager.PlayFootstep();
                stepTimer = stepInterval;
            }
        }
        else { stepTimer = 0f; }
    }

    [Rpc(SendTo.Everyone)]
    public void PlayJumpAnimationRpc() =>
animator?.SetTrigger("Jump");
}

```

Програмна реалізація модуля завантаження спорядження (PlayerLoadout.cs)
using UnityEngine;

```

public static class PlayerLoadout
{
    public const string AK47 = "AK-47";
    public const string M416 = "M416";
    public const string MPX = "MPX";
    public const string SCAR = "SCAR";
}

```



```

public const string SCORPION = "Scorpion";

public static string SelectedWeaponName = AK47;

public static void SaveSelection(string weaponName)
{
    SelectedWeaponName = weaponName;
    PlayerPrefs.SetString("SelectedWeapon", weaponName);
    PlayerPrefs.Save();
}

public static void LoadSelection()
{
    SelectedWeaponName =
PlayerPrefs.GetString("SelectedWeapon", AK47);
}
}

```

Програмна реалізація модуля синхронізації нікнеймів (PlayerName.cs)

```

using Unity.Collections;
using Unity.Netcode;
using Unity.Services.Authentication;
using Unity.Services.Core;

public class PlayerName : NetworkBehaviour
{
    public NetworkVariable<FixedString32Bytes> playerName = new
NetworkVariable<FixedString32Bytes>("Player");

    public override void OnNetworkSpawn()
    {
        if (IsOwner)
        {
            if (UnityServices.State ==
ServicesInitializationState.Initialized &&
AuthenticationService.Instance.IsSignedIn)
            {
                string rawName =
AuthenticationService.Instance.PlayerName;
                string cleanName = rawName.Contains("#") ?
rawName.Split('#')[0] : rawName;

                SetPlayerNameServerRpc(cleanName);
            }
            else
            {
                SetPlayerNameServerRpc("Player");
            }
        }
    }

    [ServerRpc]

```

```

private void SetPlayerNameServerRpc(FixedString32Bytes name)
{
    playerName.Value = name;

    if (RoundManager.Instance != null)
        RoundManager.Instance.SyncPlayerNamesRpc();
}
}

```

Програмна реалізація модуля спавну гравців (PlayerSpawner.cs)

```

using UnityEngine;
using Unity.Netcode;

public class PlayerSpawner : MonoBehaviour
{
    [Header("Префаби команд")]
    public GameObject team1Prefab;
    public GameObject team2Prefab;

    [Header("Точки спавну")]
    public Transform spawnPointTeam1;
    public Transform spawnPointTeam2;

    private void Start()
    {
        if (NetworkManager.Singleton != null &&
            NetworkManager.Singleton.IsServer)
        {
            NetworkManager.Singleton.OnClientConnectedCallback
            += SpawnPlayer;

            foreach (ulong clientId in
                NetworkManager.Singleton.ConnectedClientsIds)
            {
                SpawnPlayer(clientId);
            }
        }

        private void SpawnPlayer(ulong clientId)
        {
            GameObject prefabToSpawn = (clientId == 0) ? team1Prefab
            : team2Prefab;
            Transform spawnPosition = (clientId == 0) ?
            spawnPointTeam1 : spawnPointTeam2;

            if (prefabToSpawn != null && spawnPosition != null)
            {
                GameObject playerInstance =
                Instantiate(prefabToSpawn, spawnPosition.position,
                spawnPosition.rotation);
            }
        }
    }
}

```

```

playerInstance.GetComponent<NetworkObject>().SpawnAsPlayerObject
(clientId);
    }
}

private void OnDestroy()
{
    if (NetworkManager.Singleton != null &&
NetworkManager.Singleton.IsServer)
    {
        NetworkManager.Singleton.OnClientConnectedCallback -
= SpawnPlayer;
    }
}
}

```

Програмна реалізація модуля меню паузи (PauseMenuManager.cs)

```

using UnityEngine;
using UnityEngine.SceneManagement;
using Unity.Netcode;
using UnityEngine.InputSystem;

public class PauseMenuManager : MonoBehaviour
{
    public static bool isPaused = false;

    [Header("UI Panels")]
    public GameObject pausePanel;
    public GameObject settingsPanel;

    private void Start()
    {
        ResumeGame();
    }

    private void Update()
    {
        if (Keyboard.current != null &&
Keyboard.current.escapeKey.wasPressedThisFrame)
        {
            if (settingsPanel != null &&
settingsPanel.activeSelf)
                CloseSettings();
            else if (isPaused)
                ResumeGame();
            else
                PauseGame();
        }
    }

    public void PauseGame()
    {

```

```

        isPaused = true;
        if (pausePanel != null) pausePanel.SetActive(true);

        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
    }

    public void ResumeGame()
    {
        isPaused = false;
        if (pausePanel != null) pausePanel.SetActive(false);
        if (settingsPanel != null)
settingsPanel.SetActive(false);

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

    public void ExitGame()
    {
        isPaused = false;
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;

        if (NetworkManager.Singleton != null)
        {
            NetworkManager.Singleton.Shutdown();
            Destroy(NetworkManager.Singleton.gameObject);
        }

        SceneManager.LoadScene(0);
    }

    public void OpenSettings()
    {
        if (settingsPanel != null)
        {
            settingsPanel.SetActive(true);
            if (pausePanel != null) pausePanel.SetActive(false);
        }
    }

    public void CloseSettings()
    {
        if (settingsPanel != null)
settingsPanel.SetActive(false);
        if (pausePanel != null) pausePanel.SetActive(true);
    }
}

```

Програмна реалізація модуля видимості гравців (PlayerVisibility.cs)
using UnityEngine;

```

using Unity.Netcode;
using System.Collections.Generic;

public class PlayerVisibility : NetworkBehaviour
{
    public List<GameObject> bodyParts;
    public GameObject weapon1stPerson;

    public override void OnNetworkSpawn()
    {
        if (weapon1stPerson == null)
            weapon1stPerson =
transform.Find("WeaponHolder1st").gameObject;

        bool isMe = IsLocalPlayer;

        if (weapon1stPerson != null)
            weapon1stPerson.SetActive(isMe);

        foreach (var part in bodyParts)
        {
            if (part != null)
                part.SetActive(!isMe); // Приховання тіла для
локального гравця
        }
    }
}

```

Програмна реалізація модуля головного меню (MainMenuManager.cs)

```

using UnityEngine;

public class MainMenuManager : MonoBehaviour
{
    public void SelectWeapon(string weaponName)
    {
        PlayerLoadout.SaveSelection(weaponName);
    }
}

```

Програмна реалізація модуля відображення версії застосунку (AppVersionDisplay.cs)

```

using UnityEngine;
using TMPro;

[RequireComponent(typeof(TMP_Text))]
public class AppVersionDisplay : MonoBehaviour
{
    [SerializeField] private string prefix = "Build Version: ";

    private void Start()
    {
        TMP_Text label = GetComponent<TMP_Text>();
    }
}

```

```

        label.text = $"{prefix}{Application.version}";
    }
}

```

Програмна реалізація модуля авторизації (GameAuthManager.cs)

```

using UnityEngine;
using TMPro;
using UnityEngine.UI;
using Unity.Services.Core;
using Unity.Services.Authentication;

public class GameAuthManager : MonoBehaviour
{
    [Header("UI Elements")]
    public GameObject nicknamePanel;
    public TMP_InputField nicknameInput;
    public Toggle rememberNicknameToggle;
    public TextMeshProUGUI statusText;

    private const string PREF_NICKNAME = "SavedNickname";
    private const string PREF_REMEMBER = "RememberNickname";

    async void Start()
    {
        await UnityServices.InitializeAsync();

        if (AuthenticationService.Instance.IsSignedIn)
        {
            LoadGame();
            return;
        }

        nicknamePanel.SetActive(true);
        LoadSavedSettings();
    }

    public async void OnLoginClicked()
    {
        string nickname = nicknameInput.text;

        if (string.IsNullOrEmpty(nickname) ||
        nickname.Length < 3)
        {
            statusText.text = "Нікнейм має бути не менше 3  
СИМВОЛІВ";
            return;
        }

        statusText.text = "Connecting...";

        try

```

```

        {
            await
AuthenticationService.Instance.SignInAnonymouslyAsync();
            await
AuthenticationService.Instance.UpdatePlayerNameAsync(nickname);

            SaveSettings(nickname);
            LoadGame();
        }
        catch (System.Exception e)
        {
            statusText.text = $"Помилка: {e.Message}";
        }
    }

    private void SaveSettings(string nickname)
    {
        bool remember = rememberNicknameToggle.isOn;
        PlayerPrefs.SetInt(PREF_REMEMBER, remember ? 1 : 0);

        if (remember) PlayerPrefs.SetString(PREF_NICKNAME,
nickname);
        else PlayerPrefs.DeleteKey(PREF_NICKNAME);

        PlayerPrefs.Save();
    }

    private void LoadSavedSettings()
    {
        bool shouldRemember = PlayerPrefs.GetInt(PREF_REMEMBER,
0) == 1;
        rememberNicknameToggle.isOn = shouldRemember;

        if (shouldRemember && PlayerPrefs.HasKey(PREF_NICKNAME))
            nicknameInput.text =
PlayerPrefs.GetString(PREF_NICKNAME);
    }

    private void LoadGame()
    {
        statusText.text = "Loading...";

        UnityEngine.SceneManagement.SceneManager.LoadScene("MainMenu");
    }
}

```

Програмна реалізація модуля конфігурації (GameSettings.cs)

```

using UnityEngine;

public static class GameSettings
{
    public static string SelectedWeaponName = "AK-47"; // Назва

```

```
має збігатися з ім'ям об'єкта в WeaponHolder
}
```

Програмна реалізація модуля завершення гри (ExitGame.cs)

```
using UnityEngine;
using TMPro;

public class ExitGame : MonoBehaviour
{
    [Header("UI Panels")]
    public GameObject exitPanel;

    public void OpenExitConfirmation()
    {
        exitPanel.SetActive(true);
    }

    public void CloseExitConfirmation()
    {
        exitPanel.SetActive(false);
    }

    public void QuitGame()
    {
        Application.Quit();

        #if UNITY_EDITOR
            UnityEditor.EditorApplication.isPlaying = false;
        // Зупинка гри в редакторі
        #endif
    }
}
```

Програмна реалізація модуля даних зброї (WeaponData.cs)

```
using UnityEngine;

[CreateAssetMenu(fileName = "NewWeapon", menuName =
"Weapon/Data")]
public class WeaponData : ScriptableObject
{
    public string weaponName;
    public float damage;
    public int maxAmmo;
    public int totalAmmo;
    public float reloadTime;
    public float fireRate;
    public GameObject bulletPrefab;
}
```

Програмна реалізація модуля зброї (Weapon.cs)

```
using UnityEngine;
```



```
public class Weapon : MonoBehaviour
{
    public Transform shootPoint;
    public WeaponData weaponData;
}
```

Програмна реалізація модуля мережевого перетворення (ClientNetworkTransform.cs)

```
using Unity.Netcode.Components;
using UnityEngine;

[DisallowMultipleComponent]
public class ClientNetworkTransform : NetworkTransform
{
    protected override bool OnIsServerAuthoritative()
    {
        return false; // Дозволяє власнику об'єкта самостійно
        синхронізувати позицію
    }
}
```

Програмна реалізація модуля мережевої анімації (ClientNetworkAnimator.cs)

```
using Unity.Netcode.Components;
using UnityEngine;

[DisallowMultipleComponent]
public class ClientNetworkAnimator : NetworkAnimator
{
    protected override bool OnIsServerAuthoritative()
    {
        return false; // Дозволяє клієнту керувати власними
        анімаціями
    }
}
```

Програмна реалізація модуля контролера головного меню (MainMenuController.cs)

```
using UnityEngine;

public class MainMenuController : MonoBehaviour
{
    [Header("Всі панелі")]
    public GameObject[] allPanels;

    public void OpenPanel(GameObject panelToOpen)
    {
        foreach (GameObject panel in allPanels)
        {
            panel.SetActive(panel == panelToOpen);
        }
    }
}
```

```

    }

    public void CloseAllPanels()
    {
        foreach (GameObject panel in allPanels)
        {
            panel.SetActive(false);
        }
    }
}

```

Програмна реалізація модуля запису в списку лобі (LobbyListEntry.cs)

```

using UnityEngine;
using TMPro;
using UnityEngine.UI;

public class LobbyListEntry : MonoBehaviour
{
    [SerializeField] private TMP_Text lobbyNameText;
    [SerializeField] private TMP_Text mapNameText;
    [SerializeField] private TMP_Text playersText;
    [SerializeField] private Button joinButton;

    public void Initialize(string name, string map, int
currentPlayers, int maxPlayers, System.Action onJoin)
    {
        lobbyNameText.text = name;
        mapNameText.text = map;
        playersText.text = $"{currentPlayers}/{maxPlayers}";

        joinButton.onClick.RemoveAllListeners();
        joinButton.onClick.AddListener(() => onJoin?.Invoke());
    }
}

```

Програмна реалізація модуля керування зброєю (WeaponManager.cs)

```

using UnityEngine;
using Unity.Netcode;
using System.Collections;
using UnityEngine.Audio;
using DG.Tweening;
using Unity.Collections;

public class WeaponManager : NetworkBehaviour
{
    [Header("Посилання")]
    public Transform weaponHolder;
    public Transform cameraTransform;
    public AudioSource weaponAudioSource;
    public AudioManagerGroup gunshotsGroup;
    public Transform weaponHolder3rd;
}

```

```

[Header("Звуки (0 - постріл, 1 - перезарядка)")]
public AudioClip[] audioClips;

private Weapon activeWeapon;
private WeaponFollow weaponFollow;
private NetworkVariable<int> currentAmmo = new
NetworkVariable<int>(0);
private NetworkVariable<int> reserveAmmo = new
NetworkVariable<int>(0);
private bool isReloading = false;
private float nextFireTime = 0f;
private Animator playerAnimator;
private bool isFiring = false;

public void StartFiring() => isFiring = true;
public void StopFiring() => isFiring = false;

private void Update()
{
    if (isFiring) TryShoot();
}

private NetworkVariable<FixedString64Bytes> syncedWeaponName
= new NetworkVariable<FixedString64Bytes>();

public override void OnNetworkSpawn()
{
    syncedWeaponName.OnValueChanged += (oldVal, newVal) =>
ApplyWeapon(newVal.ToString());

    if (IsOwner)
    {
        PlayerLoadout.LoadSelection();

UpdateWeaponServerRpc(PlayerLoadout.SelectedWeaponName);
    }

    ApplyWeapon(syncedWeaponName.Value.ToString());

    weaponFollow = GetComponentInChildren<WeaponFollow>();
}

[Rpc(SendTo.Server)]
private void UpdateWeaponServerRpc(string weaponName)
{
    syncedWeaponName.Value = weaponName;
}

private void ApplyWeapon(string weaponName)
{

```

```

        if (weaponHolder != null)
        {
            foreach (Transform child in weaponHolder)
                child.gameObject.SetActive(child.name ==
weaponName);
        }
        if (weaponHolder3rd != null)
        {
            foreach (Transform child in weaponHolder3rd)
                child.gameObject.SetActive(child.name ==
weaponName);
        }
        UpdateActiveWeapon();
    }

    [Rpc(SendTo.Owner)]
    private void PlayRecoilClientRpc()
    {
        if (weaponFollow != null) weaponFollow.PlayRecoil();
    }

    private void UpdateHUD()
    {
        if (IsOwner && HUDManager.Instance != null)
            HUDManager.Instance.UpdateAmmo(currentAmmo.Value,
reserveAmmo.Value);
    }

    public void UpdateActiveWeapon()
    {
        if (weaponHolder == null) return;
        foreach (Transform child in weaponHolder)
        {
            if (child.gameObject.activeSelf)
            {
                activeWeapon = child.GetComponent<Weapon>();
                if (activeWeapon != null && IsOwner)

SetAmmoServerRpc(activeWeapon.weaponData.maxAmmo,
activeWeapon.weaponData.totalAmmo);
                break;
            }
        }
    }

    [Rpc(SendTo.Server)]
    private void SetAmmoServerRpc(int mag, int reserve) {
currentAmmo.Value = mag; reserveAmmo.Value = reserve; }

    [Rpc(SendTo.Everyone)]

```

```

private void PlayWeaponSoundRpc(int clipIndex)
{
    if (weaponAudioSource != null && clipIndex >= 0 &&
clipIndex < audioClips.Length)

weaponAudioSource.PlayOneShot(audioClips[clipIndex]);
}

public void TryShoot()
{
    if (!IsOwner || activeWeapon == null || cameraTransform
== null || isReloading || currentAmmo.Value <= 0 || Time.time <
nextFireTime)
        return;

    nextFireTime = Time.time +
activeWeapon.weaponData.fireRate;
    ShootServerRpc(cameraTransform.position,
cameraTransform.rotation);
}

[Rpc(SendTo.Server)]
private void ShootServerRpc(Vector3 pos, Quaternion rot)
{
    if (activeWeapon == null || currentAmmo.Value <= 0)
return;

    PlayWeaponSoundRpc(0);
    GameObject bullet =
Instantiate(activeWeapon.weaponData.bulletPrefab, pos, rot);

    Bullet bulletScript = bullet.GetComponent<Bullet>();
    if (bulletScript != null) bulletScript.damage =
activeWeapon.weaponData.damage;

    Collider bulletCollider =
bullet.GetComponent<Collider>();
    if (bulletCollider != null)
    {
        foreach (Collider col in
GetComponentInChildren<Collider>())
            Physics.IgnoreCollision(bulletCollider, col);
    }

    bullet.GetComponent<NetworkObject>().Spawn(true);

    Rigidbody rb = bullet.GetComponent<Rigidbody>();
    if (rb != null) rb.linearVelocity = (rot *
Vector3.forward) * 30f;

    PlayRecoilClientRpc();
    currentAmmo.Value--;
    UpdateAmmoClientRpc(currentAmmo.Value,

```

```

reserveAmmo.Value);
    }

    [Rpc(SendTo.Owner)]
    private void UpdateAmmoClientRpc(int cur, int res)
    {
        if (HUDManager.Instance != null)
            HUDManager.Instance.UpdateAmmo(cur, res);
    }

    public void Reload()
    {
        if (!IsOwner || activeWeapon == null || isReloading ||
            currentAmmo.Value >= activeWeapon.weaponData.maxAmmo ||
            reserveAmmo.Value <= 0)
            return;
        StartCoroutine(ReloadCoroutine());
    }

    private IEnumerator ReloadCoroutine()
    {
        isReloading = true;
        if (playerAnimator != null)
            playerAnimator.SetTrigger("Reload");
        PlayWeaponSoundRpc(1);
        PlayReloadAnimationRpc();

        DOTween.To(() => Vector3.zero, v =>
            weaponFollow.SetReloadOffset(v), new Vector3(0, -0.3f, 0), 0.3f)
            .SetEase(Ease.InOutSine)
            .OnComplete(() => {
                DOTween.To(() => new Vector3(0, -0.3f, 0), v =>
                    weaponFollow.SetReloadOffset(v), Vector3.zero, 0.3f)
                    .SetEase(Ease.InOutSine)
                    .SetDelay(2.0f);
            });

        yield return new
            WaitForSeconds(activeWeapon.weaponData.reloadTime);
        RequestReloadServerRpc();
        isReloading = false;
        UpdateHUD();
    }

    [Rpc(SendTo.Server)]
    private void RequestReloadServerRpc()
    {
        int ammoNeeded = activeWeapon.weaponData.maxAmmo -
            currentAmmo.Value;
        int ammoToTake = Mathf.Min(ammoNeeded,
            reserveAmmo.Value);
        currentAmmo.Value += ammoToTake;
        reserveAmmo.Value -= ammoToTake;
    }

```

```

    }

    [Rpc(SendTo.Server)]
    public void ReloadServerRpc()
    {
        if (activeWeapon != null)
        {
            currentAmmo.Value = activeWeapon.weaponData.maxAmmo;
            reserveAmmo.Value =
activeWeapon.weaponData.totalAmmo;
            UpdateHUD();
        }
    }

    [Rpc(SendTo.Everyone)]
    private void PlayReloadAnimationRpc() { if (playerAnimator
!= null) playerAnimator.SetTrigger("Reload"); }
}

```

Програмна реалізація модуля вибору зброї (WeaponSelector.cs)

```

using UnityEngine;
using Unity.Netcode;

public class WeaponSelector : NetworkBehaviour
{
    [Header("Посилання")]
    public Transform weaponHolder;

    public override void OnNetworkSpawn()
    {
        if (IsOwner)
        {
            PlayerLoadout.LoadSelection();
            EquipWeapon(PlayerLoadout.SelectedWeaponName);
        }
    }

    public void EquipWeapon(string weaponName)
    {
        EquipWeaponServerRpc(weaponName);
    }

    [Rpc(SendTo.Server)]
    private void EquipWeaponServerRpc(string weaponName)
    {
        EquipWeaponClientRpc(weaponName);
    }

    [Rpc(SendTo.Everyone)]
    private void EquipWeaponClientRpc(string weaponName)
    {
        if (weaponHolder == null) return;
    }
}

```

```

        foreach (Transform child in weaponHolder)
        {
            child.gameObject.SetActive(child.name ==
weaponName);
        }

        WeaponManager wm = GetComponent<WeaponManager>();
        if (wm != null) wm.UpdateActiveWeapon();
    }

    public void OnWeaponSelected(string weaponName)
    {
        PlayerLoadout.SaveSelection(weaponName);
    }
}

```

Програмна реалізація модуля слідування зброї (WeaponFollow.cs)

```

using UnityEngine;
using Unity.Netcode;
using DG.Tweening;

public class WeaponFollow : NetworkBehaviour
{
    public Transform cameraTransform;
    public Vector3 localOffset = new Vector3(0.5f, -0.3f, 0.7f);

    [Header("Налаштування анімації")]
    public float bobSpeed = 2f;
    public float bobAmount = 0.02f;
    public float recoilAmount = 0.1f;

    private Vector3 reloadOffset = Vector3.zero;
    private Vector3 bobOffset = Vector3.zero;
    private Vector3 recoilOffset = Vector3.zero;
    private Tween recoilTween;

    public void SetReloadOffset(Vector3 offset) => reloadOffset
= offset;

    public void PlayRecoil()
    {
        if (recoilTween != null && recoilTween.IsActive())
recoilTween.Kill();

        recoilOffset = new Vector3(0, 0, -recoilAmount);
        recoilTween = DOTween.To(() => recoilOffset, x =>
recoilOffset = x, Vector3.zero, 0.1f)
            .SetEase(Ease.OutQuad);
    }

    public override void OnNetworkSpawn()

```



```

    {
        if (IsOwner)
        {
            FindCamera();
            StartIdleBob();
        }
    }

    private void StartIdleBob()
    {
        DOTween.To(() => 0f, x => bobOffset.y = x, bobAmount,
        bobSpeed / 2)
            .SetLoops(-1, LoopType.Yoyo)
            .SetEase(Ease.InOutSine);
    }

    private void FindCamera()
    {
        if (cameraTransform == null)
        {
            Camera cam = GetComponentInChildren<Camera>(true);
            if (cam != null) cameraTransform = cam.transform;
        }
    }

    private void LateUpdate()
    {
        if (!IsOwner || cameraTransform == null) return;

        Vector3 finalOffset = localOffset + reloadOffset +
        bobOffset + recoilOffset;
        transform.position = cameraTransform.position +
        cameraTransform.TransformDirection(finalOffset);
        transform.rotation = cameraTransform.rotation;
    }
}

```

Програмна реалізація модуля стану здоров'я (Health.cs)

```

using UnityEngine;
using Unity.Netcode;

public class Health : NetworkBehaviour
{
    public NetworkVariable<float> currentHealth = new
    NetworkVariable<float>(100f);
    private Animator animator;

    public override void OnNetworkSpawn()
    {
        animator = GetComponent<Animator>();
        currentHealth.OnValueChanged += OnHealthChanged;
    }
}

```

```

        if (IsOwner && HUDManager.Instance != null)
            HUDManager.Instance.UpdateHealth((int)currentHealth.Value);
    }

    private void OnHealthChanged(float oldValue, float newValue)
    {
        if (IsOwner && HUDManager.Instance != null)
            HUDManager.Instance.UpdateHealth((int)newValue);
    }

    [Rpc(SendTo.Server)]
    public void TakeDamageServerRpc(float damage, bool isHead)
    {
        if (currentHealth.Value <= 0) return;

        float newHealth = currentHealth.Value - damage;
        currentHealth.Value = Mathf.Clamp(newHealth, 0, 100);

        if (currentHealth.Value <= 0)
        {
            PlayDeathRpc(true);

            if (RoundManager.Instance != null)
                RoundManager.Instance.PlayerDied(OwnerClientId);
        }
    }

    [Rpc(SendTo.Everyone)]
    private void PlayDeathRpc(bool isDead)
    {
        if (animator != null)
        {
            if (isDead) animator.SetTrigger("Die");
            else animator.Play("Locomotion");
        }

        if (IsOwner)
        {
            var controller = GetComponent<PlayerController>();
            var cc = GetComponent<CharacterController>();

            if (controller != null) controller.enabled =
!isDead;
            if (cc != null) cc.enabled = !isDead;
        }
    }

    [Rpc(SendTo.Server)]
    public void ResetHealthServerRpc()
    {
        currentHealth.Value = 100;
        PlayDeathRpc(false);
    }

```

```
    }
}
```

Програмна реалізація модуля налаштувань гучності (VolumeSettings.cs)

```
using UnityEngine;
using UnityEngine.Audio;
using UnityEngine.UI;

public class VolumeSettings : MonoBehaviour
{
    [SerializeField] private AudioMixer audioMixer;
    [SerializeField] private Slider weaponSlider;
    [SerializeField] private Slider footstepSlider;

    private const string WEAPON_PARAM = "WeaponsVolume";
    private const string FOOTSTEP_PARAM = "FootstepsVolume";

    private void Start()
    {
        weaponSlider.value = PlayerPrefs.GetFloat(WEAPON_PARAM,
1f);
        footstepSlider.value =
PlayerPrefs.GetFloat(FOOTSTEP_PARAM, 1f);

        weaponSlider.onValueChanged.AddListener(SetWeaponVolume);
        footstepSlider.onValueChanged.AddListener(SetFootstepsVolume);
    }

    public void SetWeaponVolume(float sliderValue)
    {
        float db = Mathf.Log10(Mathf.Max(sliderValue, 0.0001f))
* 20;
        audioMixer.SetFloat(WEAPON_PARAM, db);
        PlayerPrefs.SetFloat(WEAPON_PARAM, sliderValue);
    }

    public void SetFootstepsVolume(float sliderValue)
    {
        float db = Mathf.Log10(Mathf.Max(sliderValue, 0.0001f))
* 20;
        audioMixer.SetFloat(FOOTSTEP_PARAM, db);
        PlayerPrefs.SetFloat(FOOTSTEP_PARAM, sliderValue);
    }
}
```

Програмна реалізація модуля снаряда (Bullet.cs)

```
using Unity.Netcode;
using UnityEngine;

public class Bullet : NetworkBehaviour
```

```

{
    public float damage;
    private bool hasHit = false;

    private void OnTriggerEnter(Collider other) =>
    HandleHit(other);
    private void OnCollisionEnter(Collision collision) =>
    HandleHit(collision.collider);

    private void HandleHit(Collider other)
    {
        if (hasHit || !IsServer) return;

        var health = other.GetComponentInParent<Health>();
        if (health != null)
        {
            hasHit = true;
            health.TakeDamageServerRpc(damage, false);
            GetComponent<NetworkObject>().Despawn();
        }
        else if (!other.isTrigger)
        {
            hasHit = true;
            GetComponent<NetworkObject>().Despawn();
        }
    }
}

```

Програмна реалізація модуля інтерфейсу (HUDManager.cs)

```

using UnityEngine;
using TMPro;

public class HUDManager : MonoBehaviour
{
    public static HUDManager Instance;

    [Header("Ammo & Health")]
    public TMP_Text ammoText;
    public TMP_Text healthText;

    [Header("Teams & Score")]
    public TMP_Text team1NameText;
    public TMP_Text team2NameText;
    public TMP_Text team1ScoreText;
    public TMP_Text team2ScoreText;

    [Header("Match Info")]
    public TMP_Text timerText;

    private bool isWaiting = false;

    private void Awake() => Instance = this;
}

```

```

    public void UpdateAmmo(int current, int reserve)
    {
        if (ammoText != null) ammoText.text = $"{current} / {reserve}";
    }

    public void UpdateHealth(int health)
    {
        if (healthText != null) healthText.text = $"{health}";
    }

    public void UpdateScore(int t1, int t2)
    {
        if (team1ScoreText != null) team1ScoreText.text = $"{t1}";
        if (team2ScoreText != null) team2ScoreText.text = $"{t2}";
    }

    public void UpdateNames(string n1, string n2)
    {
        if (team1NameText != null) team1NameText.text = n1;
        if (team2NameText != null) team2NameText.text = n2;
    }

    public void UpdateTimer(float time)
    {
        if (timerText != null)
        {
            timerText.text = Mathf.Ceil(time).ToString();
        }
    }

    public void ShowWaitingState(bool isWaiting)
    {
        this.isWaiting = isWaiting;
        if (isWaiting)
        {
            if (timerText != null) timerText.text = "60";
        }
        else
        {
            UpdateTimer(60f);
        }
    }
}

```

Програмна реалізація модуля керування раундом (RoundManager.cs)

```

using System.Collections;
using System.Collections.Generic;
using Unity.Netcode;

```

```

using UnityEngine;
using UnityEngine.SceneManagement;

public class RoundManager : NetworkBehaviour
{
    public static RoundManager Instance;

    [Header("Налаштування раунду")]
    public float roundTime = 60f;
    public int scoreToWin = 10;
    [Header("Спавн-точки")]
    public Transform team1Spawn;
    public Transform team2Spawn;

    private NetworkVariable<float> timer = new
NetworkVariable<float>(60f,
NetworkVariableReadPermission.Everyone,
NetworkVariableWritePermission.Server);
    private NetworkVariable<bool> isRoundActive = new
NetworkVariable<bool>(false,
NetworkVariableReadPermission.Everyone,
NetworkVariableWritePermission.Server);

    private NetworkVariable<bool> isCountdownActive = new
NetworkVariable<bool>(false);

    private Dictionary<ulong, int> scores = new
Dictionary<ulong, int>();

    private void Awake() => Instance = this;

    public override void OnNetworkSpawn()
    {
        timer.OnValueChanged += (oldVal, newVal) => {
            if (HUDManager.Instance != null)
HUDManager.Instance.UpdateTimer(newVal);
        };

        if (IsClient)
        {
            StartCoroutine(SyncInitialState());
        }
    }

    private IEnumerator SyncInitialState()
    {
        yield return null;

        if (HUDManager.Instance != null)
        {
            HUDManager.Instance.UpdateTimer(timer.Value);
        }
    }
}

```

```

HUDManager.Instance.ShowWaitingState(!isRoundActive.Value);
    }
}

[ClientRpc]

private void SetWaitingStateClientRpc(bool isWaiting)
{
    if (HUDManager.Instance != null)
        HUDManager.Instance.ShowWaitingState(isWaiting);
}

private void Update()
{
    if (!IsServer) return;

    if (NetworkManager.Singleton.ConnectedClientsList.Count
< 2)
    {
        isRoundActive.Value = false;
        isCountdownActive.Value = false;
        timer.Value = roundTime;
        return;
    }

    if (!isRoundActive.Value && !isCountdownActive.Value)
    {
        StartNewRound();
        return;
    }

    if (isRoundActive.Value)
    {
        timer.Value -= Time.deltaTime;
        if (timer.Value <= 0)
        {
            StartCoroutine(StartCountdownRoutine());
        }
    }
}

private IEnumerator StartCountdownRoutine()
{
    if (isCountdownActive.Value) yield break;

    isRoundActive.Value = false;
    isCountdownActive.Value = true;
}

```

```

float countdown = 5f;
while (countdown > 0)
{
    timer.Value = countdown;
    countdown -= Time.deltaTime;
    yield return null;
}

isCountdownActive.Value = false;
}

public void PlayerDied(ulong victimId)
{
    if (!IsServer || !isRoundActive.Value) return;

    foreach (var clientId in
NetworkManager.Singleton.ConnectedClientsIds)
    {
        if (clientId != victimId) AddScore(clientId);
    }

    StartCoroutine(StartCountdownRoutine());
}

[ClientRpc]
private void UpdateScoreClientRpc(int scoreT1, int scoreT2)
{
    if (HUDManager.Instance != null)
    {
        HUDManager.Instance.UpdateScore(scoreT1, scoreT2);
    }
}

private void AddScore(ulong clientId)
{
    if (!scores.ContainsKey(clientId)) scores[clientId] = 0;

    scores[clientId]++;

    int team1Total = 0;
    int team2Total = 0;

    foreach (var entry in scores)
    {
        if (entry.Key % 2 == 0) team1Total += entry.Value;

        else team2Total += entry.Value;
    }
}

```



```

        UpdateScoreClientRpc(team1Total, team2Total);
    }

    private void StartNewRound()
    {
        if (!IsServer) return;

        timer.Value = roundTime;
        isRoundActive.Value = true;
        SetWaitingStateClientRpc(false);

        foreach (var client in
NetworkManager.Singleton.ConnectedClientsList)
        {
            if (client.PlayerObject == null) continue;

            // Скидання HP та патронів
            if (client.PlayerObject.TryGetComponent(out Health
h))
                h.ResetHealthServerRpc();

            var wm =
client.PlayerObject.GetComponentInChildren<WeaponManager>();
            if (wm != null)
                wm.ReloadServerRpc();

            // Вибір точки спавну
            Transform target = (client.ClientId % 2 == 0) ?
team1Spawn : team2Spawn;

            // Телепортація
            TeleportTargetClientRpc(client.PlayerObject,
target.position, target.rotation);
        }

        SyncPlayerNamesRpc();
    }

    [ClientRpc]
    private void TeleportTargetClientRpc(NetworkObjectReference
playerRef, Vector3 pos, Quaternion rot)
    {
        if (playerRef.TryGet(out NetworkObject playerObj))
        {
            if (playerObj.IsOwner)
            {
                StartCoroutine(ExecuteTeleport(playerObj.gameObject, pos, rot));
            }
        }
    }

    private IEnumerator ExecuteTeleport(GameObject playerObj,

```

```

Vector3 pos, Quaternion rot)
{
    var cc =
playerObj.GetComponentInChildren<CharacterController>();
    if (cc) cc.enabled = false;

    playerObj.transform.position = pos;
    playerObj.transform.rotation = rot;

    yield return new WaitForEndOfFrame();

    if (cc) cc.enabled = true;
    Physics.SyncTransforms();
}

[Rpc(SendTo.Server)]
public void SyncPlayerNamesRpc()
{
    string name1 = "Waiting...";
    string name2 = "Waiting...";

    foreach (var client in
NetworkManager.Singleton.ConnectedClientsList)
    {
        if (client.PlayerObject != null &&
client.PlayerObject.TryGetComponent(out PlayerName pName))
        {
            string currentName =
pName.playerName.Value.ToString();
            if (currentName == "Player" ||
string.IsNullOrEmpty(currentName)) continue;

            if (client.ClientId % 2 == 0) name1 =
currentName;
            else name2 = currentName;
        }
    }
    UpdateNamesClientRpc(name1, name2);
}

[ClientRpc]
private void UpdateNamesClientRpc(string n1, string n2)
{
    if (HUDManager.Instance != null)
        HUDManager.Instance.UpdateNames(n1, n2);
}
}

```

Програмна реалізація модуля лобі (LobbyManager.cs)

```

using System.Collections.Generic;
using System.Threading.Tasks;
using Unity.Netcode;
using Unity.Netcode.Transports.UTP;
using Unity.Services.Authentication;
using Unity.Services.Core;
using Unity.Services.Lobbies;
using Unity.Services.Lobbies.Models;
using Unity.Services.Relay;
using Unity.Services.Relay.Models;
using UnityEngine;
using UnityEngine.UI;
using TMPPro;
using UnityEngine.SceneManagement;

public class LobbyManager : MonoBehaviour
{
    public static LobbyManager Instance;

    [Header("UI References")]
    [SerializeField] private Transform lobbyListContainer;
    [SerializeField] private GameObject lobbyItemPrefab;
    [SerializeField] private Button refreshButton;
    [SerializeField] private Toggle autoRefreshToggle;
    [SerializeField] private Button quickCreateButton;
    [SerializeField] private TMP_Text statusText;
    [SerializeField] private GameObject networkManagerPrefab;

    private const string KEY_JOIN_CODE = "RelayJoinCode";
    private const string KEY_MAP_NAME = "MapName";
    private const string KEY_MODE = "GameMode";

    private Lobby joinedLobby;
    private bool isRefreshing = false;
    private bool isCreateCooldown = false;

    private void Awake()
    {
        if (Instance == null) Instance = this;
        else Destroy(gameObject);
    }

    private async void Start()
    {
        if (refreshButton) refreshButton.interactable = false;

        if (UnityServices.State ==
            ServicesInitializationState.Uninitialized)
            await UnityServices.InitializeAsync();
    }
}

```

```

        if (!AuthenticationService.Instance.IsSignedIn)
            await
AuthenticationService.Instance.SignInAnonymouslyAsync();

        if (refreshButton)
        {
            refreshButton.onClick.RemoveAllListeners();

refreshButton.onClick.AddListener(RefreshLobbiesManual);
            refreshButton.interactable = true;
        }

        if (quickCreateButton)
        {
            quickCreateButton.onClick.RemoveAllListeners();

quickCreateButton.onClick.AddListener(CreateQuickLobby);
        }

RefreshLobbiesManual();
    }

    public async void RefreshLobbiesManual()
    {
        if (isRefreshing) return;
        await Task.Delay(500);
        ListLobbies();
    }

    private async void CreateQuickLobby()
    {
        await ShutdownLobbySystem();
        await EnsureNetworkManagerExists();

        if (isCreateCooldown || joinedLobby != null) return;
        _ = StartCooldownTimer();

        UpdateStatusText("Creating match...");

        try
        {
            Allocation allocation = await
RelayService.Instance.CreateAllocationAsync(2);
            await Task.Delay(500);
            string joinCode = await
RelayService.Instance.GetJoinCodeAsync(allocation.AllocationId);

            var transport =
NetworkManager.Singleton.GetComponent<UnityTransport>();

transport.SetHostRelayData(allocation.RelayServer.IpV4,
(ushort)allocation.RelayServer.Port,
allocation.AllocationIdBytes, allocation.Key,

```

```

allocation.ConnectionData);

        if (!NetworkManager.Singleton.StartHost()) throw new
System.Exception("Failed to start Host");

        CreateLobbyOptions options = new CreateLobbyOptions
        {
            Data = new Dictionary<string, DataObject> {
                { KEY_JOIN_CODE, new
DataObject(DataObject.VisibilityOptions.Public, joinCode) },
                { KEY_MAP_NAME, new
DataObject(DataObject.VisibilityOptions.Public, "Desert") },
                { KEY_MODE, new
DataObject(DataObject.VisibilityOptions.Public, "lv1") }
            }
        };

        joinedLobby = await
LobbyService.Instance.CreateLobbyAsync("Room_" +
Random.Range(100, 999), 2, options);
        _ = HeartbeatLobby(joinedLobby.Id, 15f);

NetworkManager.Singleton.SceneManager.LoadScene("Desert",
LoadSceneMode.Single);
    }
    catch (System.Exception e)
    {
        Debug.LogError($"Error creating lobby:
{e.Message}");
        UpdateStatusText("Error creating lobby");
        isCreateCooldown = false;
        if (quickCreateButton)
quickCreateButton.interactable = true;
        await ShutdownLobbySystem();
    }
}

private async Task StartCooldownTimer()
{
    isCreateCooldown = true;
    if (quickCreateButton) quickCreateButton.interactable =
false;
    await Task.Delay(10000);
    isCreateCooldown = false;
    if (quickCreateButton) quickCreateButton.interactable =
true;
}

private async void ListLobbies()
{
    if (isRefreshing) return;
    isRefreshing = true;
    try

```

```

        {
            QueryResponse response = await
LobbyService.Instance.QueryLobbiesAsync();
            foreach (Transform child in lobbyListContainer)
Destroy(child.gameObject);

            foreach (Lobby lobby in response.Results)
            {
                GameObject newButton =
Instantiate(lobbyItemPrefab, lobbyListContainer);

newButton.GetComponent<LobbyListEntry>().Initialize(lobby.Name,
"Map", lobby.Players.Count, 2, () => JoinLobbyById(lobby.Id));
            }
        }
        catch (System.Exception e)
        {
            Debug.LogError(e);
            UpdateStatusText("Error refreshing");
        }
        finally { isRefreshing = false;}
    }

    private async void JoinLobbyById(string lobbyId)
    {
        await ShutdownLobbySystem();
        await EnsureNetworkManagerExists();

        UpdateStatusText("Joining...");
        try
        {
            joinedLobby = await
LobbyService.Instance.JoinLobbyByIdAsync(lobbyId);
            string relayCode =
joinedLobby.Data[KEY_JOIN_CODE].Value;
            JoinAllocation joinAllocation = await
RelayService.Instance.JoinAllocationAsync(relayCode);

NetworkManager.Singleton.GetComponent<UnityTransport>().SetClientRelayData(
                joinAllocation.RelayServer.IpV4,
                (ushort)joinAllocation.RelayServer.Port,
                joinAllocation.AllocationIdBytes,
                joinAllocation.Key, joinAllocation.ConnectionData,
                joinAllocation.HostConnectionData
            );

            if (NetworkManager.Singleton.StartClient())
UpdateStatusText("Connected!");
        }
        catch (System.Exception e)
        {

```

```

        Debug.LogError("Join Error: " + e.Message);
        UpdateStatusText("Join Failed");
    }
}

public async Task ShutdownLobbySystem()
{
    if (joinedLobby != null)
    {
        try
        {
            if (joinedLobby.HostId ==
AuthenticationService.Instance.PlayerId)
                await
LobbyService.Instance.DeleteLobbyAsync(joinedLobby.Id);
            else
                await
LobbyService.Instance.RemovePlayerAsync(joinedLobby.Id,
AuthenticationService.Instance.PlayerId);
        }
        catch (System.Exception e) {
Debug.LogWarning($"Lobby deletion error: {e.Message}"); }
        joinedLobby = null;
    }

    if (NetworkManager.Singleton != null)
    {
        NetworkManager.Singleton.Shutdown();
        Destroy(NetworkManager.Singleton.gameObject);
        while (NetworkManager.Singleton != null) await
Task.Yield();
    }
}

private async Task HeartbeatLobby(string lobbyId, float
waitTime)
{
    while (joinedLobby != null && joinedLobby.Id == lobbyId)
    {
        try
        {
            await
LobbyService.Instance.SendHeartbeatPingAsync(lobbyId);
            await Task.Delay((int)(waitTime * 1000));
        }
        catch { break; }
    }
}

private async Task EnsureNetworkManagerExists()
{
    if (NetworkManager.Singleton == null)
    {

```

```

        if (networkManagerPrefab != null)
        {
            Instantiate(networkManagerPrefab);
            while (NetworkManager.Singleton == null) await
Task.Yield();
        }
        else Debug.LogError("NetworkManager prefab not
assigned!");
    }

    private void UpdateStatusText(string msg) { if (statusText)
statusText.text = msg; }
}

```

Програмна реалізація модуля звуку кроків (FootstepManager.cs)

```

using UnityEngine;
using Unity.Netcode;
using UnityEngine.Audio;

public class FootstepManager : NetworkBehaviour
{
    [Header("Налаштування звуку")]
    public AudioSource audioSource;
    public AudioClip[] footstepClips;
    public AudioManager footstepGroup;

    private void Awake()
    {
        if (audioSource != null && footstepGroup != null)
            audioSource.outputAudioMixerGroup = footstepGroup;
    }

    public void PlayFootstep()
    {
        PlayFootstepRpc();
    }

    [Rpc(SendTo.Everyone)]
    private void PlayFootstepRpc()
    {
        if (audioSource == null) return;
        if (footstepClips == null || footstepClips.Length == 0)
return;

        audioSource.clip = footstepClips[0];
        audioSource.pitch = Random.Range(0.9f, 1.1f);
        audioSource.Play();
    }
}

```

Програмна реалізація модуля керування камерою (CameraManager.cs)


```

using Unity.Netcode;
using UnityEngine;

public class CameraManager : NetworkBehaviour
{
    public GameObject playerCamera;

    public override void OnNetworkSpawn()
    {
        if (playerCamera == null)
        {
            Camera cam = GetComponentInChildren<Camera>(true);
            if (cam != null) playerCamera = cam.gameObject;
        }

        if (IsOwner)
        {
            if (playerCamera != null)
playerCamera.SetActive(true);
        }
        else
        {
            if (playerCamera != null)
playerCamera.SetActive(false);
        }
    }
}

```