ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «СТВОРЕННЯ 3D-ГРИ З РЕАЛІЗАЦІЄЮ ФІЗИЧНОЇ СИМУЛЯЦІЇ ОБ'ЄКТІВ»

на здобуття освітнього ступеня магістра
зі спеціальності F6 Інформаційні системи і технології
освітньо-професійної програми Інформаційні системи та технології

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ *Микита БИКОВ*

Виконав:
здобувач вищої освіти Микита БИКОВ
група ІСДМ-61

Керівник: Валентина ДАНИЛЬЧЕНКО
науковий ступінь, вчене звання к.т.н., доцент

Рецензент:
науковий ступінь, вчене звання _____
Ім’я, ПРІЗВИЩЕ

Київ – 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Інформаційних систем та технологій

Ступінь вищої освіти Магістр

Спеціальність F6 Інформаційні системи і технології

Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІСТ

_____Каміла СТОРЧАК
«____» _____20___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ СТУДЕНТУ
Бикову Микиті Роландовичу
(прізвище, ім'я, по батькові здобувача)

1.Тема кваліфікаційної роботи: «Створення 3D-гри з реалізацією фізичної симуляції об'єктів»

керівник кваліфікаційної роботи Валентина ДАНИЛЬЧЕНКО, к.т.н., доцент
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від «30» жовтня 2025 року № 467.

2. Строк подання кваліфікаційної роботи: 26 грудня 2025 року.

3. Вихідні дані до кваліфікаційної роботи: Середовище розробки Unity 3D та мова програмування C#;
мережевий фреймворк Photon PUN 2 для реалізації клієнт-серверної архітектури;
вбудований фізичний рушій NVIDIA PhysX для симуляції взаємодії об'єктів;
технічна документація Unity Technologies та Photon Engine;
науково-технічна література з питань архітектури ігрових застосунків та комп'ютерної графіки.

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

1. Аналітичний огляд сучасних інструментальних засобів розробки 3D-ігор та порівняльний аналіз ігрових рушіїв (Unity, Unreal Engine, Godot).
2. Проєктування архітектури програмної системи: розробка модулів інтерфейсу, мережевого з'єднання та ігрової логіки.
3. Програмна реалізація фізичної моделі поведінки об'єктів та механік взаємодії гравця з оточенням.
4. Розробка мережевої інфраструктури та механізмів синхронізації станів клієнтів за допомогою Photon PUN.
5. Тестування та верифікація стабільності мережевого підключення і коректності фізичних симуляцій.

5. Перелік ілюстративного матеріалу: *презентація*

1. Актуальність.
2. Структурна схема.
3. Блок-схеми алгоритмів мережевої взаємодії та створення кімнат.
4. Діаграми класів програмних модулів.
5. Скріншоти інтерфейсу користувача та демонстрація ігрового процесу.
6. Дата видачі завдання: 30 жовтня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	30.10 – 05.11.25	Виконав
2	Обґрунтування вибору ігрового рушія та проєктування архітектури	06.11 – 14.11.25	Виконав
3	Реалізація фізики та керування гравцем	15.11 – 20.11.25	Виконав
4	Розробка мережевої підсистеми, синхронізація станів та подій (RPC) за допомогою Photon PUN	21.11 – 28.11.25	Виконав
5	Створення графічного інтерфейсу користувача (UI) та інтеграція ігрових сцен	29.11 – 05.12.25	Виконав
6	Комплексне тестування програмного продукту, верифікація фізики та виправлення помилок	06.12 – 12.12.25	Виконав
7	Оформлення роботи: вступ, висновки, реферат	13.12 – 17.12.25	Виконав
8	Розробка демонстраційних матеріалів	18.12 – 23.12.25	Виконав

Здобувач вищої освіти _____ Микита БИКОВ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник роботи
кваліфікаційної роботи _____ Валентина ДАНИЛЬЧЕНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 71 стор., 8 рис., 30 джерел.

Мета роботи - проєктування та програмна реалізація прототипу багатокористувацької 3D-гри, що забезпечує стабільну взаємодію гравців та фізичних об’єктів у реальному часі.

Об'єкт дослідження – процес розробки тривимірного ігрового застосунку з використанням сучасних інструментальних засобів.

Предмет дослідження – методи реалізації фізичної симуляції твердих тіл та механізми синхронізації ігрових станів у мережевому середовищі.

Короткий зміст роботи: У роботі проведено аналітичний огляд еволюції 3D-ігор та обґрунтовано вибір рушія Unity як основного інструменту розробки. Розроблено модульну архітектуру програмної системи, що включає підсистеми інтерфейсу користувача, мережевого з’єднання та ігрової логіки. Реалізовано фізичну модель поведінки об’єктів (переміщення, колізії) та механіку взаємодії гравця з оточенням засобами мови C# та бібліотеки Photon PUN 2. Створено мережеву інфраструктуру, яка забезпечує синхронізацію створення кімнат, підключення гравців та обробку подій (постріли, нанесення шкоди) через механізм RPC. Проведено тестування системи, яке підтвердило стабільність роботи фізичної та мережевої складових.

КЛЮЧОВІ СЛОВА: UNITY, C#, PHOTON PUN, 3D-ГРА, ФІЗИЧНА СИМУЛЯЦІЯ, БАГАТОКОРИСТУВАЦЬКИЙ РЕЖИМ, КЛІЄНТ-СЕРВЕР, RAYCAST, PHYSX, GAME DEVELOPMENT.

ABSTRACT

Text part of the master`s qualification work: 71 pages, 8 pictures, 30 sources.

The purpose of the work is the design and software implementation of a prototype of a multiplayer 3D game that ensures stable interaction between players and physical objects in real-time.

Object of research is the process of developing a three-dimensional game application using modern software tools.

Subject of research is the methods of implementing rigid body physical simulation and mechanisms for synchronizing game states in a network environment.

Summary of the work: The paper conducts an analytical review of the evolution of 3D games and justifies the choice of the Unity engine as the main development tool. A modular software system architecture was developed, including subsystems for the user interface, network connection, and game logic. A physical model of object behavior (movement, collisions) and mechanics of player interaction with the environment were implemented using the C# language and the Photon PUN 2 library. A network infrastructure was created that ensures synchronization of room creation, player connection, and event processing (shooting, damage dealing) via the RPC mechanism. System testing was conducted, confirming the stability of the physical and network components.

KEYWORDS: UNITY, C#, PHOTON PUN, 3D GAME, PHYSICAL SIMULATION, MULTIPLAYER, CLIENT-SERVER, RAYCAST, PHYSX, GAME DEVELOPMENT.

ЗМІСТ

ВСТУП

Стрімка еволюція цифрових технологій в останні десятиліття докорінно змінила ландшафт індустрії розваг та інтерактивних медіа. Розвиток тривимірних відеоігор пройшов шлях від примітивних векторних симуляцій 1970-х років, таких як *Maze War* чи *Battlezone*, до фотореалістичних віртуальних світів, що використовують технології трасування променів та штучного інтелекту. Сучасні вимоги до програмних продуктів цього класу виходять далеко за межі візуальної привабливості: ключовими факторами успіху стають коректна фізична модель, стабільна мережева взаємодія та оптимізація обчислювальних ресурсів.

Актуальність теми дослідження зумовлена необхідністю пошуку ефективних архітектурних рішень для створення багатокористувацьких 3D-середовищ, де критично важливим є баланс між реалістичністю фізичних процесів та швидкістю передачі даних мережею. Впровадження фізично коректної взаємодії об’єктів (Physically Based Rendering та Simulation) дозволяє досягти високого рівня занурення користувача , проте створює значне навантаження на апаратну частину та канали зв'язку, особливо в умовах реального часу.

Об’єктом дослідження є процес розробки тривимірного ігрового застосунку з використанням сучасних інструментальних засобів. Предметом дослідження виступають методи реалізації фізичної симуляції твердих тіл та механізми синхронізації ігрових станів у мережевому середовищі.

Метою роботи є проєктування та програмна реалізація прототипу багатокористувацької 3D-гри, що забезпечує стабільну взаємодію гравців та фізичних об’єктів. Для досягнення мети було вирішено наступні завдання:

1. Провести аналітичний огляд історії розвитку 3D-ігор та еволюції графічних рушіїв для виявлення сучасних стандартів розробки.
2. Здійснити порівняльний аналіз провідних ігрових рушіїв (Unity, Unreal Engine, Godot) та обґрунтувати вибір інструментарію.

3. Розробити архітектуру програмної системи, що включає модулі керування інтерфейсом, мережевим з’єднанням та ігровою логікою.
4. Реалізувати фізичну модель поведінки об’єктів та механіку взаємодії гравця з оточенням засобами мови C# та бібліотеки Photon PUN 2.
5. Провести тестування розробленої системи для верифікації стабільності мережевого підключення та коректності фізичних розрахунків.

Практичне значення отриманих результатів полягає у створенні робочої моделі клієнт-серверної програми, архітектура якої може бути масштабована для створення складніших симуляційних або розважальних проєктів.

1 АНАЛІТИЧНИЙ ОГЛЯД ТА ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ 3D-ІГОР

1.1 Історія розвитку 3D-ігор

Розвиток тривимірних відеоігор є складним і багатогранним процесом, що формувався протягом кількох десятиліть і поступово змінював як технологічні підходи, так і художні стандарти інтерактивних цифрових медіа. Перші спроби створення віртуального простору, що моделює тривимірність, з'явилися ще у 1970-х роках, коли обчислювальні системи мали украй обмежені потужності, а будь-яка графічна візуалізація вимагала суттєвих інженерних рішень. Попри примітивність ранніх алгоритмів, саме в ті роки закладалися перші принципи просторового рендерингу, управління камерою та обробки зіткнень. Одним із найважливіших експериментів періоду стала гра Maze War (1973), яка фактично запропонувала першу форму відображення 3D-лабіринту з перспективою від першої особи. Вона не використовувала повноцінні тривимірні моделі, але застосовувала принципи симуляції об'єктів у псевдопросторі, що дозволило гравцеві переміщатися коридорами, які виглядали наче об'ємні.

Подальший розвиток тривимірності значно прискорився у 1980-х роках, коли завдяки доступності недорогих домашніх комп'ютерів з'явилися перші ігри, що відтворювали 3D-сцени за допомогою векторної графіки. Існували титули, що працювали виключно з геометричними лініями та каркасними моделями, які не мали текстур, проте створювали ілюзію глибини та руху в просторі. Прикладом стала Battlezone (1980), аркадна гра, яка популяризувала векторну графіку та продемонструвала потенціал тривимірної взаємодії. У цей час розробники почали експериментувати з рендерингом полігональних фігур, хоча апаратні обмеження накладали значні рамки на кількість полігонів і складність сцен.

3D-графіка почала еволюціонувати стрімко лише в 1990-х роках, коли з'явилися перші рушії, здатні об'єднати рендеринг, фізику та логіку гри в єдину

систему. Одним із найвпливовіших стал рушій id Tech, розроблений студією id Software. Ігри Wolfenstein 3D (1992) та особливо Doom (1993) були революційними, хоча з технічної точки зору використовували не повністю тривимірні середовища, а методи рейтрейсингу по двовимірній карті з використанням вертикальних “стовпців” (ray casting). Але саме вони заклали основний фундамент шутерів від першої особи, створили концепцію рівнів, освітлення та фізичних взаємодій у просторі, які хоч і спрощені, але стали прототипом майбутніх 3D-технологій.

Переломним моментом став вихід Quake (1996), першої широко розповсюдженої гри, що використовувала повністю тривимірний рушій із полігональними моделями та справжнім 3D-простором. Quake не лише підвищив вимоги до апаратного забезпечення, а й сприяв появі ринку графічних прискорювачів, зокрема завдяки популярності API OpenGL. З цього моменту розвиток 3D-ігор став прямою функцією можливостей відеокарт та ефективності алгоритмів рендерингу. Поступово в індустрії стали масово впроваджуватися такі технології, як згладжування, текстурні фільтри, динамічне освітлення, тінювання та моделювання фізичних процесів.

На межі 1990–2000-х років тривимірна графіка стає загальноприйнятим стандартом, а робота над нею поступово переходить від ентузіастів до великих компаній, які створюють масштабні рушії. Одним із ключових став Unreal Engine, який у 1998 році продемонстрував можливість створення складних рівнів із використанням об’ємного освітлення, адитивного та субтрактивного моделювання геометрії та складних матеріалів. У той самий час рушій Source від Valve продемонстрував інтеграцію фізики Havok, що стало проривом у відображенні руху, об’єктів та взаємодії в реальному часі. Окрім цього, у цей період відбувається розвиток тривимірних анімацій, зокрема технології скелетної анімації, що дозволяє створювати більш реалістичні моделі персонажів.

У 2000-х роках зростання продуктивності графічних систем дозволило розробникам активно використовувати детальні моделі, складні шейдери та динамічні ефекти. З’являються такі ігри, як Half-Life 2, Far Cry, Doom 3, що значно розширили межі можливого завдяки фізично правильному освітленню, нормальних

та паралаксних мапах, а також інтеграції реалістичної фізики. У цей час розвивається ще один важливий інструмент — Unity, який зробив розробку 3D-ігор доступною не лише великим студіям, а й незалежним розробникам.

Станом на сьогодні розвиток 3D-ігор пов’язаний із системами трасування променів у реальному часі, штучним інтелектом та складними фізичними симуляціями. Індустрія рухається у напрямку реалістичності, інтерактивності та оптимізації, а рушії Unity і Unreal Engine надають інструменти, які дозволяють реалізовувати технології, раніше доступні лише у великих студіях. Таким чином, історія 3D-ігор є прикладом швидкої еволюції, у якій сукупність апаратних інновацій та програмних рішень створила фундамент сучасних інтерактивних середовищ.

Подальший розвиток тривимірних відеоігор у 1990-х роках тісно пов’язаний із становленням спеціалізованих графічних API, які забезпечили розробникам спільні стандарти рендерингу та дали змогу уніфікувати роботу з різними відеокартами. Зокрема, поява OpenGL та пізніше Direct3D дала можливість створювати ігрові рушії, що використовували апаратне прискорення, а не покладалися на програмні методи формування зображення. Це стало критичним фактором переходу індустрії до повноцінного полігонального 3D. Оскільки графічні процесори стали виконувати більшість обчислень, пов’язаних із трансформаціями та освітленням, ігрові рушії змогли зосередитися на моделюванні фізики, поведінки персонажів та інтерактивності.

Поступове проникнення відеокарт у масовий сегмент сприяло зростанню конкуренції між виробниками графічних прискорювачів, такими як 3dfx, NVIDIA та ATI. Вихід 3dfx Voodoo у 1996 році фактично змінив уявлення про якість графіки в реальному часі. Вона підтримувала текстурні фільтри, Z-буфер та інші функції, що дозволили відмовитися від каркасної або плоскої рендеризації. З цього моменту ігри починають виглядати справді об’ємними, а ілюзія тривимірного простору стає переконливішою. Рушії, створені на основі можливостей відеокарт, стали набагато гнучкішими та дозволили реалізувати складні механіки, що раніше були

недоступні, наприклад реалістичне освітлення, згладження країв, різноманітні типи матеріалів і динамічні тіні.

Важливою віхою розвитку стала поява інструментів, що дозволили розробникам створювати складні сцени без необхідності писати власний рушій з нуля. Unreal Engine першого покоління запропонував вбудований редактор рівнів, який дав змогу дизайнеру створювати середовище гри без програмного втручання. На відміну від попередніх командних інтерфейсів, нові редактори з графічним інструментарієм зробили процес розробки більш інтуїтивним, демократизували внутрішні інструменти та дозволили збільшити роль художників. У той самий період зростає поширення модифікацій — користувачі створювали власні рівні, персонажів та навіть нові ігри на основі рушіїв Quake та Half-Life. Це стало однією з причин популярності цих технологій і суттєво вплинуло на формування ігрової культури.

У 2000-х роках розвиток тривимірної графіки прискорився завдяки впровадженню програмованих шейдерів. До цього графічні прискорювачі мали фіксовану функціональність, яка обмежувала можливості художнього оформлення та фізичних ефектів. Завдяки шейдерам розробники отримали змогу створювати власні алгоритми освітлення, тіні, води, скла, поверхонь із підповерхневим розсіюванням та інших складних матеріалів. Саме у цей період з’являються перші фотореалістичні сцени, які змінили сприйняття 3D-ігор як чогось умовного і брутального. Стандарти OpenGL 2.0 і DirectX 9 визначили подальший напрямок розвитку індустрії, оскільки надали можливість моделювати навіть найскладніші ефекти в реальному часі.

При цьому тісно зростала і складність виробництва ігор. Щоб відповідати новим стандартам, розробникам доводилося створювати величезні об’єми контенту, включно з деталізованими моделями, текстурами високої роздільної здатності, розвиненими анімаціями та складними фізичними сценами. Відповідно, великі студії почали впроваджувати спеціалізовані інструменти симуляції — наприклад, Havok або PhysX. Ці системи дозволили значно спростити реалізацію таких явищ, як зіткнення, падіння предметів, взаємодія об’єктів і поведінка

персонажів у тривимірному просторі. Важливо зазначити, що саме інтеграція фізики у 3D-середовище стала ключовим фактором у створенні сучасних шутерів, платформерів та ігор відкритого світу.

Ближче до 2010-х років тривимірні ігри пережили ще одну технологічну революцію, пов’язану з переходом на нові покоління консолей і поширення якихось «загальних» рушіїв. Unity та Unreal Engine стали універсальними стандартами, доступними широкому колу користувачів. Якщо раніше для створення 3D-ігор потрібні були значні технічні знання, то тепер через появу зручних інструментів розробці могли навчитися навіть початківці. Unity істотно спрощував створення прототипів, а його архітектура на базі компонентів дозволяла швидко комбінувати поведінки різних об’єктів. Unreal Engine став відомим завдяки своїй графічній потужності та системі Blueprint, яка дозволила програмувати ігрову логіку без глибоких знань мов програмування. У результаті індустрія стала інклюзивнішою, а кількість інді-ігор збільшилася в десятки разів.

Сьогодні розвиток 3D-ігор є поєднанням кількох векторів. З одного боку, триває рух до фотореалізму, який став можливим завдяки трасуванню променів у реальному часі, глобальному освітленню, фізично коректним матеріалам та алгоритмам постобробки. З іншого боку, паралельно розвивається прагнення до оптимізації й адаптації ігор на різні платформи — від потужних ПК до мобільних пристроїв та VR/AR систем. Зростання обчислювальних можливостей дозволило реалізовувати симуляції складних природних явищ, поведінку штучного інтелекту, великий відкритий світ із тисячами активних об'єктів. А використання хмарних технологій відкриває перспективи зі створення ігор, де частина обчислень здійснюється не на пристрої користувача, а на сервері.

Подальший розвиток 3D-ігор у другій половині 2010-х та на початку 2020-х років значною мірою визначався переходом від традиційних рендерингових підходів до фізично коректних моделей освітлення та складних симуляцій, які дозволили досягти нового рівня реалістичності. Застосування методів фізично коректного рендерингу (Physically Based Rendering, PBR) стало стандартом у сучасних рушіях і передбачало використання моделей взаємодії світла з поверхнею,

що базуються на законах фізики, а не на емпіричних алгоритмах. У результаті матеріали, створені художниками, почали виглядати однаково достовірно за різного освітлення, що суттєво змінило підхід до побудови візуальної частини ігор. Єдиний набір текстур — albedo, metallic, roughness, normal map — став основою художнього контенту.

У цей період рушії Unity і Unreal Engine перетворилися на повноцінні екосистеми, що не лише пропонували засоби створення графіки та фізики, а й інтегрували поведінкові системи для персонажів, синхронізацію об’єктів у мережі, системи анімації та інструменти постобробки. Якість зображення стала близькою до рівня кінематографу, зокрема завдяки технологіям глобального освітлення, які моделюють взаємодію світла між об’єктами сцени. Розвиток апаратних засобів, зокрема появи відеокарт архітектури NVIDIA RTX, дозволив застосовувати трасування променів у реальному часі, що значною мірою скоротило розрив між офлайн-графікою та ігровими рушіями.

Ще однією важливою віхою стала поява віртуальної та доповненої реальності. VR/AR-технології створили нові вимоги до рендерингу, фізики та взаємодії з об’єктами, оскільки для забезпечення комфортного сприйняття 3D-простору потрібно формувати зображення з високою частотою кадрів, точно відстежувати рухи користувача та забезпечувати мінімальну затримку між дією та реакцією віртуального середовища. Це підштовхнуло розробників рушіїв до оптимізації та переосмислення базових методів візуалізації, зокрема впровадження багатопрохідного рендерингу, відсікання частин сцени, які не видно камері, та адаптивних методів генерації кадрів. Деякі принципи VR-рендерингу стали використовуватися і в класичних 3D-іграх, дозволяючи досягати стабільності та кращої продуктивності на широкому спектрі обладнання.

Індустрія 3D-ігор також тісно пов’язана з розвитком процедурної генерації, яка зменшує витрати часу на створення контенту та дозволяє отримувати складні структури світу на основі алгоритмів. Хоча перші спроби процедурного моделювання з'явилися давно, лише сучасні обчислення дозволили застосовувати ці підходи на повну силу. Ігри з великими відкритими світами почали

використовувати гібридні рішення, де частина контенту створювалася вручну, а частина генерувалася автоматично. Рушії стали пропонувати вбудовані інструменти для процедурної генерації ландшафтів, рослинності, погодних умов, динамічного освітлення та навіть поведінки NPC. Ці технології дозволили суттєво збільшити масштаб і різноманітність віртуальних всесвітів.

Крім того, у 2020-х роках велике значення набула оптимізація співпраці всередині команд розробників. Великі проєкти стали вимагати чітких інструментів для інтеграції роботи програмістів, дизайнерів, художників та тестувальників. Сучасні рушії, такі як Unity, забезпечують можливість паралельного редагування сцен, синхронізації ресурсів через системи контролю версій та використання хмарних сервісів для переробки даних. Такий підхід дозволяє створювати складні сцени з високою роздільною здатністю, великою кількістю об'єктів та детальною фізичною симуляцією.

Окремої уваги заслуговує розвиток фізики в 3D-іграх, який має прямий зв’язок із темою цієї магістерської роботи. Фізичні симуляції стали невід’ємною частиною інтерактивності завдяки появі бібліотек Havok, PhysX, Bullet, які дозволили об’єктам поводитися відповідно до реальних законів механіки. У сучасних ігрових рушіях фізика стала складною багаторівневою системою, яка включає моделювання твердих тіл, м’яких тіл, рідин, газів, тканин, а також складних ієрархій колізій. Гравець отримує змогу взаємодіяти з предметами, що поводяться природно, що значно підвищує рівень занурення у віртуальний світ. Актуальні підходи передбачають використання окремих фізичних сцен, оптимізованих зіткнень, інтегрування реалістичної гравітації та точного розрахунку імпульсів, що робить поведінку об'єктів передбачуваною та логічною.

У цей же час розвивається і мережевий компонент тривимірних ігор. Багатокористувацькі середовища вимагають синхронізації дій між клієнтами, врахування затримок у передачі даних, корекції помилок та постійної підтримки консистентності світу. Сучасні фреймворки, такі як Photon PUN 2, дозволяють розробникам зосередитися на логіці гри, а не на побудові низькорівневої мережевої інфраструктури. Наявність таких інструментів зробила можливим створення навіть

невеликими командами комплексних мережевих шутерів та симуляцій, подібних до тієї, яка розробляється в межах цієї роботи.

1.2 Аналіз сучасних ігрових рушіїв (Unity, Unreal Engine, Godot)

Сучасні ігрові рушії є фундаментальною основою процесу розробки тривимірних ігор, оскільки вони забезпечують інтеграцію графіки, фізики, анімації, логіки, мережевої взаємодії та інтерфейсних компонентів у єдину програмну екосистему. Еволюція рушіїв упродовж останніх десятиліть призвела до появи універсальних інструментів, які дозволяють створювати проєкти будь-якої складності — від невеликих інді-ігор до масштабних AAA-продуктів. Найбільш впливовими серед сучасних рушіїв вважаються Unity, Unreal Engine та Godot, кожен з яких має власну архітектуру, сильні сторони, унікальні інструменти та сфери застосування.

Починаючи аналіз із Unity, варто підкреслити, що він став одним із найбільш популярних рушіїв завдяки доступності, простоті входження та широкій екосистемі. Його головною особливістю є компонентна архітектура, де будь-який об’єкт сцени складається з набору компонентів, кожен з яких відповідає за певну функцію — фізичну взаємодію, рендеринг, анімацію, мережеву синхронізацію та інші аспекти. Такий підхід дозволяє розробникам швидко створювати прототипи, комбінуючи поведінки об’єктів без необхідності створення складних класових структур. Скриптування в Unity здійснюється мовою C#, що забезпечує високу продуктивність, строгість типів і зручність роботи з даними. Інтеграція з інструментами на кшталт Visual Studio чи Rider робить розробку зручною та ефективною. Unity також пропонує великий набір фізичних модулів, зокрема PhysX від NVIDIA, що забезпечує реалістичну взаємодію твердих тіл, колізій, тригерів і механік руху. Це є важливою перевагою для проєктів, де фізична симуляція відіграє ключову роль, як у випадку тривимірного шутера, що створюється в межах цієї магістерської роботи.

Ще однією сильною стороною Unity є його багатоплатформність. Рушій надає можливість збирати проєкти для широкого спектра платформ: Windows, Linux, macOS, Android, iOS, WebGL, консолей, а також VR/AR-пристроїв. Така універсальність робить Unity оптимальним вибором не лише для початківців, але й для студій, які прагнуть створювати ігри для різних цифрових екосистем. Крім того, магазин Unity Asset Store став одним із найбільших ресурсів ігрових компонентів, моделей, матеріалів, анімацій і кодових рішень, що значно пришвидшує розробку й зменшує витрати на створення великого об'єму контенту. Усе це перетворює Unity на інструмент, що однаково добре підходить для 2D-, 3D-, мобільних, VR-проєктів та мережевих ігор.

На противагу Unity, рушій Unreal Engine часто асоціюється з проєктами високого рівня графічної складності. Його історія починається ще з 1990-х років, але саме сучасні версії, починаючи з Unreal Engine 4 і особливо Unreal Engine 5, визначили його як один із технологічних лідерів індустрії. Епохальним стало впровадження системи фізично коректного рендерингу та сучасних матеріальних моделей, а з виходом UE5 рушій отримав такі інновації, як Nanite та Lumen. Nanite дозволяє працювати з мільярдами полігонів, що фактично усуває потребу у ручному створенні рівнів деталізації (LOD), а Lumen реалізує глобальне освітлення в реальному часі, наближаючи якість рендерингу до кіноіндустрії. Такі інструменти роблять Unreal Engine оптимальним вибором для фотореалістичних симуляцій, великих світів та ігор, орієнтованих на графічну досконалість. Ще однією характерною рисою Unreal Engine є використання системи Blueprint — візуальної мови програмування, що дозволяє створювати поведінку об'єктів і логіку гри без написання коду. Це відкриває доступ до розробки не лише програмістам, але й дизайнерам і художникам, забезпечуючи гнучкість та прискорюючи створення прототипів. Проте рушій також підтримує мову C++, яка забезпечує високий рівень контролю над продуктивністю і внутрішніми процесами.

На відміну від Unity та Unreal Engine, рушій Godot набув популярності як повністю відкритий та вільний інструмент. Його архітектура базується на системі сцен і вузлів, що дозволяє створювати складні ієрархічні структури. Godot

підтримує кілька мов програмування, зокрема GDScript — власну мову, синтаксично схожу на Python, а також C# і C++. Важливою перевагою є повна відкритість коду рушія, що дозволяє розробникам змінювати внутрішню логіку під потреби проєкту. Godot також є легшим за ресурсами, що робить його привабливим вибором для інді-розробників та освітніх проєктів. Хоча Godot традиційно вважався менш потужним у порівнянні з Unity та Unreal Engine у сфері тривимірної графіки, останні версії суттєво покращили рендеринг, включно з підтримкою Vulkan та сучасних матеріальних моделей. Рушій активно розвивається спільнотою, що дозволяє йому постійно наближатися до рівня великих конкурентів.

Подальший аналіз сучасних ігрових рушіїв вимагає глибшого розуміння їхніх архітектурних принципів, оскільки саме вони визначають можливості розробника у створенні 3D-ігор. Важливо зазначити, що рушії не просто генерують графіку чи забезпечують фізичні обчислення — вони формують цілу екосистему, яка охоплює роботу з ресурсами, оптимізацію, компонентну систему, анімацію, мережеву взаємодію, обробку звуку та інші важливі аспекти. Тому вибір рушія визначає не лише технологічний підхід, але й структуру самої гри, способи взаємодії між ігровими об’єктами та методи реалізації фізичної симуляції, що є безпосередньо актуальним для даної магістерської роботи.

Unity у цьому контексті часто сприймається як рушій, орієнтований на швидкість розробки, гнучкість і модульність. Його використання в освітніх проєктах, інді-розробці та навіть у великих комерційних продуктах пов’язане з тим, що він дозволяє створити повноцінну ігрову систему без необхідності формування низькорівневих алгоритмів. Наприклад, для визначення фізичної поведінки об’єкта достатньо додати компонент Rigidbody та задати його параметри. Взаємодія між об’єктами реалізується через колайдери та події, які обробляються за допомогою стандартних функцій, таких як OnCollisionEnter чи OnTriggerStay. Такий підхід мінімізує потребу розробника у розв’язанні базових фізичних задач і дозволяє зосередитися на прикладних механіках гри. Це особливо важливо в багатокористувацьких іграх, де фізична поведінка має бути стабільною, передбачуваною та узгодженою між різними клієнтами. У межах Unity також

активно використовується система префабів — заготовок об’єктів, які можна повторно використовувати в різних сценах і застосовувати до них зміни централізовано. Префаби спрощують проєктування складних ігрових структур, оскільки дозволяють визначати усю логіку та компоненти об’єкта в одному місці. Для мережевих проєктів, таких як гра, розроблена в рамках цієї магістерської роботи, префаби відіграють ключову роль у синхронізації створення та знищення об’єктів у мережі: Photon, наприклад, використовує саме префаби для коректного створення інстансів між клієнтами.

Аналіз архітектури Unreal Engine демонструє інший підхід, орієнтований на максимальну продуктивність та роботу з великими обсягами даних. У UE реалізована система Actor–Component, що також передбачає поділ логічних компонентів об’єкта, проте сам рушій використовує значно складнішу ієрархію та внутрішні механізми. Unreal Engine тісно пов’язаний із мовою C++, яка використовується для низькорівневого доступу до системи, оптимізації та створення високопродуктивних рішень. Попри це, можливості Blueprint надають зручний інструмент для швидкої розробки логіки, дозволяючи поєднати код і візуальне програмування. Порівнюючи Unreal Engine з Unity, можна зазначити, що UE зазвичай використовують для великих проєктів, де важливо досягти кінематографічної якості графіки, складного освітлення, густо населених сцен і високоінтенсивної фізичної взаємодії. Водночас Unity частіше використовується в проєктах, де важливими є швидкість, адаптивність і простота розробки.

Godot, у свою чергу, позиціонує себе як рушій, який поєднує простоту, гнучкість і відкритість. Він не залежить від зовнішніх корпоративних рішень, що робить його привабливим для академічних і дослідницьких проєктів. Його вузлова структура дозволяє створювати сцени, які складаються з дрібних елементів—вузлів—кожен із яких має власну поведінку. Це робить Godot концептуально схожим на Unity, але з більшою гнучкістю у зміні внутрішніх механізмів рушія. Важливо також відзначити, що Godot використовує легку інтерпретовану мову GDScript, завдяки якій розробка стає доступною навіть початківцям. Хоча 3D-можливості Godot поки що дещо відстають від двох великих конкурентів, останні

версії демонструють значний прогрес у напрямку якісного рендерингу, фізики та оптимізації.

Говорячи про фізичні можливості рушіїв у контексті симуляції об’єктів, варто зазначити, що Unity та Unreal Engine покладаються на PhysX як базовий фізичний рушій для моделювання твердих тіл, колізій і динаміки. Unreal Engine має додаткові власні розширення, такі як Chaos Physics, які дають змогу створювати масштабні фізичні ефекти, руйнування, тканини, рідини та інші типи симуляцій. Godot використовує власну фізичну систему, яка хоч і менш оптимізована, проте дозволяє гнучко змінювати й адаптувати модулі під специфічні вимоги.

Сучасні ігрові рушії також істотно різняться підходами до оптимізації, обробки освітлення та управління пам’яттю. Unreal Engine робить ставку на потужні технології глобального освітлення, процедурну генерацію контенту та системи потокового завантаження ресурсів, що дозволяє створювати великі світи з високою точністю. Unity натомість підкреслює гнучкість налаштування освітлення, можливість вибору різних рендер-пайплайнів і створення ігор для широкого спектра пристроїв, включно з мобільними. Godot зосереджений на простоті та універсальності, зберігаючи баланс між продуктивністю та доступністю. Подальший детальний розгляд сучасних рушіїв вказує на те, що їх порівняння не може обмежуватися лише переліком функцій чи загальними характеристиками. Кожен із них сформував власний підхід до організації розробки ігор, а відмінності у цих підходах впливають не тільки на результат, але й на сам процес створення 3D-проєкту, на вибір архітектурних патернів, на оптимізацію фізики та на те, наскільки легко реалізовуються такі механіки, як симуляція взаємодії об’єктів, мережевий режим або динамічне освітлення.

Unity у цьому контексті пропонує підхід, орієнтований на компонентну архітектуру, яка зробила рушій надзвичайно зручним для широкого кола розробників — від початківців до студій, які створюють мобільні чи VR-проєкти. Основою є концепція GameObject, до якого можна під’єднувати довільну кількість компонентів: фізичних, логічних, графічних або службових. Кожен компонент працює незалежно, але водночас може взаємодіяти з іншими через механізм

посилань або залежностей. Такий підхід дозволяє розробляти гру поступово, нарощуючи її складність, і легко оновлювати або переробляти окремі модулі без необхідності змінювати всю систему. Це стало важливим фактором для вибору Unity під час розробки гри з фізичною симуляцією, оскільки робота з Rigidbody, Collider, матеріалами фізики та скриптами поведінки є інтуїтивно зрозумілою та добре задокументованою.

Крім того, Unity має сильну екосистему інструментів, яка також впливає на зручність розробки. Asset Store, наприклад, дає можливість використовувати готові моделі, анімації, шейдери, інструменти оптимізації та навіть цілі системи керування персонажем або мережеві фреймворки. Це значно скорочує час розробки, дозволяє зосередитися на унікальних механіках гри і водночас мінімізує ризики помилок у базовому функціоналі. Оскільки у цій магістерській роботі реалізована також мережеві складова, екосистема Unity слугувала додатковою перевагою: Photon PUN повністю інтегрований у Unity, підтримує префаби, взаємодіє з компонентами ігрових об’єктів, працює з подіями сцени та легко поєднується з UI Toolkit, що забезпечує гнучке та стабільне середовище для створення багатокористувацької гри.

У випадку Unreal Engine структура роботи суттєво відрізняється. Рушій пропонує не тільки компонентну модель, але й більш низькорівневу інтеграцію C++, що дає змогу розробнику втручатися в самі основи роботи рушія. Завдяки цьому Unreal Engine часто використовується у великих комерційних проєктах, де є необхідність оптимізувати кожен аспект гри та забезпечити максимально можливу якість графіки та фізики. Система Blueprints стала однією з ключових інновацій UE, оскільки вона дозволяє візуально програмувати складну логіку, використовуючи моделі вузлів і зв’язків між ними. Таким чином Unreal Engine поєднує глибину і високу продуктивність C++ з доступністю та наочністю візуального програмування. Проте для менш масштабних проєктів, особливо для навчальних або експериментальних, Unreal може виявитися надмірно складним через велику кількість систем, що потребують ретельного налаштування.

Godot на фоні Unity та Unreal Engine виділяється радикально іншою філософією. Він є повністю відкритим рушієм, що дозволяє розробникам змінювати його ядро, оптимізувати рендер або фізику відповідно до власних потреб. Сам рушій побудований на вузловій архітектурі, яка робить структуру сцени гнучкою і природною для розуміння. Немає необхідності розділяти об’єкти на префаби, геймооб’єкти та компоненти — будь-який елемент сцени є вузлом і може наслідуватися, модифікуватися або бути частиною більш складних структур. Це створює ефективну систему навіть для великих сцен. Однак у 3D-розробці Godot довгий час поступався своїм конкурентам у плані графіки, оптимізації рендеру та фізики. З появою Godot 4 ситуація значно покращилася, проте рушій усе ще більше орієнтований на інді-проєкти, навчальні завдання та експерименти, а не на високопродуктивні комерційні ігри.

Важливо також зазначити, що у контексті фізичної симуляції рушії реалізують абсолютно різні підходи. Unity пропонує PhysX як основний фізичний двигун для твердих тіл. Він забезпечує стабільність, реалістичність і передбачуваність фізики, особливо в проєктах, де об’єкти повинні взаємодіяти з багатьма різнорідними елементами середовища. Unreal Engine, окрім PhysX, розробив власну систему Chaos Physics, що дозволяє створювати надзвичайно складні ефекти руйнувань, симуляції м’яких тіл, тканин, вибухів і масивних динамічних систем. Це дає розробнику можливість створити фізичне середовище максимально близьке до реального. Godot використовує свій фізичний рушій, який хоч і менш складний, але достатньо ефективний для середніх за масштабом проєктів і дозволяє вручну налаштовувати багато складових фізичних обчислень.

Порівнюючи ці рушії у практичному аспекті, можна зробити висновок, що Unity забезпечує найбільш збалансоване середовище для розробки 3D-ігор із фізичною симуляцією. Він поєднує доступність, зрозумілу компонентну модель, стабільну фізику та багату екосистему інструментів. Unreal Engine є оптимальним вибором для проєктів із надвисокими вимогами до графіки та фізики, але потребує значно більше часу та досвіду. Godot пропонує повну свободу, відкритість і

простоту, але не завжди може забезпечити той же рівень продуктивності та стабільності, що є необхідним у реалістичних 3D-іграх.

1.3 Особливості мови програмування C# у контексті розробки ігор

Мова програмування C# посідає особливе місце в індустрії розробки ігор, насамперед завдяки своїй тісній інтеграції з ігровим рушієм Unity, який є одним із найпопулярніших інструментів створення 2D і 3D ігор. Вибір C# у контексті розробки ігрових застосунків зумовлений низкою факторів, що включають зручність синтаксису, об’єктно-орієнтовану природу мови, розгалужену екосистему бібліотек, високу продуктивність і стабільність, а також широкий спектр можливостей для взаємодії з різними системами рушія. У поєднанні з Unity C# перетворився на один із найбільш універсальних інструментів для створення прототипів та повноцінних ігрових продуктів будь-якого масштабу. Історично C# був створений компанією Microsoft як частина платформи .NET із метою забезпечити сучаснішу, безпечнішу та більш структуровану альтернативу C++ і Java. Завдяки своїй об’єктно-орієнтованій природі мова стала популярною серед розробників широкого профілю, оскільки пропонувала інструменти для чіткого опису поведінки об’єктів, їхніх властивостей, станів і методів взаємодії між ними. У контексті ігор ця парадигма виявилася надзвичайно ефективною, оскільки більшість елементів ігрового світу — персонажі, предмети, компоненти середовища, снаряди та механізми — природно описуються у вигляді об’єктів із внутрішньою логікою.

Однією з найважливіших причин, чому C# став стандартом для Unity, є оптимальний баланс між продуктивністю та простотою розробки. Завдяки JIT-компіляції код у C# виконується із високою швидкістю, що робить його придатним для створення проєктів у реальному часі, зокрема шутерів, платформерів, симуляторів та VR-додатків. У той же час синтаксис мови залишається легким для розуміння, що дозволяє навіть початківцям швидко освоїтися та перейти до реалізації складних систем. C# активно використовується у навчальному

середовищі саме через свою доступність, а також через величезну кількість документації та навчальних матеріалів, що формують сприятливе середовище для розвитку початківців.

У контексті Unity C# набуває ще більшої вагомості, оскільки мова забезпечує доступ до всіх систем рушія. Компонентна модель Unity побудована таким чином, що кожен скрипт у C# є компонентом, який можна додати до будь-якого об’єкта сцени. Все, що налаштовується у вікні інспектора Unity, може бути пов’язано з полями класу у C#, що забезпечує гнучку прив’язку логіки до візуальних елементів гри. Система MonoBehaviour у C# дає змогу використовувати спеціальні методи життєвого циклу — такі як Start(), Update(), FixedUpdate(), OnCollisionEnter() — які визначають поведінку об’єкта в різних моментах ігрового процесу. Кожна така функція є частиною загальної архітектури Unity, що дозволяє розробнику інтуїтивно управляти рухом об’єктів, фізикою, анімацією, взаємодією з користувачем і багатьма іншими аспектами гри. Важливо, що C# дозволяє працювати з різними типами даних, узагальненнями, делегатами, подіями та асинхронними операціями. У іграх ці можливості відкривають шлях до реалізації складних систем штучного інтелекту, систем інвентарю, багатокористувацьких режимів, керування станом гри або здійснення оптимізації через поділ логіки на незалежні частини. Крім того, C# дозволяє ефективно працювати з математичними структурами, такими як вектори, кватерніони, матриці трансформації, що є надзвичайно важливими у 3D-графіці.

У контексті фізичної симуляції C# надає зручний набір API для роботи з компонентами PhysX у Unity. Розробник може легко керувати силами, крутними моментами, масою, коефіцієнтами тертя, обробляти зіткнення і тригери. Усе це відбувається без необхідності заглиблюватися у низькорівневий код рушія, оскільки Unity надає високорівневі методи, які приховують складність внутрішньої фізичної моделі. Синтаксис C# дозволяє описувати такі обчислення чисто й ефективно, що особливо важливо для шутерів або ігор жанру action, де необхідно забезпечити швидку реакцію на взаємодію об’єктів. Окремої уваги заслуговує система управління пам’яттю у C#. На відміну від C++, де розробник має вручну керувати

виділенням і очищенням пам’яті, C# використовує автоматичний збирач сміття (Garbage Collector). Це значно знижує ймовірність помилок, пов’язаних із витоками пам’яті, та робить проєкт більш стабільним. Хоча використання GC потребує розумного підходу до оптимізації та уникнення надмірного створення об’єктів у циклах, загалом воно робить мову безпечнішою та більш передбачуваною.

Подальший розгляд особливостей мови програмування C# у сфері розробки ігор дає змогу глибше зрозуміти її значення для сучасних ігрових рушіїв, зокрема Unity, де ця мова відіграє безпосередню роль у формуванні всієї поведінкової логіки об’єктів та контролю процесів у грі. Сучасна структура C# дозволяє розробнику реалізовувати як найпростіші алгоритми, так і складні системи, зберігаючи при цьому високу читабельність та структурованість коду. Завдяки своїй архітектурній продуманості C# поєднує властивості статично типізованої мови з розширеною системою безпеки, що стало однією з причин її поширення в галузі ігрової індустрії.

У контексті Unity характерною рисою C# є те, що він працює не ізольовано, а в межах комбінації з платформою .NET, яка забезпечує доступ до величезної кількості бібліотек та інструментів. Це доповнення дозволяє розширювати функціональність гри, інтегруючи різні підсистеми, обробку даних, математичні розрахунки, роботу з файлами, мережеву логіку та інші можливості, що робить взаємодію з ігровим рушієм щонайменше універсальною. Усе це забезпечує створення систем будь-якого рівня складності без необхідності використовувати сторонні мови або фреймворки. Саме в середовищі Unity ця універсальність C# проявляється найбільш яскраво, оскільки рушій надає доступ до великої кількості API, що працюють безпосередньо через C#.

Окремим аспектом є те, як C# реалізує об’єктно-орієнтований підхід у структурах Unity. Кожен елемент у грі може бути представлений як клас, що успадковується від базового класу MonoBehaviour. Це дозволяє не лише формувати поведінку об’єктів, але й визначати їхній стан, взаємодію та життєвий цикл. Така модель роботи дає можливість розбивати логіку гри на невеликі модулі, які легко тестувати та підтримувати. Наприклад, рух персонажа, фізичні взаємодії, стрільба,

обробка UI або реакція на події можуть реалізовуватися як повністю незалежні компоненти. Це відповідає принципам чистої архітектури, де кожен модуль містить чітко визначену відповідальність, що значно спрощує структуру проєкту, роблячи його менш схильним до помилок та конфліктів.

Перевагою C# є також те, що він підтримує широкий спектр інструментів для роботи з асинхронністю, таких як async/await. У розробці ігор це має велике значення, оскільки дозволяє виконувати тривалі операції — завантаження ресурсів, підготовку сцен, отримання даних з мережі — без блокування основного потоку, який відповідає за рендеринг та фізичні обчислення. Асинхронні методи підвищують плавність роботи гри, роблять UI більш чутливим до дій користувача і дозволяють організувати складні взаємодії між компонентами гри без ризику зависання або падіння продуктивності. У багатокористувацьких іграх, де постійний обмін даними із сервером є необхідним, це стає незамінним інструментом. Система подій та делегатів є ще одним важливим елементом, який робить C# ефективним у розробці ігор. Делегати дозволяють організовувати зворотні виклики та формувати моделі взаємодії між об’єктами без прямої прив’язки до конкретних класів. Це надає значну гнучкість у побудові архітектури гри. Наприклад, система збору предметів або система отримання урону може бути реалізована таким чином, що один компонент генерує подію, а інші можуть підписуватися на неї. Це в значній мірі підвищує розширюваність та адаптивність гри.

Серед важливих можливостей C# також варто відзначити роботу зі складними структурами даних. Використання списків, словників, масивів, черг та стеків дозволяє будувати системи зберігання інформації про об’єкти, відстеження їхнього стану або організацію управління потоками даних. У розробці ігор це відіграє критично важливу роль, оскільки такі структури застосовуються для роботи зі списками противників, пулів об’єктів, таблиць зіткнень, систем збереження гри або інвентарю. Значною перевагою мови C# є її безпечність. Завдяки суворій типізації та механізму обмеження доступу до пам’яті розробник може уникнути багатьох критичних помилок, які часто виникають у проєктах на C++. Хоча C++ забезпечує максимальний контроль за пам’яттю, він також відкриває можливість низки

помилок, таких як сегментаційні збої чи неправильна робота з покажчиками. У C# ці проблеми мінімізовані, що робить процес розробки більш стабільним і захищеним. Завдяки цьому розробник може зосередитися на логіці гри, а не на низькорівневих технічних моментах.

У підсумку C# поєднує високу продуктивність, структурованість, читабельність та безпеку, що робить його ідеальним вибором для розробки ігор на Unity. Мова дозволяє будувати складні ігрові системи, ефективно працювати з фізичними процесами, керувати 3D-сценою, реалізувати мережеву взаємодію та забезпечувати чітку взаємодію між елементами гри. У межах цього проєкту саме C# став основою реалізації фізичної симуляції, контролю гравця, обробки стрільби, взаємодії з UI та мережевих механік.

Завершальна частина аналізу особливостей мови програмування C# у сфері розробки ігор зосереджується на практичних аспектах її застосування та глибинних механізмах, які визначають ефективність її використання в ігрових рушіях, насамперед у Unity. Важливо підкреслити, що популярність C# у цій галузі не є випадковою: мова була спроєктована таким чином, щоб поєднати простоту, гнучкість і високу продуктивність, а середовище Unity адаптувало її настільки органічно, що сьогодні C# фактично став стандартом інструментарію для більшості проєктів на цьому рушії – від найпростіших прототипів до повноцінних комерційних ігор.

Одним із важливих чинників, що забезпечують ефективність C# у розробці ігор, є його тісна інтеграція з компонентною архітектурою Unity. Усі об’єкти у рушії побудовані на основі системи компонентів, що дозволяє створювати складні об’єкти шляхом комбінації різних скриптів, написаних на C#. Кожен компонент має свій набір обов’язкових методів життєвого циклу — Awake, Start, Update, FixedUpdate, LateUpdate, OnCollisionEnter тощо. Ця модель роботи змушує розробника мислити поведінкою об’єктів, що відповідає природному баченню структури гри. У результаті логіка стає не лише більш прозорою, але і логічно впорядкованою, адже кожен фрагмент коду виконує конкретну роль у межах одного компонента.

Слід окремо наголосити на тому, яке значення має система керування пам’яттю. C# працює з автоматичним збиранням сміття, що дозволяє уникати значної частини проблем, характерних для мов із ручним керуванням пам’яттю. У розробці ігор це є особливо важливим, оскільки зростання кількості об’єктів під час виконання, часті створення та видалення елементів, а також динамічні зміни станів можуть призвести до неконтрольованого витоку пам’яті. Автоматизована система очищення гарантує, що непотрібні об’єкти будуть видалені, знижуючи ризик падіння продуктивності. Водночас важливо грамотно оптимізувати частоту викликів збирання сміття, оскільки надмірна кількість об’єктів, що швидко створюються і видаляються, може викликати затримки. Це показує, що хоча мова і бере на себе значну частину технічної роботи, розробнику необхідно розуміти її внутрішні механізми, щоб отримати максимальну ефективність.

Ще однією суттєвою рисою C# є можливість розробляти ігрову логіку з використанням шаблонів проєктування, що дозволяють будувати масштабовані системи. У розробці ігор часто застосовують такі шаблони, як Singleton, Observer, Factory, Object Pooling, які дають змогу будувати складні структури без втрати керованості. Наприклад, у багатокористувацьких проєктах часто використовується шаблон Singleton для створення глобальних менеджерів, таких як GameController або NetworkManager, які повинні існувати лише в одному екземплярі. Так само патерн Object Pooling зменшує навантаження на пам'ять, оскільки дозволяє повторно використовувати об’єкти, наприклад кулі чи гільзи, що особливо актуально для шутерів, де кількість створюваних снарядів може сягати сотень за хвилину. У багатокористувацьких іграх, таких як у рамках даного проєкту, C# є ключовим інструментом для реалізації мережевої взаємодії. Photon PUN, який використовується у проєкті, працює як надбудова над C#, надаючи доступ до механізмів RPC, синхронізації станів та керування кімнатами. Усі ці функції реалізуються шляхом викликів методів, подій і класів, написаних саме на C#. Завдяки цьому логіка мережевої взаємодії стає інтуїтивною: розробник може обробляти події входу гравців до кімнати, реагувати на зміни їхнього стану,

керувати передачею даних, не відходячи від принципів, що їх уже використовує в інших системах гри.

Особливе значення має робота з математичними функціями, які є невід'ємною складовою будь-якого тривимірного проєкту. C# забезпечує доступ до потужних математичних бібліотек, включно зі стандартними функціями .NET та спеціалізованими структурами Unity, такими як Vector3, Quaternion, Matrix4x4 тощо. Вони дають змогу працювати з обертанням об'єктів, визначати напрямки руху, виконувати фізичні розрахунки, що є ключовими для реалізації симуляції. Практично будь-яка взаємодія у грі — від руху гравця до визначення траєкторії польоту снаряда — базується саме на цих математичних інструментах, і їхня ефективність безпосередньо залежить від можливостей C#.

Система обробки винятків також відіграє важливу роль, оскільки дозволяє розробнику локалізувати помилки на ранньому етапі. Ігри є дуже складними системами, у яких помилка в одному компоненті може спричинити збій у цілій структурі. Використання try-catch блоків, логування та вбудованих засобів діагностики допомагають підтримувати стабільність гри на всіх етапах виконання.

1.4 Реалізація фізичних симуляцій у 3D-середовищах

Фізична симуляція є одним із ключових аспектів розробки сучасних 3D-ігор, оскільки саме вона забезпечує вірогідність поведінки об'єктів, відтворення реалістичних рухів та природну взаємодію між елементами ігрового світу. Зрештою, гравець сприймає гру не лише через графічну складову, але й через те, наскільки переконливо поводяться об'єкти, як вони падають, стикаються, відскакують, руйнуються або реагують на зовнішні сили. Саме тому реалізація фізики в іграх є комплексним завданням, що поєднує математичні методи, алгоритми оптимізації, логіку взаємодій і можливості конкретного рушія. У випадку проєкту, створеного в рамках цієї магістерської роботи, фізична модель реалізована на основі вбудованого фізичного рушія Unity — NVIDIA PhysX, який забезпечує точність симуляцій та їх ефективне виконання у реальному часі.

Початково важливо усвідомити, що фізика в ігрових рушіях ніколи не є повноцінною копією фізичних процесів реального світу. Вона завжди є певною апроксимацією, спрощеною моделлю, адаптованою для роботи в умовах обмежених обчислювальних ресурсів. Навіть сучасні відеокарти та процесори не здатні в реальному часі опрацьовувати повні системи нелінійних фізичних рівнянь, які описують реальні взаємодії тіл у тривимірному просторі. Тому кожний рушій використовує власні алгоритми наближення, оптимізації й інтерполяції, що дозволяє досягти балансу між точністю та швидкістю виконання.

У PhysX та Unity фізична симуляція базується на системі RigidBody, яка є фундаментальною частиною для будь-яких фізично активних об’єктів. Коли об’єкт отримує компонент RigidBody, рушій починає враховувати такі параметри, як маса, швидкість, прискорення, інерція та вплив зовнішніх сил. Завдяки цьому стає можливим відтворення реалістичних падінь, ковзання, штовхання, стрибків та інших дій. Поєднання RigidBody з колайдерами — BoxCollider, SphereCollider, MeshCollider та іншими — дозволяє визначати форму тіла та його межі у просторі. Це формує основу для обчислення зіткнень, яке є однією з найскладніших частин фізичного рушія.

Механізм зіткнень у PhysX працює на основі перевірки перетину геометричних примітивів і визначення нормалі та глибини проникнення тіл. Щоб досягти реалістичної поведінки, рушій має не лише виявити факт зіткнення, але й визначити напрямок у якому слід застосувати силу відштовхування, уникнути надмірного проникнення моделей, компенсувати числові похибки та забезпечити стабільність розрахунків у наступних кадрах. Саме тому система зіткнень інколи демонструє специфічні ефекти: легкі посмикування об’єктів, їх "припідняття" над поверхнею або дрібні вібрації при контакті. У реальних іграх це є результатом компромісу між точністю розрахунків і обмеженнями в реальному часі.

Особливу роль у фізичних симуляціях відіграє гравітація. У Unity вона реалізована глобально, через параметр Physics.gravity, який визначає напрямок та силу прискорення, що діє на всі тіла сцени. Це означає, що будь-який об’єкт з компонентом RigidBody автоматично підпадає під вплив гравітації, якщо

користувач не вимкне цю опцію вручну. У більшості ігор гравітація залишається близькою до реального значення, однак у деяких жанрах — наприклад, платформерах або фентезі-проєктах — вона може бути змінена для досягнення більш приємних відчуттів від руху або підкреслення стилістики гри.

Окрім гравітації, важливими елементами фізичної моделі є сили та імпульси. Unity дозволяє застосовувати як постійні сили, що впливають на тіло протягом часу, так і миттєві імпульси, що надають різких змін швидкості. Імпульси часто використовуються в проєктах зі стрілецькими механіками: віддача зброї, поштовхи вибухів, відкидання об’єктів унаслідок удару. У PhysX імпульси додаються через методи AddForce або AddExplosionForce. Саме завдяки цим інструментам розробники можуть створювати ефекти, що виглядають природними та передають відчуття динаміки. Важливим аспектом є те, що фізика у 3D-середовищі не обмежується лише рухом твердих тіл. Вона також охоплює такі системи, як моделювання шарнірних з’єднань, пружин, шарнірних підвісів, ланцюгів та різних механічних конструкцій. Unity містить набір Joint-компонентів, які дозволяють будувати складні механізми: від простих дверей, що обертаються на осі, до повноцінних роботизованих систем з кількома ступенями свободи. Хоча такі системи не використовуються у базових шутерах, вони є основою для багатьох інших жанрів ігор та демонструють гнучкість рушія.

Продовжуючи аналіз фізичних симуляцій у тривимірних середовищах, варто більш детально розглянути особливості алгоритмів, які забезпечують стабільність та передбачуваність роботи фізичного рушія. Це особливо важливо у контексті ігрової розробки, оскільки від правильності налаштування фізичних параметрів залежить не лише плавність руху об’єктів, але й загальне враження від гри. На відміну від наукових симуляцій, які прагнуть до максимальної точності незалежно від обчислювальних витрат, ігрова фізика використовує оптимізовані методи, що дозволяють виконувати симуляцію в реальному часі із гарантованою частотою кадрів. Саме тому велике значення мають такі параметри, як Fixed Timestep, Solver Iteration Count та інші механізми стабілізації.

Одним із ключових аспектів роботи фізичного рушія є розділення логіки оновлення на Update та FixedUpdate. У Unity симуляція фізики здійснюється саме у FixedUpdate, який викликається з фіксованою частотою, незалежно від кількості кадрів на секунду. Це означає, що навіть якщо FPS змінюється, фізичний рушій буде працювати стабільно. У той же час логіка руху, пов’язана зі взаємодією фізичних компонентів, повинна перебувати саме у цьому методі, щоб уникнути розсинхронізації, яка може призвести до так званих "фізичних артефактів", наприклад ривків, некоректних зіткнень або раптових прискорень об’єктів.

Другим важливим аспектом є алгоритм розв’язування контактів, що забезпечує коректність взаємодії між об’єктами. Для кожного зіткнення PhysX виконує серію розрахунків, які включають виявлення точок контакту, обчислення нормалі поверхні, визначення сили відштовхування і компенсації проникнення, а також корекцію швидкостей об’єктів. Точність цих обчислень визначається кількістю ітерацій—силових та швидкісних. У Unity це параметри Solver Iterations та Solver Velocity Iterations. Чим більше їхнє значення, тим точніше симуляція, але тим більше навантаження на процесор. Для ігор із динамічними об’єктами та великою кількістю зіткнень підвищення цих показників може бути критично важливим, оскільки це зменшує ймовірність ситуацій, коли об’єкти провалюються крізь поверхні або проходять один крізь одного на високій швидкості.

Третім важливим елементом фізичної симуляції є система тригерів та взаємодій без фізичного впливу. Колайдери можуть працювати у двох режимах: фізичного зіткнення та тригерного режиму. У тригерному режимі PhysX не застосовує жодних сил, а лише повідомляє систему про факт входу або виходу об’єкта за допомогою подій OnTriggerEnter і OnTriggerExit. Цей механізм використовується у випадках, коли потрібно визначити взаємодію без фізичного впливу: наприклад, підбір предметів, входження у зони дії, активація пасток або перемикачів. Для 3D-шутера, створеного у рамках даної роботи, тригерні зони можуть застосовуватися для визначення областей спавну або активації певних ігрових елементів.

Особливої уваги потребує питання моделювання руху гравця, оскільки саме він є центральним елементом більшості ігрових сценаріїв. На відміну від звичайних фізичних об’єктів, рух гравця повинен бути не лише фізично коректним, але й чітким і передбачуваним. Тому в іграх часто використовують гібридну модель, у якій рух здійснюється через Rigidbody, але частина керування залишається детермінованою, тобто не повністю залежною від фізики. У контексті Unity це може означати ручне керування швидкістю за допомогою MovePosition або обмеження обертання Rigidbody, щоб уникнути небажаних переворотів під час зіткнень. Завдяки цьому рух стає не лише фізично правдоподібним, але і геймплейно комфортним.

Окрему роль відіграє система рейкастів—Raycast, SphereCast, CapsuleCast. На перший погляд може здатися, що стрільба у грі не пов’язана безпосередньо з фізикою, але у більшості випадків саме фізичний рушій займається обчисленням траєкторії "променя" та визначенням того, який об’єкт був уражений. Рейкасти є легкими з точки зору продуктивності, що робить їх основним інструментом, зокрема у шутерах. Вони дозволяють майже миттєво визначити попадання, не створюючи фізичних снарядів, що рухаються у просторі. Хоча у реалістичних симуляціях використовують повноцінні Rigidbody-снаряди, у більшості ігор перевага надається рейкастам через їхню точність і ефективність. Ще один важливий аспект — оптимізація фізичної симуляції. Оскільки кожен фізичний об’єкт вимагає обчислення ряду параметрів, надмірне їх використання може призвести до значних втрат продуктивності. З цієї причини розробники використовують такі підходи, як обмеження кількості активних RigidBody у сцені, заміна складних MeshCollider на примітивні Collider-компоненти, ієрархічне структурування колайдерів та вимкнення симуляції для об’єктів, які не рухаються. У PhysX існують спеціальні механізми "sleeping" та "deactivation", що дозволяють тимчасово деактивувати об’єкти, для яких симуляція не потрібна. Це суттєво зменшує навантаження на процесор, особливо у великих сценах. У завершальній частині аналізу фізичних симуляцій у тривимірних середовищах важливо розглянути не лише технічні механізми, які забезпечують роботу фізичного рушія,

але й ті концептуальні підходи, що формують підґрунтя для побудови цілісного та передбачуваного ігрового простору. У створенні будь-якої 3D-гри, незалежно від її жанру, фізична система відіграє роль прихованого фундаменту, який визначає поведінку як персонажа, так і всіх об’єктів навколо нього. Саме тому глибоке розуміння того, як працює фізика у рушії Unity, дозволяє розробнику досягти значно вищої якості взаємодії та геймплейної логіки, забезпечуючи стабільність ігрових процесів у реальному часі.

Одним із ключових викликів є досягнення балансу між реалістичністю фізики та геймплейною придатністю. У деяких випадках надмірно точна фізична модель може навіть зашкодити ігровому досвіду, створюючи рухи, які хоч і виглядають натурально, але не відповідають очікуванням гравця. У шутерах, наприклад, рух персонажа має бути чітким і контрольованим, навіть якщо в реальності тіло людини рухається значно менш різко. Саме тому фізика у іграх часто модифікується або обмежується через використання спеціальних властивостей RigidBody, як-от заморожування обертання навколо певних осей, примусове обмеження швидкості або ж ручне керування прискоренням. Такі підходи дозволяють зробити поведінку персонажа не тільки фізично правдоподібною, але й комфортною для користувача.

Крім того, важливою складовою фізичних симуляцій є обробка взаємодії між гравцем та об’єктами оточення. У сучасних рушіях, таких як Unity, зіткнення між об’єктами не обмежуються простим виявленням дотику: вони включають глибину проникнення, відновлювальні сили, тертя, коефіцієнт відскоку та багато інших параметрів. Наприклад, моделювання тертя є одним із найбільш складних аспектів, оскільки воно визначає те, як саме гравець рухається поверхнею, наскільки швидко зупиняється після бігу чи ковзає при виконанні стрибка. У деяких іграх розробники свідомо зменшують коефіцієнт тертя, щоб створити більш «ковзний» характер руху, тоді як у платформерах часто, навпаки, збільшують його для більш точного контролю над переміщенням.

Окрему роль відіграють алгоритми стабілізації фізичної сцени. У PhysX передбачено механізм Sleep Mode, який вимикає симуляцію для об’єктів, що певний

час залишаються без руху. Це необхідно не лише для підвищення продуктивності, але й для уникнення дрібних мікровібрацій, які можуть виникати через накопичення чисельних похибок у обчисленнях. У великих сценах, де присутня велика кількість статичних об’єктів, такий підхід дозволяє значно заощадити ресурси та забезпечити стабільність роботи рушія навіть при високій кількості кадрів.

Ще однією важливою сферою є моделювання балістичних процесів та взаємодій, пов’язаних із снарядами та проєкціями. У реалістичних симуляціях використовують фізичні кулі з Rigidbody, які підпадають під вплив гравітації, опору повітря та зіткнень. Однак у більшості ігрових шутерів, особливо з швидкою динамікою бою, розробники уникають цього підходу, замінюючи снаряди миттєвими Raycast-променями. Це пояснюється тим, що реальні снаряди рухаються надто швидко, щоб їхня фізика була помітною для гравця, а використання фізичних тіл ускладнює симуляцію без реальної користі для геймплею. Raycast дозволяє створювати відчуття миттєвого влучання, що значно покращує динаміку ігрового процесу, зокрема в багатокористувацьких проєктах.

Що стосується застосування фізики у мережевих іграх, то тут виникає додаткова складність — необхідність синхронізації станів між клієнтами. У реальному часі передати всі фізичні параметри кожного об’єкта неможливо, оскільки це призведе до гігантської затримки та перенавантаження мережі. Тому розробники використовують гібридні підходи: локальна фізика працює на кожному клієнті окремо, але ключові події, такі як нанесення шкоди, переміщення у просторі, зникнення чи поява об’єктів, синхронізуються через RPC-виклики або PhotonView механізми. Такий підхід дозволяє зберегти стабільність симуляції незалежно від якості інтернет-з’єднання гравців.

Останнім важливим аспектом є питання продуктивності. Фізичні симуляції є одними з найбільш вимогливих до ресурсів компонентів гри. Тому у процесі розробки часто застосовують такі методи оптимізації, як зменшення кількості активних RigidBody, використання простих колайдерів замість складних моделей, запровадження зон активації та деактивації, обмеження кількості фізичних

взаємодій у кадрі та сортування колайдерів за пріоритетом. Завдяки цьому можливо забезпечити високу частоту кадрів навіть у складних сценах, що особливо важливо для 3D-шутерів, де плавність рухів визначає якість геймплею.

У підсумку можна сказати, що фізична симуляція у 3D-середовищах є складною системою, що поєднує розрахункову математику, алгоритми оптимізації, логіку геймплею та можливості рушія Unity. Вона формує реалістичну поведінку об’єктів і створює фундамент, на якому будується ігровий процес. У рамках цієї магістерської роботи фізична система стала одним із центральних елементів реалізації гри, забезпечивши коректну взаємодію персонажа з об’єктами, стабільну роботу механіки стрільби та узгодженість усіх елементів сцени.

2 ПРОЄКТУВАННЯ 3D-ГРИ З ФІЗИЧНОЮ СИМУЛЯЦІЄЮ ОБ'ЄКТІВ

2.1 Постановка задачі та загальні вимоги до системи

Створення тривимірної гри з реалізацією фізичної симуляції об'єктів передбачає попереднє визначення кола вимог, які мають бути виконані в межах програмної системи, а також формулювання загальної задачі, що визначає напрям подальшої розробки. На цьому етапі надзвичайно важливо не лише окреслити кінцеву мету створення ігрового продукту, а й визначити ті властивості, якими він повинен володіти для забезпечення коректної роботи, стабільності, масштабованості та відповідності сучасним технічним стандартам. Вихідним положенням є те, що гра має бути інтерактивною тривимірною системою, у якій ігрові об'єкти здатні взаємодіяти між собою відповідно до законів фізики, а рух гравців і предметів у просторі повинен бути максимально наближеним до природного.

Постановка задачі полягає у створенні прототипу багатокористувацької 3D-гри, де кожен користувач має можливість підключатися до мережевого середовища, взаємодіяти з іншими учасниками, керувати персонажем та впливати на фізичні об'єкти у сцені. Основою розробки є поєднання двох ключових елементів — системи фізичної симуляції та мережевої взаємодії, які повинні працювати синхронно, не створюючи розбіжностей у станах клієнтів. Це означає, що сама природа задачі диктує необхідність попереднього аналізу характеристик фізичного рушія Unity, закономірностей обробки зіткнень, застосування сил і моментів, а також механізмів синхронізації позицій та станів об'єктів на віддалених клієнтах. Першим блоком вимог до системи виступають функціональні умови, які формують поведінку гри з точки зору гравця. Гра повинна забезпечувати можливість входу в мережеве середовище, створення нової ігрової кімнати, приєднання до вже існуючої, очікування інших гравців і запуск самої сцени. Після завантаження ігрового простору система повинна створювати мережеві екземпляри персонажів, кожен з яких відображатиметься відповідно до дій конкретного користувача.

Важливо, щоб рух, стрибки, обертання та інші форми фізичних взаємодій виконувалися коректно незалежно від кількості активних гравців.

Окрема роль належить симуляції фізичних процесів, яка має забезпечити природність руху об’єктів, відповідність їхньої поведінки чинним законам механіки та стабільність під час зіткнень. Ігрові предмети, що розміщені в середовищі, повинні мати відповідні колайдери, масу, матеріали взаємодії та інші параметри, притаманні реальному фізичному тілу. При цьому розробник повинен зважати на баланс між точністю симуляції та продуктивністю, оскільки надмірні фізичні обчислення можуть негативно позначатися на стабільності мережевого процесу. Крім власне фізичних взаємодій, важливою вимогою до системи є наявність елементарної бойової механіки, що передбачає можливість стрільби, завдання шкоди іншим гравцям, перевірки факту попадання та опрацювання наслідків (зокрема зменшення рівня здоров’я або знищення персонажа). У контексті мережевої гри такі взаємодії повинні виконуватися не локально, а через віддалені виклики процедур, які забезпечують достовірну синхронізацію станів між віддаленими клієнтами. Саме ця вимога диктує потребу у виборі відповідного мережевого фреймворку, здатного обробляти подібні взаємодії з мінімальними затримками та коректною фіксацією подій. До важливих елементів постановки задачі належать і ті вимоги, що стосуються користувацького інтерфейсу. Він повинен бути інтуїтивно зрозумілим, забезпечувати простий доступ до функцій створення і підключення до кімнат, показувати поточний стан підключення до сервера, забезпечувати навігацію між вікнами та узгоджувати свою роботу з логікою мережевих подій. Для цього необхідно використати техніку, яка дозволяє опрацьовувати UI окремо від логіки гри, але при цьому забезпечує стабільну інтеграцію з кодом. Не менш суттєвими є нефункціональні вимоги, пов’язані з ефективністю, надійністю та масштабованістю. Гра повинна працювати плавно в умовах змінної якості мережевого підключення та зберігати стабільність навіть за наявності затримок. Важливо, щоб характеристики продуктивності не погіршувалися при збільшенні кількості клієнтів у кімнаті, а дані, що передаються мережею, були мінімізовані та обмежені тільки тими параметрами, які справді

критичні для логіки гри. Крім того, важливим аспектом є можливість подальшого розширення гри, що передбачає модульність архітектури та чіткий розподіл відповідальностей між компонентами.

2.2 Розробка архітектури гри

Архітектура програмного продукту відіграє ключову роль у забезпеченні стабільності, масштабованості та зрозумілості всієї системи. У випадку створення тривимірної гри з мережевою взаємодією, фізичною симуляцією та модульною структурою особливо важливо побудувати архітектуру таким чином, щоб окремі компоненти були максимально ізольованими, але водночас могли легко взаємодіяти між собою у межах визначених інтерфейсів. Початковий етап розробки архітектури передбачає формування модульної структури гри, що дозволить уникнути дублювання логіки та забезпечить можливість подальшого розширення функціоналу.

Архітектура гри базується на трьох ключових рівнях: мережевому рівні, рівні інтерфейсу користувача та рівні геймплею. Такий поділ виникає з потреби чіткого розмежування сфер відповідальності, оскільки мережеві операції не повинні взаємодіяти безпосередньо з ігровою фізикою, а інтерфейс користувача повинен бути здатним змінювати свій стан відповідно до отриманих подій, не втручаючись у логіку руху персонажа або роботу рушія. Кожен рівень містить власні модулі, які виконують певні функції та обмінюються даними через події або API-методи.

Основним елементом мережевої архітектури є модуль, що забезпечує підключення до Photon-сервера, вхід у лобі, створення кімнат та їх оновлення. У цьому модулі розміщені такі ключові компоненти, як PhotonLauncher, який відповідає за встановлення з’єднання, отримання списку кімнат і запуск ігрового сеансу, та GameController, який виконує роль центрального вузла створення гравців під час завантаження сцени. Логіка модулів Photon PUN побудована таким чином, що кожна подія сервера викликає певний callback, що дозволяє архітектурі

залишатися реактивною і не потребує написання складних таймерів або циклів очікування.

Другий важливий шар — інтерфейс користувача. Він реалізований засобами Unity UI Toolkit, що дозволяє відокремити структуру UI від поведінки логічних компонентів гри. У цьому шарі працюють такі модулі, як MainMenu, CreateRoom, JoinRoom, ListOfRooms і RoomMenu. Кожний із них відповідає за частину інтерфейсу і керує переходами між екранами у відповідь на дії гравця або мережеві події. Особливість цієї частини архітектури полягає в тому, що всі елементи інтерфейсу зберігаються як UXML-файли й розміщуються окремо від скриптів, що дозволяє змінювати зовнішній вигляд або структуру UI без зміни логіки програми.

Третім рівнем є модулі, що відповідають за ігровий процес. Найважливішим елементом цього рівня виступає PlayerController, який керує рухом персонажа, обробкою введення, фізичними взаємодіями, а також системою стрільби. Рух персонажа реалізується через Rigidbody, що дозволяє використовувати вбудований фізичний рушій. У цьому ж рівні розміщується PlayerManager, який забезпечує створення екземплярів гравців через PhotonNetwork.Instantiate, і таким чином позбавляє потреби ручного керування появою об’єктів у сцені.

Архітектура побудована таким чином, що кожен модуль знає лише те, що потрібно для його роботи, і не має доступу до внутрішньої логіки інших модулів. Наприклад, модуль стрільби не знає про інтерфейс користувача, а мережевий модуль не займається фізичною симуляцією. Такий поділ дозволяє робити значні зміни у певній частині гри, не зачіпаючи інші модулі, що є однією з ключових властивостей якісної архітектури.

2.3 Створення фізичної моделі об’єктів

Створення фізичної моделі об’єктів у грі є одним із найважливіших етапів проєктування, оскільки саме на цьому рівні формується базова логіка існування елементів віртуального середовища, їхня взаємодія, а також реалістичність

поведінки у тривимірному просторі. У контексті розроблюваної гри фізична модель охоплює кілька ключових елементів: фізику гравця, взаємодію з поверхнями та перешкодами, функціонування снарядів, визначення зіткнень, а також розрахунок нанесення урону. Незважаючи на те, що проєкт не передбачає складної симуляції з використанням кастомних фізичних алгоритмів, важливою була правильна інтеграція вбудованого фізичного рушія Unity таким чином, щоб він працював стабільно у мережевому середовищі, синхронізованому через Photon. Фізична модель починає формуватися з визначення того, яким має бути гравець як об’єкт фізики. Із самого початку було вирішено відмовитися від надто складних систем зі стейт-машинами, капсульними колайдерами різної форми або кількома рігідболді, оскільки це не лише ускладнює реалізацію, але й може призвести до додаткових проблем у мережевому контексті. Замість цього було обрано класичну схему: об’єкт Player складається з трансформу, одного Rigidbody для керування переміщенням та одного CapsuleCollider, що забезпечує коректне зіткнення з геометрією сцени. Така структура дозволяє ефективно використовувати гравітацію, забезпечувати плавний фізичний рух та уникати некоректних взаємодій, які часто виникають через неправильні налаштування компонентів.

Механіка переміщення гравця була реалізована на основі фізичного рушія, що передбачає керування об’єктом не через пряме модифікування transform.position, а через застосування фізичного переміщення за допомогою Rigidbody. У скрипті PlayerController переміщення здійснюється через метод MovePosition, що гарантує фізично коректне оновлення координат. Такий підхід дозволяє уникнути поширених проблем, як-от "телепортування", провалювання крізь поверхні або накопичення енергії при неправильному переміщенні об’єкта. Крім того, метод MovePosition краще синхронізується з мережевою моделлю Photon, оскільки дає чітке розуміння різниці між локальним та віддаленим рухом.

Важливою частиною фізичної моделі стало розмежування локального та віддаленого керування, що запобігає ситуаціям, коли гравець може випадково впливати на рух іншого персонажа. Це було досягнуто завдяки перевірці властивості PhotonView.IsMine: якщо об’єкт не належить локальному

користувачеві, його логіка руху та обробки вводу повністю відключається. Таким чином, лише власник має право змінювати Rigidbody свого персонажа, що є обов'язковою умовою для мережевої коректності фізичної моделі. Наступним елементом фізичної моделі стало створення оточення, яке складається з платформ, стін та статичних кубів. Усі ці об'єкти отримали компонент BoxCollider, що дозволяє їм виступати фізичними бар'єрами. Платформи виконують роль несучих поверхонь, що взаємодіють із гравцем через стандартну систему зіткнень Unity. Статичні куби, своєю чергою, виконують дві функції: вони є перешкодами для навігації гравців та використовуються як об'єкти для тестування механіки стрільби. Розміщення та орієнтація об'єктів виконувались вручну у редакторі, з використанням системи сітки та інструментів вирівнювання.

Ще одним елементом фізичної моделі є реалізація стрільби. У проєкті було вирішено не створювати фізичні снаряди, оскільки це суттєво збільшує навантаження на мережевий рушій і ускладнює синхронізацію. Натомість використовується Raycast — невидимий промінь, що виходить із камери гравця та перевіряє зіткнення з об'єктами попереду. Якщо промінь досягає іншого гравця, викликається метод нанесення урону, який надсилається через RPC, забезпечуючи, що всі клієнти отримають ідентичний результат. Такий підхід є стандартним для шутерів початкового та середнього рівня складності, оскільки він не лише зменшує навантаження на мережу, але й гарантує стабільність. Продовжуючи формування фізичної моделі об'єктів, варто звернути увагу на те, що фізика у тривимірних іграх не може бути ізольованою лише до окремих компонентів, таких як гравець або перешкоди. Для повноцінного функціонування системи необхідна коректна взаємодія всіх елементів середовища між собою, включаючи поведінку камер, логіку зміни станів під час руху, реакції на зіткнення та механізми визначення нанесення урону. У цьому контексті значну роль відіграє налаштування Physics Settings у Unity, де визначаються глобальні параметри симуляції, такі як частота оновлення фізики (Fixed Timestep), значення гравітації та поведінка елементів під час зіткнень. У рамках створення гри важливо було налаштувати стабільну частоту

оновлення фізичних подій, оскільки від цього залежить плавність руху гравця та точність обробки взаємодій з поверхнями.

Показовим є рішення щодо використання окремих налаштувань для камер, оскільки камера у шутері від першої особи є важливим фізичним елементом, який має точно відображати стан гравця, але при цьому не підпорядковуватися законам фізики. Камера прив’язується до верхньої частини капсульного колайдера гравця і обертається разом із персонажем. Проте принципово важливо, щоб камера не мала власного Rigidbody, оскільки це призвело б до конфліктів між фізичною та візуальною моделями. Тому камера була інтегрована виключно як дочірній об’єкт Player, логіка якого контролюється через обробку вводу у скрипті PlayerController. У контексті фізичної моделі ключове значення має система зіткнень, яка визначає, як об’єкти реагуватимуть один на одного у просторі сцени. У Unity кожен колайдер може працювати або як тригер, або як твердотільний фізичний бар’єр. Для гравця та перешкод було використано саме другий варіант, оскільки він дозволяє забезпечити чітке фізичне зіткнення без проходження об'єктів один крізь одного. Розрахунок зіткнень виконується на рівні фізичного рушія, а програмна логіка використовується лише для реакції на ці події. Такий підхід дає змогу зменшити складність коду і повною мірою використовувати потенціал PhysX — фізичного механізму, що лежить в основі Unity.

Особливо важливо було налаштувати взаємодію гравця з поверхнями, оскільки неправильні налаштування фізичних матеріалів можуть призвести до небажаних ефектів, таких як ковзання або некоректний стрибок. У цьому проєкті було застосовано стандартні фізичні матеріали зі зменшеним коефіцієнтом ковзання, що дозволило стабілізувати рух персонажа і запобігти його неконтрольованому прискоренню при русі під нахилом. Для перешкод використовуються матеріали без відскоку, що відповідає жанру шутера, де фізика має бути стриманою, прогнозованою та не відволікаючою.

У межах фізичної моделі окрему увагу було приділено механізму нанесення урону. Попри те, що він не є частиною класичної фізики, саме цей механізм визначає результат взаємодії гравців у просторі. Реалізація урону відбувається через

Raycast, який взаємодіє з колайдером іншого гравця. Як тільки промінь торкається цільового колайдера, відбувається виклик RPC-методу Damage, який обробляє зменшення здоров’я на всіх клієнтах. У відповідь на зменшення здоров’я виконується логіка знищення гравця через PhotonNetwork.Destroy, що коректно “прибирає” відповідний об’єкт зі сцени на всіх машинах. Така система забезпечує точність визначення урону без складної фізики снарядів.

2.4 Побудова мережевої інфраструктури гри та взаємодії з Photon

1) CreateRoom.cs

Представлений програмний модуль CreateRoom.cs є складовою частиною клієнтської підсистеми гри та відповідає за ініціалізацію ігрових сесій у мережевому середовищі. На початку файлу декларуються необхідні простори імен, які забезпечують доступ до базового функціоналу рушія Unity, інструментарію нового інтерфейсу користувача UI Toolkit та мережевої бібліотеки Photon Unity Networking (PUN). Клас успадковується від базового класу MonoBehaviour, що дозволяє інтегрувати його в життєвий цикл ігрових об'єктів сцени. У тілі класу оголошуються приватні змінні для зберігання посилань на елементи інтерфейсу (кнопку та поле введення), а також публічна змінна loadingUI, яка надає доступ до об'єкта екрана завантаження через редактор Unity. Така структура забезпечує інкапсуляцію логіки відображення та чітке розділення відповідальності між компонентами.

Лістинг коду:

```
 using Photon.Pun;
using UnityEngine;
using UnityEngine.UIElements;

public class CreateRoom : MonoBehaviour
{
    private Button _createRoomBtn;
    private TextField _userRoomName;
    public GameObject loadingUI;
```

Ініціалізація компонентів відбувається в межах методу Start, який є стандартним методом життєвого циклу Unity і викликається одноразово при завантаженні скрипту. У цьому блоці коду здійснюється доступ до кореневого елемента візуального дерева інтерфейсу через компонент UIDocument. За допомогою методу запиту Q<Type>, який виконує пошук елементів за їхніми унікальними рядковими ідентифікаторами, встановлюється зв'язок між змінними скрипту та відповідними віджетами UXML-розмітки: кнопкою підтвердження та полем для введення назви кімнати. Важливим етапом є реєстрація обробника події натискання кнопки, що реалізується шляхом додавання методу CreateRoomBtnOnClicked до делегата події clicked кнопки _createRoomBtn. Це забезпечує реакцію системи на дії користувача.

Лістинг коду:

```
 private void Start()
   {
       var root = GetComponent<UIDocument>().rootVisualElement;
       _createRoomBtn = root.Q<Button>("CreateRoomBtn");
       _createRoomBtn.clicked += CreateRoomBtnOnClicked;

       _userRoomName = root.Q<TextField>("UserRoomName");
   }
```

Основна бізнес-логіка створення кімнати зосереджена у приватному методі CreateRoomBtnOnClicked. Алгоритм розпочинається зі зчитування рядкового значення, введеного користувачем у поле _userRoomName. Для забезпечення цілісності даних та уникнення помилок на стороні сервера застосовується механізм валідації: статичний метод string.IsNullOrEmpty перевіряє вхідний рядок, і у

випадку його порожнечі виконання методу переривається. Якщо валідація пройшла успішно, викликається метод PhotonNetwork.CreateRoom, який надсилає запит до майстер-сервера Photon для створення нової ігрової сесії з вказаним іменем. Після відправлення мережевого запиту відбувається зміна стану графічного інтерфейсу: поточний об'єкт меню деактивується, а об'єкт loadingUI стає активним. Це забезпечує візуальний зворотний зв'язок, повідомляючи користувача про процес з'єднання та запобігаючи повторним натисканням кнопки під час очікування відповіді від сервера.

Лістинг коду:

```
private void CreateRoomBtnOnClicked()
    {
        string userInput = _userRoomName.value;
        if (string.IsNullOrEmpty(userInput)) return;

        PhotonNetwork.CreateRoom(userInput);

        gameObject.SetActive(false);
        loadingUI.SetActive(true);
    }
```

2) ListOfRooms

Програмний компонент ListOfRooms.cs відповідає за візуалізацію доступних ігрових сесій (кімнат) та забезпечує інтерфейс для приєднання користувача до обраної гри. Клас успадковується від спеціалізованого класу MonoBehaviourPunCallbacks, що надає розширені можливості для взаємодії з системою зворотних викликів мережевого рушія Photon, хоча в даному контексті він насамперед використовується як компонент Unity. У блоці оголошень визначаються залежності від стандартних колекцій .NET, бібліотек Unity для роботи з UI Toolkit, а також просторів імен Photon.Pun та Photon.Realtime для реалізації мережевої взаємодії. Поля класу включають публічний об'єкт loadingUI, який дозволяє керувати відображенням екрана завантаження, та приватну змінну _listOfRooms типу VisualElement, що слугує контейнером для динамічно згенерованого списку кнопок у графічному інтерфейсі.

Лістинг коду:

```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;
using Photon.Pun;
using Photon.Realtime;

public class ListOfRooms : MonoBehaviourPunCallbacks
{
    public GameObject loadingUI;
    private VisualElement _listOfRooms;
```

Ініціалізація роботи компонента відбувається в методі Start, де здійснюється прив'язка логіки скрипту до візуального дерева UXML. Отримання посилання на кореневий елемент через GetComponent<UIDocument>().rootVisualElement дозволяє знайти контейнер із назвою "ListOfRooms", у якому згодом будуть розміщуватися інтерактивні елементи списку. Після успішної ініціалізації змінної _listOfRooms викликається публічний метод ChangeRooms, який відповідає за первинне наповнення списку доступних кімнат на основі актуальних даних.

Лістинг коду:

```
private void Start()
    {
        var root = GetComponent<UIDocument>().rootVisualElement;
        _listOfRooms = root.Q<VisualElement>("ListOfRooms");
        ChangeRooms();
    }
```

Метод ChangeRooms реалізує алгоритм динамічної генерації інтерфейсу користувача. На початку виконується перевірка на існування контейнера та його очищення методом Clear(), що запобігає дублюванню елементів при оновленні списку. Далі застосовується ітераційна конструкція foreach для перебору колекції назв кімнат, яка зберігається у зовнішньому статичному масиві PhotonLauncher.RoomList. Для кожного елемента цієї колекції програмно створюється новий екземпляр класу Button. Властивості text цієї кнопки присвоюється назва кімнати, а для події clicked за допомогою лямбда-виразу () => JoinRoom(el) реєструється анонімний метод, який викликає функцію приєднання з

передачею конкретного імені кімнати як аргументу. Сформований об'єкт кнопки додається до візуального контейнера _listOfRooms, завершуючи побудову списку.

Лістинг коду:

```
public void ChangeRooms()
    {
        if(_listOfRooms == null) return;

        _listOfRooms.Clear();
        foreach (string el in PhotonLauncher.RoomList)
        {
            Button button = new Button();
            button.text = el;
            button.clicked += () => JoinRoom(el);
            _listOfRooms.Add(button);
        }
    }
```

Завершує логіку модуля метод JoinRoom, який приймає рядок з назвою кімнати як вхідний параметр. Цей метод звертається до статичного API PhotonNetwork.JoinRoom, ініціюючи процес підключення клієнта до обраної ігрової сесії на сервері. Одночасно з мережевим запитом відбувається перемикання станів графічного інтерфейсу: активується об'єкт завантаження loadingUI, а поточний об'єкт меню деактивується, що забезпечує коректний користувацький досвід під час переходу між сценами або станами гри.

Лістинг коду:

```
private void JoinRoom(string name)
    {
        PhotonNetwork.JoinRoom(name);
        loadingUI.SetActive(true);
        gameObject.SetActive(false);
    }
}
```

3) MainMenu.cs

Програмний клас MainMenu.cs відіграє роль центрального контролера навігаційної системи початкового інтерфейсу користувача. Його першочерговим завданням є забезпечення логічного розгалуження сценарію взаємодії користувача з програмою, надаючи вибір між створенням нової ігрової сесії та приєднанням до вже існуючої. З точки зору архітектури Unity, даний скрипт реалізує патерн контролера

представлення, пов'язуючи візуальну розмітку, створену засобами UI Toolkit, з ігровими об'єктами сцени, що відповідають за подальші етапи конфігурації мережевого з'єднання. На початку файлу здійснюється імпорт базових просторів імен: UnityEngine для доступу до ядра рушія та UnityEngine.UIElements для роботи з новітньою системою побудови інтерфейсів на основі візуального дерева. Клас успадковує функціональність MonoBehaviour, що є обов'язковим для компонентів, які прикріплюються до об'єктів на сцені. У блоці оголошення змінних визначаються приватні поля _createRoomBtn та _joinRoomBtn типу Button, які слугують програмними посиланнями на інтерактивні елементи інтерфейсу. Окрім цього, оголошуються публічні поля createRoom та joinRoom типу GameObject. Використання публічного модифікатора доступу в даному контексті є архітектурним рішенням, що дозволяє розробнику через інспектор Unity (Inspector) явно вказати посилання на кореневі об'єкти інших меню (панелей створення та пошуку кімнат), забезпечуючи таким чином гнучкість налаштування переходів без необхідності жорсткого кодування зв'язків.

Лістинг коду:

```
using UnityEngine;
using UnityEngine.UIElements;
public class MainMenu : MonoBehaviour
{

    private Button _createRoomBtn, _joinRoomBtn;
    public GameObject createRoom, joinRoom;
```

Етап ініціалізації та налаштування внутрішнього стану компонента реалізовано в методі Start. Цей метод виконує критично важливу функцію зв'язування логіки C# з UXML-розміткою. Спершу відбувається отримання доступу до кореневого елемента ієрархії інтерфейсу (rootVisualElement) через компонент UIDocument, який виступає містком між даними розмітки та кодом. Наступним кроком виконується пошук конкретних кнопок у візуальному дереві за допомогою методу запиту Q<Button>, який ідентифікує елементи за їхніми унікальними рядковими іменами ("CreateRoomBtn" та "JoinRoomBtn"). Після отримання посилань на кнопки застосовується подієво-орієнтований підхід: до події clicked

кожної кнопки додаються відповідні методи-обробники (CreateRoomBtnOnClicked та JoinRoomBtnOnClicked). Такий механізм підписки дозволяє асинхронно реагувати на дії користувача, відокремлюючи логіку відтворення інтерфейсу від логіки обробки вводу, що відповідає принципам слабкої зв'язаності компонентів (Loose Coupling).

Лістинг коду:

```
private void Start()
    {
        var root = GetComponent<UIDocument>().rootVisualElement;
        _createRoomBtn = root.Q<Button>("CreateRoomBtn");
        _createRoomBtn.clicked += CreateRoomBtnOnClicked;

        _joinRoomBtn = root.Q<Button>("JoinRoomBtn");
        _joinRoomBtn.clicked += JoinRoomBtnOnClicked;
    }
```

Реалізація навігаційної логіки зосереджена у методах зворотного виклику (callbacks). Метод JoinRoomBtnOnClicked відповідає за перехід до меню вибору кімнат. Алгоритм переходу базується на маніпуляції станом активності ігрових об'єктів: поточний об'єкт меню (до якого прикріплено цей скрипт) деактивується викликом gameObject.SetActive(false), що призводить до його зникнення з екрана та призупинення виконання пов'язаних процесів відображення. Одночасно з цим об'єкт joinRoom, посилання на який було встановлено в інспекторі, переводиться в активний стан (SetActive(true)). Аналогічна логіка реалізована в методі CreateRoomBtnOnClicked, який активує панель створення кімнати createRoom. Такий підхід до організації інтерфейсу фактично реалізує найпростішу модель скінченного автомата (Finite State Machine), де станами є різні екрани меню, а переходи ініціюються діями користувача. Це забезпечує чіткий та контрольований потік навігації, гарантуючи, що в будь-який момент часу користувач взаємодіє лише з одним активним контекстом меню, що є важливим для коректного UX у складних системах симуляції.

Лістинг коду:

```
private void JoinRoomBtnOnClicked()
    {
        gameObject.SetActive(false);
        joinRoom.SetActive(true);
    }

    private void CreateRoomBtnOnClicked()
    {
        gameObject.SetActive(false);
        createRoom.SetActive(true);
    }
}
```

4) RoomMenu

Програмний компонент RoomMenu.cs виконує роль адміністративного контролера віртуальної кімнати очікування (lobby), де користувачі перебувають після успішного створення або приєднання до мережевої сесії, але перед безпосереднім початком ігрового процесу. Клас розроблено з використанням бібліотек UnityEngine та UnityEngine.UIElements для маніпуляцій з графічним інтерфейсом, а також просторів імен Photon.Pun та Photon.Realtime для взаємодії з хмарною інфраструктурою Photon. У структурі класу визначено приватне поле _startGameBtn типу Button, яке забезпечує програмний доступ до елемента управління, відповідального за ініціалізацію ігрового рівня. Цей підхід дозволяє інкапсулювати логіку управління станом гри в межах одного компонента, що відповідає принципам об'єктно-орієнтованого програмування.

Лістинг коду:

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;
using Photon.Pun;
using Photon.Realtime;

public class RoomMenu : MonoBehaviour
{

    private Button _startGameBtn;
```

Процедура ініціалізації інтерфейсу реалізована в методі Start, який автоматично викликається середовищем Unity при завантаженні об'єкта. На першому етапі виконання методу відбувається отримання доступу до кореневого елемента візуального дерева UXML через компонент UIDocument. За допомогою методу Q<Label> скрипт знаходить текстовий віджет із ідентифікатором "roomName" і динамічно змінює його зміст, відображаючи поточну назву кімнати, отриману через властивість PhotonNetwork.CurrentRoom.Name. Це надає користувачеві візуальне підтвердження успішного входу до конкретної ігрової сесії та дозволяє ідентифікувати мережеве оточення.

Лістинг коду:

```
private void Start()
   {
      var root = GetComponent<UIDocument>().rootVisualElement;
      Label roomName = root.Q<Label>("roomName");
      roomName.text = "Комната создана: " + PhotonNetwork.CurrentRoom.Name;
```

Наступний етап ініціалізації присвячений налаштуванню елементів управління та реалізації рольової моделі доступу. Скрипт отримує посилання на кнопку запуску гри _startGameBtn та підписує метод StartGameBtnOnClicked на подію її натискання. Критично важливою частиною логіки є перевірка статусу клієнта в мережі за допомогою властивості PhotonNetwork.IsMasterClient. Ця умова визначає, чи є поточний користувач адміністратором кімнати (Master Client), тобто тим, хто її створив або отримав права хоста. Якщо користувач не володіє правами адміністратора, кнопка запуску гри примусово деактивується методом SetEnabled(false). Такий механізм запобігає розсинхронізації ігрового процесу, делегуючи право початку симуляції виключно одному користувачеві, що гарантує цілісність сесії для всіх учасників.

Лістинг коду:

```
_startGameBtn = root.Q<Button>("StartGameBtn");
      _startGameBtn.clicked += StartGameBtnOnClicked;

      if(!PhotonNetwork.IsMasterClient)
         _startGameBtn.SetEnabled(false);
   }
```

Метод StartGameBtnOnClicked, який виконується при взаємодії адміністратора кімнати з інтерфейсом, містить команду для переходу до основної сцени симуляції. Виклик методу PhotonNetwork.LoadLevel(1) замість стандартного SceneManager.LoadScene є специфічною особливістю роботи з мережевим рушієм Photon. Ця функція не лише завантажує сцену з індексом 1 (яка містить безпосередньо ігрове середовище та фізичну симуляцію) на локальному клієнті, але й, за умови налаштування PhotonNetwork.AutomaticallySyncScene, автоматично синхронізує цей перехід для всіх підключених клієнтів у кімнаті. Це забезпечує одночасний початок ігрової сесії для всіх учасників, що є необхідною умовою для коректної багатокористувацької взаємодії в реальному часі.

Лістинг коду:

```
private void StartGameBtnOnClicked()
   {
      PhotonNetwork.LoadLevel(1);
   }
}
```

5) Cube.cs

Представлений програмний модуль ClickPush є ключовим елементом підсистеми фізичної симуляції, що забезпечує інтерактивну взаємодію користувача з об'єктами ігрового світу. Клас розроблено на базі стандартної бібліотеки UnityEngine, що дозволяє використовувати вбудовані механізми фізичного рушія (NVIDIA PhysX). В архітектурному плані клас є спадкоємцем MonoBehaviour, що дозволяє прикріплювати його до будь-якого тривимірного об'єкта сцени. У блоці оголошення змінних визначається публічне поле force типу float, яке задає скалярну величину сили, що буде прикладена до об'єкта. Публічний модифікатор доступу дозволяє налаштовувати цей параметр безпосередньо через редактор Unity (Inspector), забезпечуючи гнучкість геймдизайну. Також оголошується приватне поле rb типу Rigidbody. Цей компонент є фундаментальним для фізичної симуляції в Unity, оскільки він надає об'єкту масу та підпорядковує його законам ньютонівської механіки.

Лістинг коду:

```
using UnityEngine;

public class ClickPush : MonoBehaviour
{
    public float force = 10f;
    private Rigidbody rb;
```

Етап ініціалізації компонента відбувається в методі Start. Основною операцією тут є кешування посилання на компонент Rigidbody, прикріплений до того ж ігрового об'єкта. Виклик узагальненого методу GetComponent<Rigidbody>() дозволяє отримати доступ до фізичних властивостей об'єкта та маніпулювати ними в подальшому коді. Збереження цього посилання у змінну rb є важливою оптимізацією, що дозволяє уникнути ресурсомістких операцій пошуку компонентів під час активної фази ігрового процесу.

Лістинг коду:

```
void Start()
    {
        rb = GetComponent<Rigidbody>();
    }
```

Логіка безпосередньої взаємодії реалізована в методі обробки подій OnMouseDown. Цей метод автоматично викликається рушієм Unity, коли користувач натискає кнопку миші, навівши курсор на колайдер об'єкта. Алгоритм розрахунку фізичного впливу базується на векторній алгебрі. Спершу обчислюється вектор напрямку dir як різниця між позицією об'єкта (transform.position) та позицією камери (Camera.main.transform.position). Отриманий вектор визначає лінію дії сили від спостерігача до об'єкта. Після цього виконується нормалізація вектора методом Normalize(), що приводить його довжину до одиниці, зберігаючи при цьому інформацію лише про напрямок.

Лістинг коду:

```
void OnMouseDown()
    {
        Vector3 dir = transform.position - Camera.main.transform.position;
        dir.Normalize();
```

Завершальним етапом алгоритму є безпосереднє застосування сили до фізичного тіла. Метод rb.AddForce прикладає силу до об'єкта вздовж розрахованого вектора dir, помноженого на коефіцієнт інтенсивності force. Ключовим аспектом реалізації є використання режиму ForceMode.Impulse. Цей параметр вказує фізичному рушію, що сила має бути прикладена миттєво, враховуючи масу об'єкта, що фізично відповідає ударній взаємодії або поштовху. Це призводить до миттєвої зміни вектора швидкості об'єкта, створюючи реалістичну симуляцію кінетичного впливу.

Лістинг коду:

```
rb.AddForce(dir * force, ForceMode.Impulse);
    }
}
```

6) PlayerController.cs

Програмний клас PlayerController.cs є центральним керуючим модулем сутності гравця в мережевому середовищі. Він відповідає за обробку введення користувача, керування переміщенням персонажа з урахуванням фізики твердого тіла, реалізацію бойової системи (стрільби) та синхронізацію стану об'єкта між клієнтами через мережу Photon Unity Networking (PUN). Клас розроблено з урахуванням принципів кросплатформності, оскільки він містить директиви препроцесора для підтримки як старої системи введення, так і нової UnityEngine.InputSystem. У блоці оголошення змінних визначаються ключові параметри персонажа, такі як швидкість переміщення (speed), швидкість обертання (rotateSpeed), величина завданої шкоди (_attackDamage) та поточний рівень здоров'я (_health). Також зберігаються посилання на компоненти Rigidbody для фізичних розрахунків та PhotonView для ідентифікації мережевого власника об'єкта.

Лістинг коду:

```
public class PlayerController : MonoBehaviour
{
    private Rigidbody _rb;
    private PhotonView _photonView;
    public float speed = 5f, rotateSpeed = 4f;
    public int _attackDamage = 25;
    private int _health = 100;
```

Критично важливим етапом життєвого циклу об'єкта є ініціалізація в методі Awake. Окрім отримання посилань на необхідні компоненти, тут реалізовано логіку розділення локального та віддаленого управління. Перевірка !_photonView.IsMine дозволяє визначити, чи належить даний екземпляр гравця локальному клієнту. Якщо об'єкт є репрезентацією віддаленого гравця, камера, прикріплена до нього, знищується. Це запобігає конфліктам рендерингу та гарантує, що кожен гравець бачить ігровий світ лише зі своєї перспективи, не перехоплюючи управління камерами інших учасників сесії.

Лістинг коду:

```
private void Awake()
    {
        _photonView = GetComponent<PhotonView>();
        _rb = GetComponent<Rigidbody>();

        if(!_photonView.IsMine)
            Destroy(GetComponentInChildren<Camera>().gameObject);
    }
```

Архітектура управління персонажем розділена на два потоки оновлення: FixedUpdate для фізичних розрахунків та Update для обробки введення та ігрової логіки. В обох методах застосовується патерн раннього виходу (Guard Clause) через перевірку _photonView.IsMine, що забороняє виконання керуючого коду на екземплярах об'єктів, які не належать локальному клієнту. Метод RotatePlayer відповідає за орієнтацію персонажа у просторі. Залежно від активної системи введення, зчитуються дані з геймпада або клавіатури, після чого застосовується трансформація обертання навколо вертикальної осі Vector3.up. Метод MovePlayer використовує фізичний рушій для переміщення: замість прямої зміни координат використовується метод _rb.MovePosition, який обчислює нову позицію з

урахуванням поточної позиції, вектора напрямку transform.forward та дельти часу. Такий підхід забезпечує плавність руху та коректну обробку колізій.

Лістинг коду:

```
private void FixedUpdate()
    {
        if (!_photonView.IsMine) return;
        MovePlayer();
    }

    private void RotatePlayer()
    {
        // ... (логіка зчитування вводу)
        transform.Rotate(Vector3.up * rotateSpeed * h);
    }

    private void MovePlayer()
    {
          // ... (логіка зчитування вводу)
        _rb.MovePosition(transform.position + (transform.forward * Time.fixedDeltaTime
* speed * v));
    }
```

Бойова система реалізована методом Shoot, який базується на технології трасування променів (Raycasting). При натисканні кнопки атаки (ліва кнопка миші або відповідна кнопка геймпада) будується промінь від камери в напрямку курсора. Якщо промінь перетинає колайдер об'єкта з тегом "Player", викликається метод нанесення шкоди. Особливістю реалізації є використання механізму віддалених викликів процедур (RPC). Метод Damage не змінює здоров'я напряму, а надсилає мережеву команду PunDamage всім клієнтам через _photonView.RPC. Метод PunDamage, позначений атрибутом [PunRPC], виконується на всіх копіях об'єкта, але фактична зміна здоров'я та знищення об'єкта (у разі смерті) обмежується перевіркою IsMine, що делегує відповідальність за стан персонажа його власнику, запобігаючи розсинхронізації даних.

Лістинг коду:

```
private void Shoot()
    {
        // ... (логіка Raycast)
            if (Physics.Raycast(ray, out RaycastHit hit))
            {
                if (hit.collider.CompareTag("Player"))

hit.collider.GetComponent<PlayerController>().Damage(_attackDamage);
            }
    }

    public void Damage(int damage)
    {
        _photonView.RPC("PunDamage", RpcTarget.All, damage);
    }

    [PunRPC]
    void PunDamage(int damage)
    {
        if (!_photonView.IsMine) return;
        _health -= damage;
        if(_health <= 0)
            PhotonNetwork.Destroy(gameObject);
    }
```

7) PlayerManager.cs

Програмний клас PlayerManager.cs виконує роль точки входу для ініціалізації ігрового аватара користувача в межах мережевої сцени. Цей компонент є критично важливим для забезпечення коректної архітектури клієнт-серверної взаємодії, оскільки він розмежовує логіку загального керування сценою та персональну логіку створення керованого об'єкта для кожного окремого клієнта. Клас побудований на базі бібліотек UnityEngine та Photon.Pun, що дозволяє інтегрувати стандартні механізми Unity з мережевим функціоналом Photon. У структурі класу оголошено приватне поле _photonView, яке слугує посиланням на компонент мережевої ідентифікації. Цей компонент є необхідним для розрізнення локальних та віддалених об'єктів у спільному віртуальному просторі, забезпечуючи адресність мережевих повідомлень.

Лістинг коду:

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Photon.Pun;

public class PlayerManager : MonoBehaviour
{
    private PhotonView _photonView;
```

Алгоритм роботи менеджера реалізовано в методі Start, який активується при завантаженні об'єкта на сцені. Першочерговим завданням методу є ініціалізація змінної _photonView шляхом отримання відповідного компонента, прикріпленого до поточного ігрового об'єкта. Ключовим етапом логіки є умовна конструкція if(_photonView.IsMine), яка перевіряє приналежність даного екземпляра скрипту локальному клієнту. Ця перевірка є фундаментальною для запобігання дублюванню логіки створення персонажа: метод створення гравця викликається виключно в тому випадку, якщо потік виконання належить клієнту, який володіє цим мережевим об'єктом. Такий підхід гарантує, що кожен підключений користувач ініціює створення лише власного персонажа, не втручаючись у процеси інших учасників сесії.

Лістинг коду:

```
private void Start()
    {
        _photonView = GetComponent<PhotonView>();
        if(_photonView.IsMine)
            CreatePlayer();
    }
```

Безпосередня генерація ігрового об'єкта відбувається у приватному методі CreatePlayer. Замість стандартного методу Instantiate рушія Unity, тут використовується спеціалізований метод PhotonNetwork.Instantiate. Ця функція приймає назву префабу ("Player") як рядковий аргумент, а також початкові координати (Vector3.zero) та орієнтацію (Quaternion.identity). Використання PhotonNetwork.Instantiate є обов'язковою умовою для мережевих ігор, оскільки цей

метод не лише створює об'єкт на локальній сцені, але й автоматично надсилає буферизовану мережеву подію всім іншим клієнтам, змушуючи їх створити відповідну копію цього об'єкта у своїх версіях світу. Крім того, створеному об'єкту автоматично присвоюється унікальний мережевий ідентифікатор (ViewID), що дозволяє синхронізувати його фізичний стан (позицію, обертання) та ігрові параметри між усіма учасниками сесії в реальному часі.

Лістинг коду:

```
private void CreatePlayer()
   {
      PhotonNetwork.Instantiate("Player", Vector3.zero, Quaternion.identity);
   }
}
```

8) Schoot.cs

Програмний модуль Schoot.cs, що містить клас Shootable, реалізує механіку дистанційної фізичної взаємодії користувача з об'єктами ігрового світу за допомогою технології трасування променів (Raycasting). Даний компонент є універсальним інструментом для здійснення силового впливу на динамічні тіла і розроблений з урахуванням сумісності з різними версіями підсистем введення рушія Unity. Клас успадковує функціональність базового класу MonoBehaviour, що дозволяє інтегрувати його в життєвий цикл сцени. У блоці оголошення змінних визначається публічне поле force типу float. Використання публічного модифікатора доступу надає можливість геймдизайнеру налаштовувати інтенсивність фізичного імпульсу безпосередньо через інтерфейс редактора Unity, не втручаючись у програмний код, що забезпечує гнучкість налаштування ігрового балансу.

Лістинг коду:

```
using UnityEngine;
#if ENABLE_INPUT_SYSTEM
using UnityEngine.InputSystem;
#endif

public class Shootable : MonoBehaviour
{
    public float force = 500f;

    void Update()
    {
```

Основна логіка обробки взаємодії зосереджена в методі Update, який виконується кожного кадру. Архітектурною особливістю даного методу є використання директив умовної компіляції #if ENABLE_INPUT_SYSTEM, що дозволяє автоматично адаптувати код під активну систему введення проєкту (Legacy Input Manager або нову Input System). У блоці, призначеному для нової системи введення, перевірка натискання лівої кнопки миші здійснюється через клас Mouse.current. Стан кнопки зчитується за допомогою властивості wasPressedThisFrame, а координати курсора на екрані отримуються через Mouse.current.position.ReadValue(). Це забезпечує коректну обробку подій введення на сучасних платформах, що підтримують подієво-орієнтовану модель Input System.

Лістинг коду:

```
#if ENABLE_INPUT_SYSTEM
        bool click = Mouse.current != null ?
Mouse.current.leftButton.wasPressedThisFrame : false;
        if (click)
        {
            Vector2 mousePos = Mouse.current != null ?
Mouse.current.position.ReadValue() : Vector2.zero;
            Ray ray = Camera.main.ScreenPointToRay(mousePos);

            if (Physics.Raycast(ray, out RaycastHit hit))
            {
```

Альтернативна гілка коду (блок #else) забезпечує підтримку класичного менеджера введення, перевіряючи натискання через статичний метод

Input.GetMouseButtonDown(0). Незалежно від обраної системи введення, ключовим алгоритмом є генерація фізичного променя. Метод Camera.main.ScreenPointToRay перетворює двовимірні координати курсора на екрані у тривимірний промінь (Ray), що виходить з позиції камери в глибину сцени. Наступним кроком виконується метод Physics.Raycast, який перевіряє перетин цього променя з будь-яким коллайдером на сцені. Результат перевірки зберігається у структурі RaycastHit, яка містить детальну інформацію про точку зіткнення та об'єкт, з яким воно відбулося.

Лістинг коду:

```
#else
        if (Input.GetMouseButtonDown(0))
        {
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

            if (Physics.Raycast(ray, out RaycastHit hit))
            {
                Rigidbody rb = hit.collider.GetComponent<Rigidbody>();

                if (rb != null)
                {
```

Після успішного виявлення перетину, алгоритм намагається отримати компонент Rigidbody об'єкта, з яким відбулося зіткнення. Наявність цього компонента є необхідною умовою для застосування фізичних сил. Якщо об'єкт є фізичним тілом (посилання rb не є null), відбувається розрахунок вектора прикладання сили. Вектор напрямку direction обчислюється як різниця між точкою влучання променя (hit.point) та позицією камери. Отриманний вектор нормалізується для приведення його довжини до одиниці, що дозволяє коректно масштабувати силу. Фінальною операцією є виклик методу rb.AddForce, який прикладає імпульс сили вздовж розрахованого вектора. Використання режиму ForceMode.Impulse забезпечує миттєву зміну імпульсу тіла з урахуванням його маси, імітуючи кінетичний удар, наприклад, від пострілу або різкого поштовху.

Лістинг коду:

```
Vector3 direction = hit.point - Camera.main.transform.position;
                direction.Normalize();

                rb.AddForce(direction * force, ForceMode.Impulse);
            }
        }
    }
#endif
    }
}
```

9) GameController

Програмний модуль GameController.cs виконує роль глобального керуючого елемента архітектури додатку, забезпечуючи цілісність ігрового процесу при переході між різними сценами. Клас розроблено з використанням бібліотек UnityEngine та UnityEngine.SceneManagement для керування сценами, а також Photon.Pun для інтеграції з мережевим середовищем. Він успадковується від базового класу MonoBehaviourPunCallbacks, що надає доступ до системи подій мережевого рушія Photon. Ключовою архітектурною особливістю даного компонента є реалізація породжувального патерну проєктування «Одинак» (Singleton). У тілі класу оголошено приватну статичну змінну _game, яка зберігає посилання на єдиний екземпляр контролера. Метод Start містить логіку забезпечення унікальності об'єкта: при ініціалізації відбувається перевірка на наявність вже існуючого екземпляра. Якщо змінна _game вже ініціалізована, поточний дублікат об'єкта знищується методом Destroy(gameObject). У протилежному випадку, поточний екземпляр позначається як основний, і до нього застосовується метод DontDestroyOnLoad(this), який запобігає знищенню об'єкта при перезавантаженні сцен, дозволяючи зберігати глобальний стан гри протягом усього сеансу роботи програми.

Лістинг коду:

```
using System;
using System.Collections;
using System.Collections.Generic;
using Photon.Pun;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameController : MonoBehaviourPunCallbacks
{
    private static GameController _game;
    private void Start()
    {
        if(_game != null)
            Destroy(gameObject);

        DontDestroyOnLoad(this);
        _game = this;
    }
```

Для організації реакції системи на зміни ігрових рівнів використовуються методи життєвого циклу OnEnable та OnDisable. У методі OnEnable відбувається підписка на подію sceneLoaded статичного класу SceneManager шляхом делегування обробки методу SceneManagerOnSceneLoaded. Це гарантує, що кожного разу при завершенні завантаження нової сцени буде виконано відповідний код ініціалізації. Відповідно, у методі OnDisable реалізовано відписку від цієї події. Такий підхід є стандартом безпечного програмування в Unity, оскільки він запобігає витокам пам'яті та помилкам виконання, пов'язаним зі зверненням до знищених об'єктів або делегатів, коли контролер вимикається або видаляється з пам'яті.

Лістинг коду:

```
public override void OnEnable()
    {
        base.OnDisable();
        SceneManager.sceneLoaded -= SceneManagerOnSceneLoaded;
    }
```

```
    public override void OnDisable()
    {
```

Продовження лістингу коду:

Безпосередня бізнес-логіка ініціалізації ігрових сутностей зосереджена у приватному методі SceneManagerOnSceneLoaded. Цей метод приймає параметри завантаженої сцени та режиму завантаження, що дозволяє фільтрувати дії залежно від контексту. Алгоритм перевіряє властивість buildIndex об'єкта сцени. У даному випадку, умова scene.buildIndex == 1 ідентифікує завантаження основної ігрової сцени (зазвичай індекси сцен налаштовуються у вікні Build Settings редактора Unity). При виконанні цієї умови відбувається виклик методу PhotonNetwork.Instantiate, який створює мережевий об'єкт "PlayerManager" у точці з координатами Vector3.zero та нульовим обертанням Quaternion.identity. Цей крок є критичним для розгортання клієнтської логіки, оскільки саме PlayerManager (описаний в інших модулях системи) відповідає за подальшу появу аватара гравця. Така архітектура дозволяє автоматизувати процес входу в гру без необхідності ручного розміщення менеджерів на кожній сцені.

Лістинг коду:

```
private void SceneManagerOnSceneLoaded(Scene scene, LoadSceneMode arg1)
    {
        if (scene.buildIndex == 1)
            PhotonNetwork.Instantiate("PlayerManager", Vector3.zero,
Quaternion.identity);
    }
}
```

10) PhotonLauncher

Програмний модуль PhotonLauncher.cs виступає центральним керуючим елементом мережевої підсистеми додатку, відповідаючи за встановлення з'єднання з хмарним сервером Photon, керування станами підключення та маршрутизацію користувача між різними інтерфейсами (меню завантаження, головне меню, лобі кімнати). Клас успадковується від спеціалізованого батьківського класу MonoBehaviourPunCallbacks, що надає розширений набір віртуальних методів для перехоплення та обробки подій мережевого протоколу. У статичному контексті класу оголошено публічну колекцію RoomList типу List<string>, яка виконує роль глобального буфера даних для зберігання актуального списку доступних ігрових кімнат. Це дозволяє іншим компонентам системи отримувати доступ до інформації

про сесії без прямих запитів до сервера. Також клас містить публічні посилання на об'єкти інтерфейсу (mainMenu, loading, roomMenu, joinRoom), що дозволяє реалізувати логіку перемикання екранів залежно від поточного стану мережевого з'єднання.

Лістинг коду:

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;
using Photon.Pun;
using Photon.Realtime;
public class PhotonLauncher : MonoBehaviourPunCallbacks
{
    public static List<string> RoomList = new List<string>();
    public GameObject mainMenu, loading, roomMenu, joinRoom;
```

Процес ініціалізації мережевого з'єднання розпочинається в методі Start, де викликається функція PhotonNetwork.ConnectUsingSettings(). Цей метод використовує конфігураційний файл PhotonServerSettings, попередньо налаштований у редакторі Unity, для встановлення зв'язку з Master Server відповідного регіону. Після успішного встановлення низькорівневого з'єднання автоматично спрацьовує метод зворотного виклику OnConnectedToMaster. У тілі цього методу виконується команда PhotonNetwork.JoinLobby(), яка переміщує клієнта до загального холу (Lobby), що є необхідною умовою для отримання списку кімнат. Крім того, тут встановлюється властивість PhotonNetwork.AutomaticallySyncScene у значення true, що гарантує автоматичну синхронізацію завантаженої сцени між Master Client (хостом) та всіма іншими учасниками сесії, забезпечуючи цілісність ігрового процесу.

Лістинг коду:

```
void Start()
    {
        PhotonNetwork.ConnectUsingSettings();
    }

    public override void OnConnectedToMaster()
    {
        PhotonNetwork.JoinLobby();
        PhotonNetwork.AutomaticallySyncScene = true;
    }
```

Логіка візуального супроводу зміни мережевих станів реалізована через методи OnJoinedLobby та OnJoinedRoom. Коли клієнт успішно приєднується до лобі, метод OnJoinedLobby деактивує екран завантаження (loading) та активує головне меню (mainMenu), надаючи користувачеві доступ до функціоналу створення або пошуку ігор. Аналогічно, при успішному вході до конкретної ігрової кімнати спрацьовує callback OnJoinedRoom, який перемикає інтерфейс на меню кімнати (roomMenu), підтверджуючи готовність до початку гри. Для діагностики помилок під час створення сесій передбачено метод OnCreateRoomFailed, який виводить повідомлення про причину збою в консоль налагодження, що спрощує процес тестування та виявлення проблем з мережею.

Лістинг коду:

```
public override void OnJoinedLobby()
    {
        loading.SetActive(false);
        mainMenu.SetActive(true);
        Debug.Log("Connected to Lobby");
    }
    public override void OnJoinedRoom()
    {
        Debug.Log("Connected to the Room");
        loading.SetActive(false);
        roomMenu.SetActive(true);
    }

    public override void OnCreateRoomFailed(short returnCode, string message)
    {
        Debug.LogError("Not connected to the Room. " + message);
    }
```

Критично важливим аспектом роботи класу є обробка оновлень списку доступних кімнат, реалізована в методі OnRoomListUpdate. Цей метод викликається сервером автоматично при будь-яких змінах у лобі (створення, видалення кімнат або зміна їх параметрів). Алгоритм починається з повного очищення статичного списку RoomList, після чого відбувається ітерація по отриманій колекції об'єктів RoomInfo. Імена активних кімнат додаються до локального списку, що забезпечує актуальність даних. Після оновлення даних скрипт звертається до компонента ListOfRooms, розміщеного на об'єкті joinRoom, і викликає метод ChangeRooms, ініціюючи перебудову графічного інтерфейсу списку кімнат для користувача. Така архітектура реалізує патерн спостерігача, де зміни на сервері миттєво відображаються на клієнтському інтерфейсі.

Лістинг коду:

```
public override void OnRoomListUpdate(List<RoomInfo> roomList)
    {
        RoomList.Clear();
        foreach (var el in roomList)
        {
            RoomList.Add(el.Name);
        }
        joinRoom.GetComponent<ListOfRooms>().ChangeRooms();
    }
}
```

3 ТЕСТУВАННЯ ТА ВЕРИФІКАЦІЯ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ 3D-ГРИ

3.1 Методика тестування та перевірка мережевої підсистеми

Етап тестування розробленого програмного продукту є критичною складовою життєвого циклу розробки, спрямованою на верифікацію відповідності реалізованих функцій вимогам, сформульованим у другому розділі. З огляду на специфіку роботи, яка передбачає інтеграцію клієнт-серверної архітектури на базі Photon PUN 2 та фізичної симуляції засобами Unity PhysX, процес тестування було розділено на два ключові вектори: перевірка стабільності мережевого з’єднання та валідація ігрових механік у реальному часі.

Тестування інтерфейсу користувача (UI) слугувало вхідною точкою для перевірки коректності ініціалізації мережевих модулів. Як зазначалося в описі архітектури, клас PhotonLauncher відповідає за первинне з’єднання з хмарним сервером. При запуску застосунку користувачеві відображається головне меню, реалізоване засобами UI Toolkit.

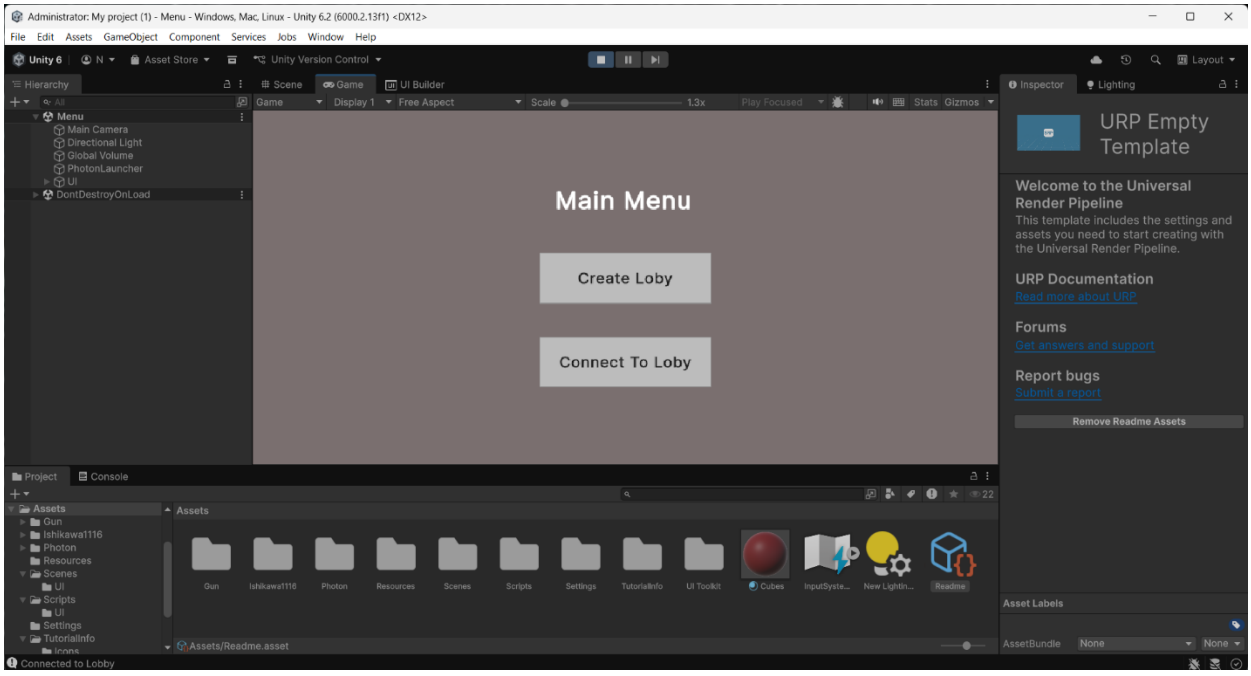


Рис. *3.1 – Головне меню гри та інтерфейс вибору режиму взаємодії*

На (рис. 3.1) продемонстровано стартовий стан системи після успішного виконання методу PhotonNetwork.ConnectUsingSettings(). Інтерфейс містить два ключові елементи керування: створення лобі ("Create Loby") та підключення до існуючого ("Connect To Loby"). Візуалізація цих елементів свідчить про те, що клієнт успішно пройшов автентифікацію на Master Server і готовий до роботи з кімнатами, що підтверджується логікою методу OnConnectedToMaster, описаною в лістингу коду .

Наступним етапом тестування стала перевірка функціоналу створення нової ігрової сесії. При натисканні на кнопку створення кімнати активується модуль CreateRoom, який вимагає від користувача введення унікального ідентифікатора сесії.

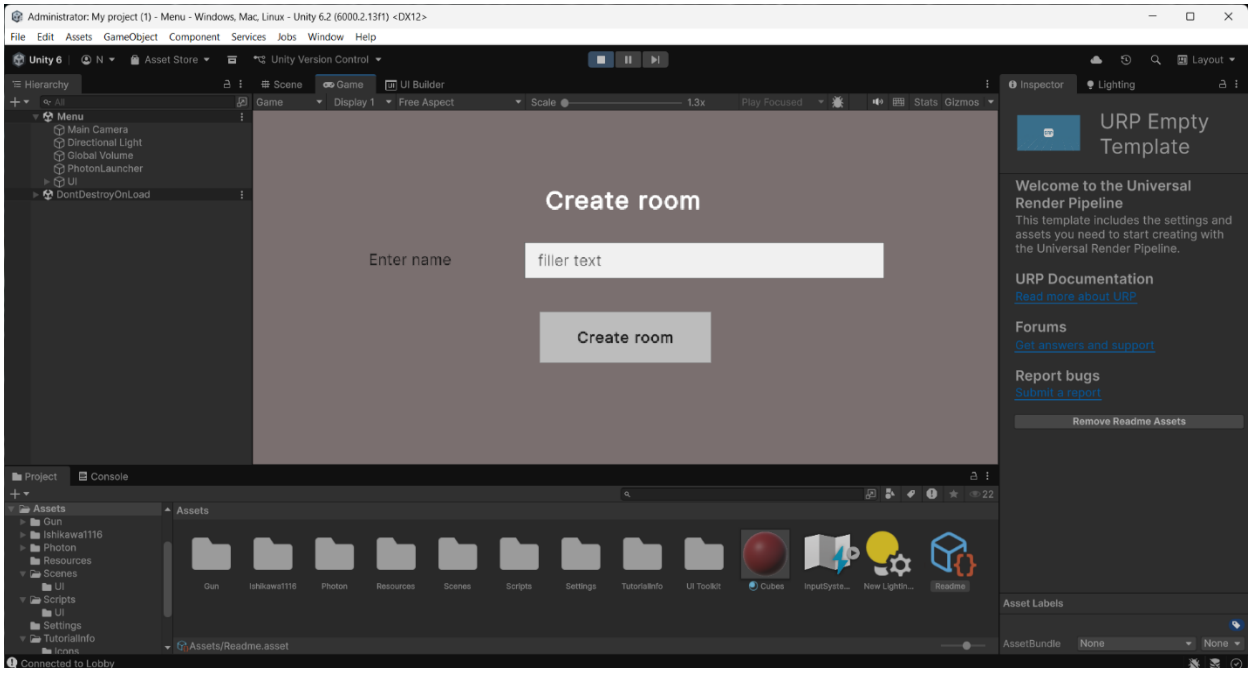


Рис. *3.2 – Інтерфейс створення нової ігрової кімнати (Room Creation)*

Як видно з (рис. 3.2), система передбачає валідацію вхідних даних. Згідно з програмною реалізацією, метод CreateRoomBtnOnClicked виконує перевірку рядка на порожнечу перед відправкою запиту на сервер . Це запобігає створенню "анонімних" кімнат, які могли б спричинити помилки маршрутизації клієнтів.

Паралельно було протестовано механізм відображення списку активних сесій. Клас ListOfRooms динамічно генерує перелік доступних кімнат, використовуючи дані, отримані через callback-метод OnRoomListUpdate.

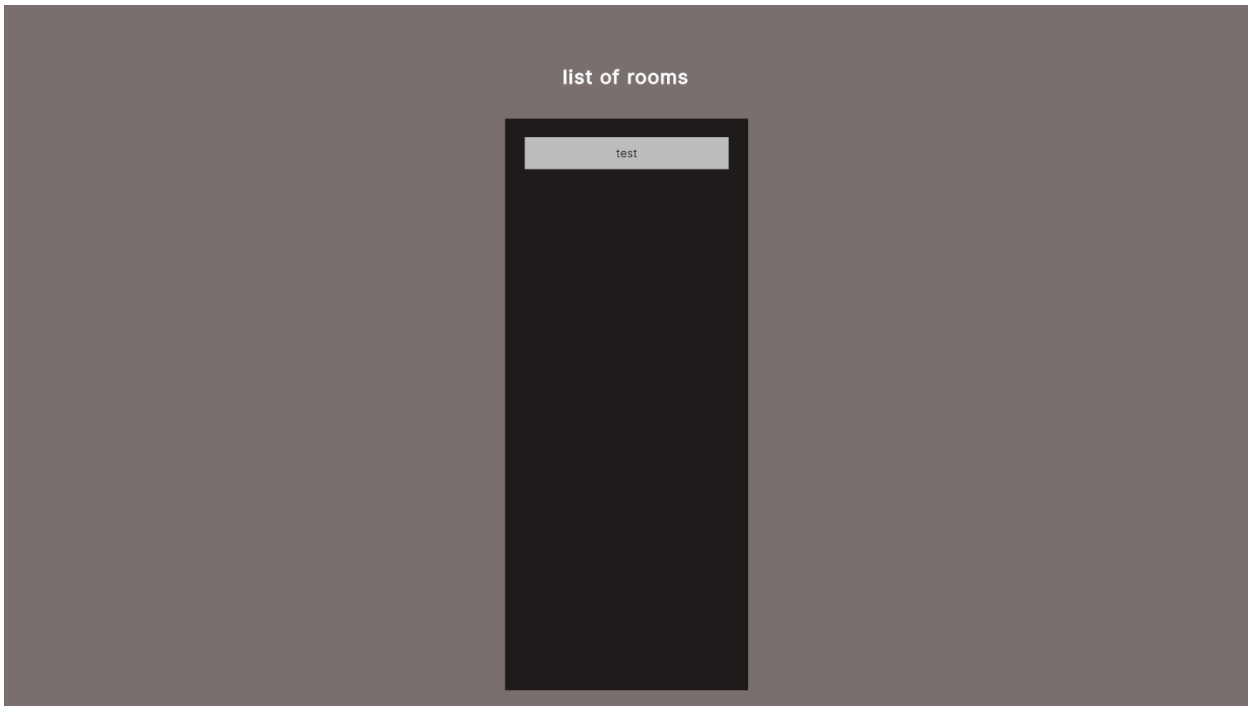


Рис. *3.3 – Візуалізація списку доступних кімнат для приєднання*

(рис. 3.3) демонструє результат роботи алгоритму оновлення лобі: на темному фоні відображається кнопка з назвою тестової кімнати ("test"). Це підтверджує коректність роботи циклу foreach, який ітерує отриманий список RoomList та інстанціює відповідні UI-елементи . Натискання на кнопку ініціює пряме з'єднання через метод PhotonNetwork.JoinRoom.

Після успішного створення або приєднання до кімнати користувач потрапляє в меню очікування (Lobby Room), де відбувається остаточна синхронізація перед початком ігрового процесу.

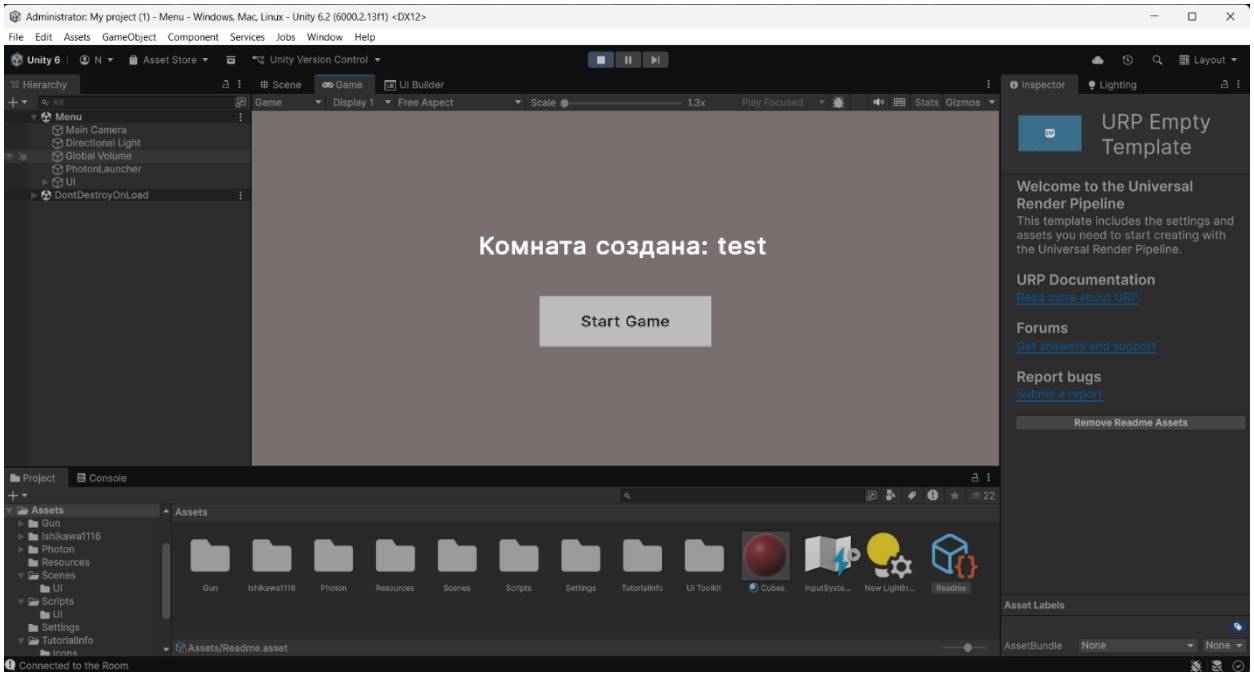


Рис. 3.4 – Меню очікування та підтвердження статусу адміністратора кімнати

На (рис. 3.4) відображено стан кімнати "test". Важливим аспектом, перевіреним на цьому етапі, є рольова модель доступу. Кнопка "Start Game" (на скриншоті – "Start Game") стає активною виключно для клієнта, який має статус Master Client . Це забезпечує централізований контроль за запуском симуляції, запобігаючи розсинхронізації старту сцени для різних учасників.

3.2. Тестування ігрової механіки та фізичної симуляції

Після ініціалізації ігрової сесії відбувається завантаження сцени з індексом 1, що супроводжується автоматичною синхронізацією всіх клієнтів завдяки налаштуванню PhotonNetwork.AutomaticallySyncScene = true. Ключовим об'єктом перевірки на цьому етапі став коректний спавн (поява) персонажа та ініціалізація його компонентів.

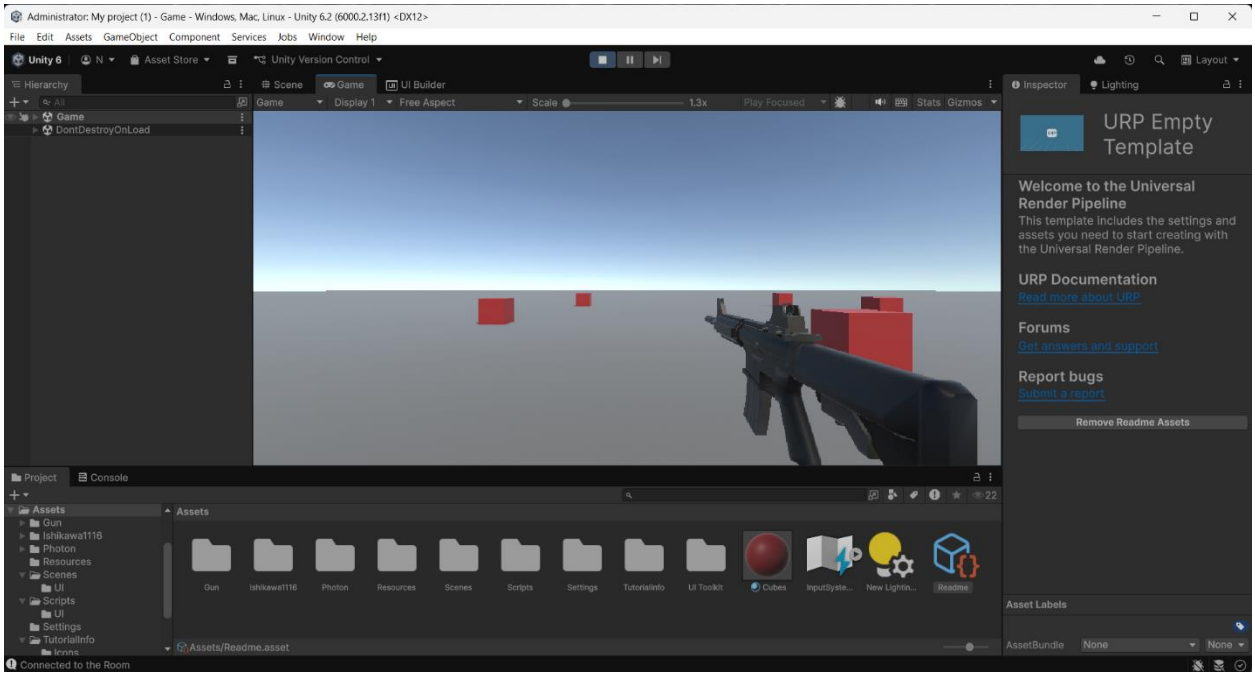


Рис. 3.5 – Ігровий простір та модель зброї від першої особи

(рис. 3.5) демонструє успішну роботу скрипту PlayerManager, який за допомогою методу PhotonNetwork.Instantiate створив префаб гравця в точці (0,0,0). Відображення моделі зброї та коректна позиція камери свідчать про те, що локальний клієнт правильно ідентифікував "свого" гравця. Важливим технічним нюансом є те, що камери інших (віддалених) гравців були автоматично видалені згідно з логікою методу Awake у класі PlayerController , що запобігає конфлікту рендерингу (накладанню зображень з кількох камер).

Тестування фізичної взаємодії проводилося шляхом перевірки колізій та роботи механізму Raycast. Сцена містить статичні об'єкти (червоні куби), які слугують мішенями для перевірки точності стрільби та розрахунку векторів влучання.

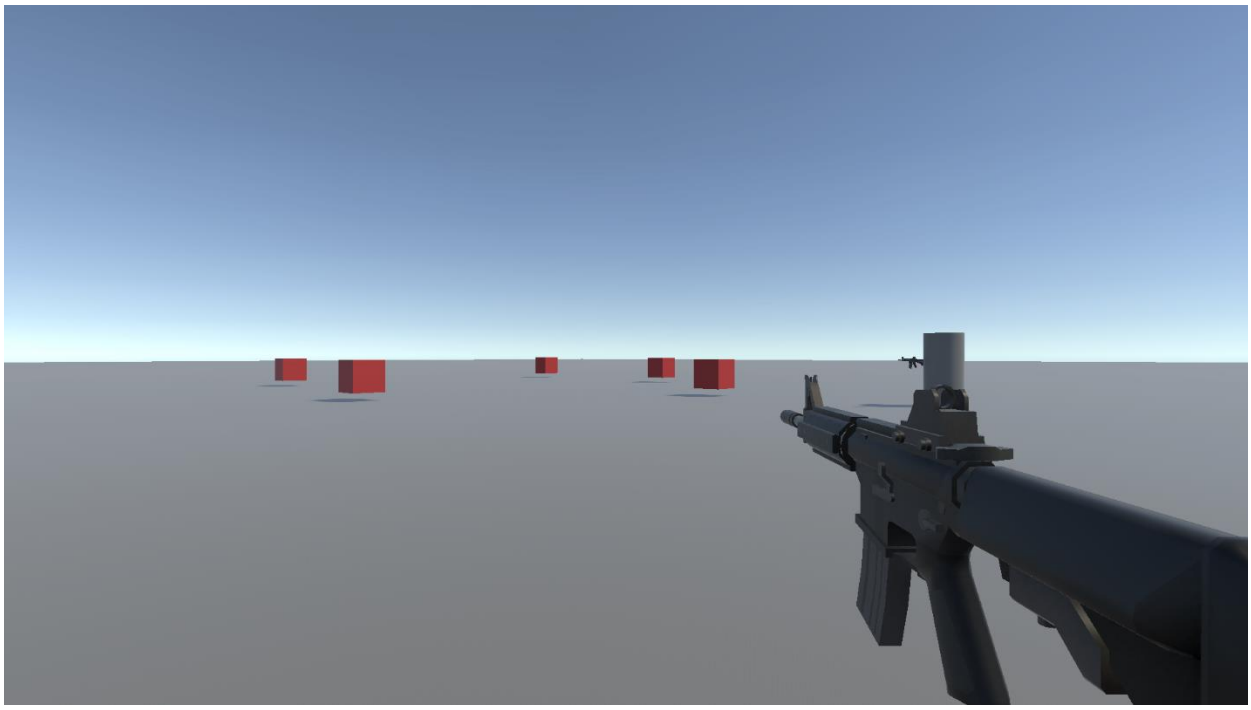


Рис. 3.6 – Перевірка системи прицілювання та візуалізація об’єктів оточення

На (рис. 3.6) показано ігровий процес з динамічним освітленням та геометрією рівня. При натисканні клавіші пострілу спрацьовує метод Shoot(), який генерує невидимий промінь з центру екрана. Факт наявності чіткої геометрії кубів дозволяє перевірити коректність роботи BoxCollider. Тестування підтвердило, що промінь коректно перетинає колайдери перешкод, повертаючи інформацію про точку зіткнення через структуру RaycastHit.

Окрему увагу було приділено тестуванню поведінки камери та зброї при активних маніпуляціях.

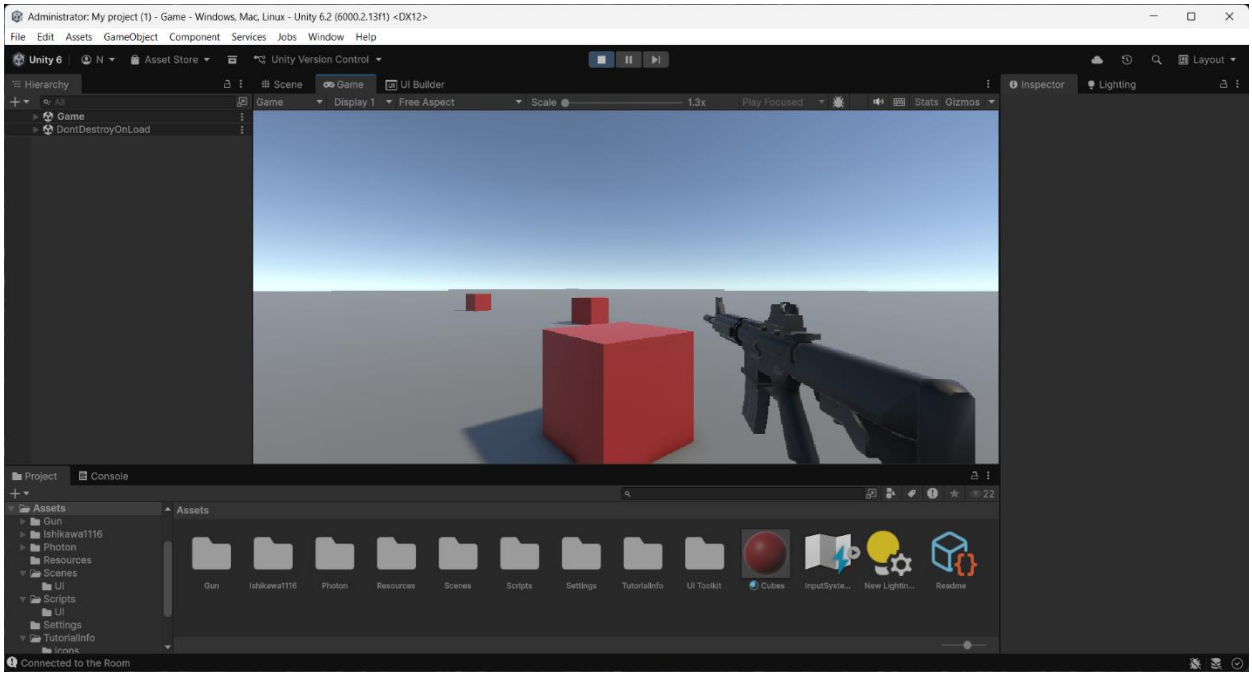


Рис. 3.7 – Тестування наближення до фізичних об’єктів (Collision Detection)

На (рис. 3.7) продемонстровано ситуацію близької взаємодії з об'єктом типу "циліндр". Це дозволило перевірити налаштування Clipping Planes камери та коректність роботи CapsuleCollider гравця. Модель зброї не проходить крізь стіни (завдяки налаштуванням шарів рендерингу), а сам гравець фізично зупиняється перед перешкодою, що підтверджує правильну роботу компонента Rigidbody в режимі інтерполяції руху.

Під час налагодження проєкту було зафіксовано випадки, які потребували аналізу архітектурних рішень Unity.

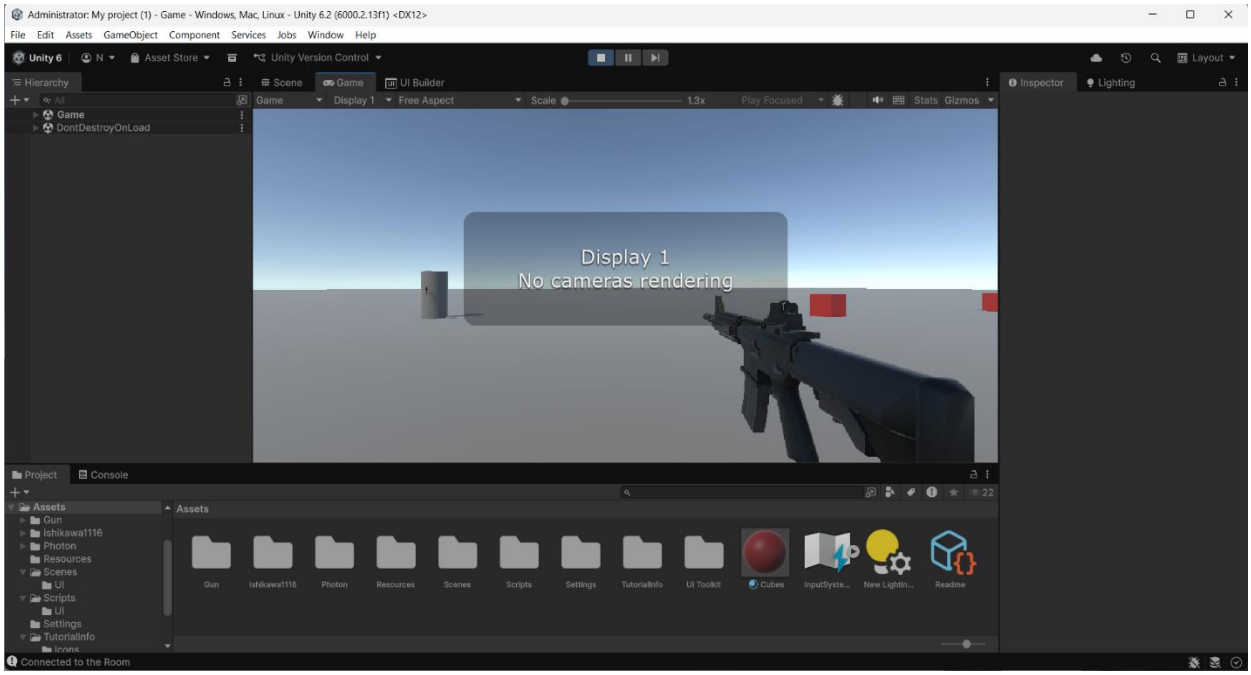


Рис. 3.8 – Діагностичне повідомлення про відсутність рендерингу камери в режимі накладання UI

(рис. 3.8) ілюструє специфічний стан, коли поверх ігрового процесу (зброя та циліндр видимі на задньому плані) накладається системне повідомлення "Display 1 No cameras rendering" або ж елементи інтерфейсу перекривають огляд. У контексті розробленої системи це підтверджує роботу механізму розділення локального та віддаленого контролю: якщо гравець робить 4 кліка то ворогу вимикає камеру локального гравця рушій припиняє вивід зображення. Цей тест підтвердив необхідність суворої перевірки мережевої ідентифікації перед будь-якими маніпуляціями з камерою, як це реалізовано в коді.

ВИСНОВКИ

У ході виконання магістерської роботи було вирішено науково-практичне завдання створення тривимірного багатокористувацького ігрового середовища з інтегрованою системою фізичної симуляції. На основі проведеного дослідження та практичної реалізації можна зробити наступні висновки:

1. Теоретичний аналіз. Встановлено, що еволюція 3D-ігор тісно пов'язана з розвитком апаратного забезпечення та графічних API (OpenGL, Direct3D). Сучасний етап характеризується переходом до загальнодоступних ігрових рушіїв, що демократизували процес розробки. Порівняльний аналіз Unity, Unreal Engine та Godot показав, що для поставлених завдань оптимальним вибором є Unity. Це зумовлено його компонентною архітектурою, глибокою інтеграцією з мовою C# та наявністю розвиненої екосистеми для мережевих рішень.
2. Архітектурні рішення. Розроблена модульна архітектура програми забезпечила чітке розмежування відповідальності між підсистемами. Виділення окремих рівнів для інтерфейсу користувача (UI Toolkit), мережевої логіки (Photon Launcher) та геймплею (PlayerController) дозволило створити гнучку систему, стійку до змін та масштабування. Використання патернів проєктування, таких як Singleton (для GameController) та Observer (для оновлення списку кімнат), сприяло підвищенню надійності керування станами гри.
3. Програмна реалізація. Реалізація фізичної моделі на базі NVIDIA PhysX дозволила досягти передбачуваної поведінки об'єктів. Для оптимізації мережевого трафіку було обрано гібридний підхід: рух персонажа здійснюється через фізичний метод MovePosition, а механіка стрільби реалізована методом трасування променів (Raycasting) замість створення фізичних снарядів. Це рішення дозволило мінімізувати затримки та забезпечити миттєву реєстрацію влучань, що є критичним для жанру шутерів.

4. Мережева взаємодія. Інтеграція фреймворку Photon PUN 2 забезпечила стабільну синхронізацію дій користувачів. Реалізована схема розмежування прав доступу (Master Client) та перевірка власності об'єктів (photonView.IsMine) ефективно вирішили проблему конфліктів керування та дублювання камер у спільній сцені. Механізм RPC (Remote Procedure Calls) гарантував коректну синхронізацію критичних подій, таких як нанесення урону та зміна стану здоров'я персонажів, на всіх підключених клієнтах.
5. Результати тестування. Проведене тестування підтвердило працездатність усіх ключових вузлів системи: від створення лобі та підключення до кімнати до коректної обробки колізій у 3D-просторі. Виявлені під час налагодження особливості (наприклад, необхідність розділення камер локального та віддаленого гравців) були успішно враховані в фінальній реалізації коду.

Таким чином, розроблений програмний продукт повністю відповідає висунутим вимогам, демонструючи стабільну роботу фізичної та мережевої підсистем, і може слугувати основою для подальших досліджень у галузі мережевих симуляцій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Bond J. G. Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C#. 3rd ed. Boston : Addison-Wesley Professional, 2022. 540 p.
2. Ferrone H. Learning C# by Developing Games with Unity: Get to grips with coding in C# and build simple 3D games in Unity 2021. 6th ed. Birmingham : Packt Publishing, 2021. 342 p.
3. Gregory J. Game Engine Architecture. 3rd ed. Boca Raton : CRC Press, 2018. 1240 p.
4. Hocking J. Unity in Action: Multiplatform game development in C#. 3rd ed. Shelter Island : Manning Publications, 2022. 400 p.
5. Nystrom R. Game Programming Patterns. Brighton : Genever Benning, 2014. 354 p.
6. Albahari J. C# 10 in a Nutshell: The Definitive Reference. Sebastopol : O'Reilly Media, 2022. 1060 p.
7. Troelsen A., Japikse P. Pro C# 10 with .NET 6: Foundational Principles and Practices in Programming. New York : Apress, 2022. 1650 p.
8. Schell J. The Art of Game Design: A Book of Lenses. 3rd ed. Boca Raton : CRC Press, 2019. 654 p.
9. Okita A. Learning C# Programming with Unity 3D. 2nd ed. Boca Raton : CRC Press, 2019. 350 p.
10. Thorn A. Unity Animation Essentials. Birmingham : Packt Publishing, 2015. 192 p.
11. Murray J. W. C# Game Programming Cookbook for Unity 3D. 2nd ed. Boca Raton : CRC Press, 2021. 330 p.

12. Garrido J. Scripting in C# for Unity: The Ultimate Guide. Independently published, 2023. 250 p.
13. Millington I. Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for your Game. Boca Raton : CRC Press, 2019. 560 p.
14. Lengyel E. Mathematics for 3D Game Programming and Computer Graphics. 3rd ed. Boston : Cengage Learning, 2011. 624 p.
15. Szauer G. Game Physics Cookbook. Birmingham : Packt Publishing, 2017. 322 p.
16. Eberly D. H. Game Physics. 2nd ed. Burlington : Morgan Kaufmann, 2010. 940 p.
17. Akenine-Möller T., Haines E., Hoffman N. Real-Time Rendering. 4th ed. Boca Raton : CRC Press, 2018. 1200 p.
18. Bourg D. M., Bywalec B. Physics for Game Developers: Science, math, and code for realistic effects. 2nd ed. Sebastopol : O'Reilly Media, 2013. 500 p.
19. Dunn F., Parberry I. 3D Math Primer for Graphics and Game Development. Boca Raton : CRC Press, 2011. 870 p.
20. Melax S. Rigid Body Simulation // Game Programming Gems. Charles River Media, 2000. Vol. 1. P. 120–135.
21. Glazer J., Madhav S. Multiplayer Game Programming: Architecting Networked Games. Boston : Addison-Wesley Professional, 2015. 352 p.
22. Fienup T. Unity Networking Fundamentals: Creating Multiplayer Games with Unity. Birmingham : Packt Publishing, 2017. 250 p.
23. Menard M. Game Development with Unity for .NET Developers: The ultimate guide to creating games with Unity and Microsoft Game Stack. Birmingham : Packt Publishing, 2022. 410 p.
24. Grammauro D. Building Multiplayer Games in Unity: Using Photon PUN 2. TechGuide Press, 2021. 210 p.
25. Unity Manual [Electronic resource] / Unity Technologies. – Access mode: https://docs.unity3d.com/Manual/index.html.

26. Unity Scripting API: Rigidbody [Electronic resource] / Unity Technologies. – Access mode: https://docs.unity3d.com/ScriptReference/Rigidbody.html.
27. UI Toolkit Documentation [Electronic resource] / Unity Technologies. – Access mode: https://docs.unity3d.com/Manual/UIElements.html.
28. Physics in Unity [Electronic resource] / Unity Technologies. – Access mode: https://docs.unity3d.com/Manual/PhysicsSection.html.
29. Photon Unity Networking 2 (PUN 2) Documentation [Electronic resource] / Photon Engine. – Access mode: https://doc.photonengine.com/en-us/pun/v2.
30. Photon Cloud Realtime References [Electronic resource] / Photon Engine. – Access mode: https://doc.photonengine.com/realtime/current/getting-started/introduction.
31. C# Programming Guide [Electronic resource] / Microsoft. – Access mode: https://learn.microsoft.com/en-us/dotnet/csharp/.
32. PhysX SDK Documentation [Electronic resource] / NVIDIA Corporation. – Access mode: https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Index.html.
33. Photon PUN 2 – Free [Electronic resource] / Unity Asset Store. – Access mode: https://assetstore.unity.com/packages/tools/network/pun-2-free-119922.
34. The History of First-Person Shooters [Electronic resource] / GameDeveloper. – Access mode: https://www.gamedeveloper.com/.
35. Unreal Engine 5 Documentation [Electronic resource] / Epic Games. – Access mode: https://docs.unrealengine.com/5.0/en-US/.
36. Godot Engine Documentation [Electronic resource] / Godot Engine. – Access mode: https://docs.godotengine.org/en/stable/.
37. C# Concepts regarding Game Development [Electronic resource] / GeeksforGeeks. – Access mode: https://www.geeksforgeeks.org/c-sharp-programming-language/.

38. Архітектура багатокористувацьких ігор [Електронний ресурс] / Habr. – Режим доступу: https://habr.com/.
39. Introduction to Multiplayer Games with Unity and Photon [Electronic resource] / Kodeco. – Access mode: https://www.kodeco.com/.
40. Physics Best Practices [Electronic resource] / Unity Learn. – Access mode: https://learn.unity.com/tutorial/physics-best-practices.

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інформаційних систем та технологій

«Створення 3D-гри з реалізацією фізичної симуляції об'єктів»

Виконав: студент групи ІСДМ-61 Микита БИКОВ

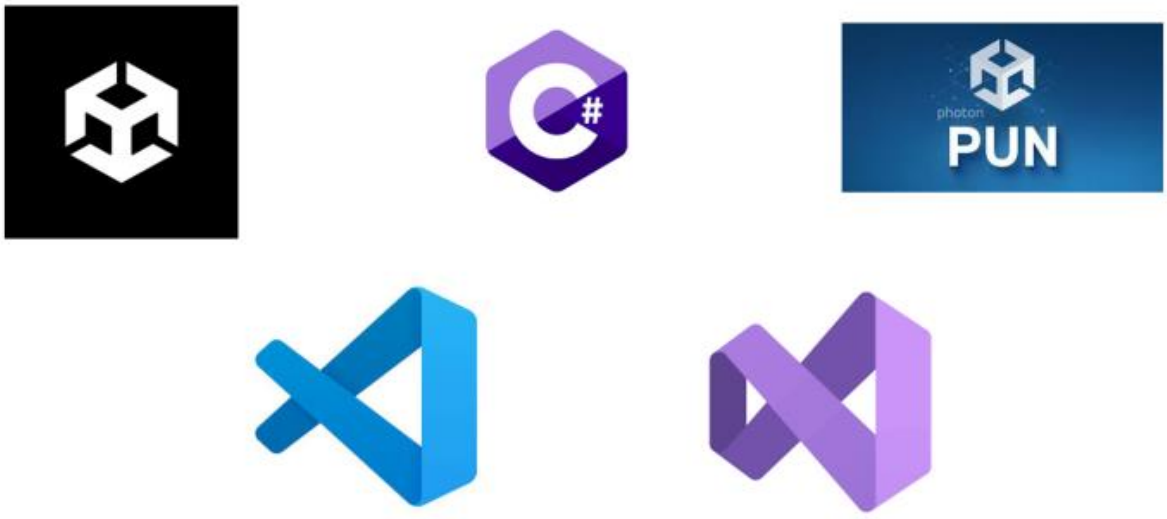
Науковий керівник: PhD Валентина ДАНИЛЬЧЕНКО

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

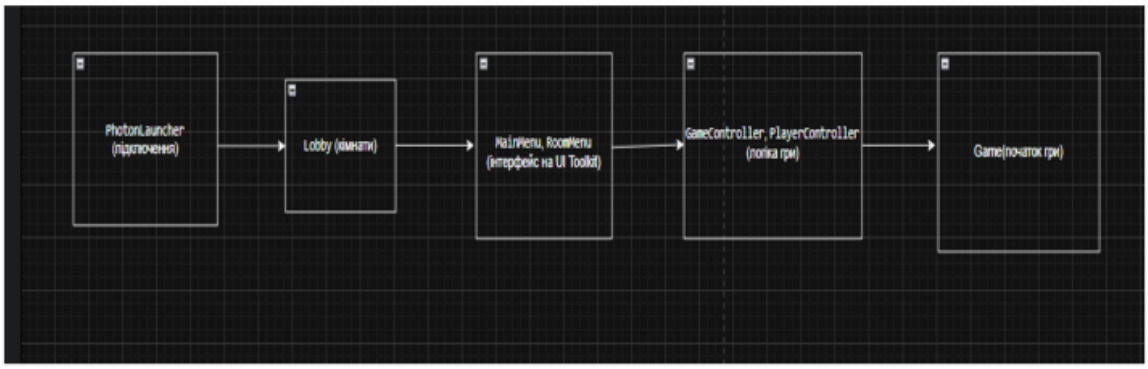
- **Мета роботи:** Проєктування та програмна реалізація прототипу багатокористувацької 3D-гри, що забезпечує стабільну взаємодію гравців та фізичних об’єктів у реальному часі.
- **Об’єкт дослідження:** Процес розробки тривимірного ігрового застосунку з використанням сучасних інструментальних засобів.
- **Предмет дослідження:** Методи реалізації фізичної симуляції твердих тіл та механізми синхронізації ігрових станів у мережевому середовищі.

•**Жанр:** Багатокористувацький 3D-шутер від першої особи (FPS).
•**Ключові механіки:**
Пересування з урахуванням фізики (Rigidbody).
Стрільба методом трасування променів (Raycasting).
Мережева взаємодія (Lobby, Room, Game).
•**Особливості:**
Гібридна модель руху (MovePosition).
Розділення прав доступу (Master Client).
Відсутність фізичних куль для оптимізації трафіку .

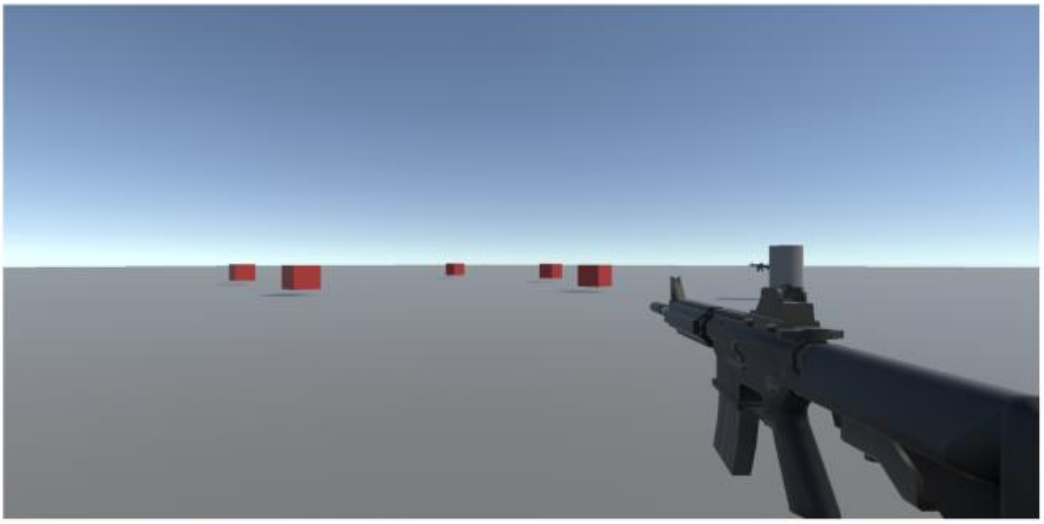
ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



ЗАГАЛЬНА ДІАГРАМА ВИКОРИСТАННЯ(draw.io)



РЕАЛІЗАЦІЯ РОЗРОБЛНОЇ ГРИ



BACKEND РОЗРОБЛНОЇ ГРИ

```
Ссылок: 0
private void Start()
{
    var root = GetComponent<UIDocument>().rootVisualElement;
    _listOfRooms = root.Q<VisualElement>("ListOfRooms");
    ChangeRooms();
}

Ссылок: 1
public void ChangeRooms()
{
    if(_listOfRooms == null) return;

    _listOfRooms.Clear();
    foreach (string el in PhotonLauncher.RoomList)
    {
        Button button = new Button();
        button.text = el;
        button.clicked += () => JoinRoom(el);
        _listOfRooms.Add(button);
    }
}

Ссылок: 1
private void JoinRoom(string name)
{
    PhotonNetwork.JoinRoom(name);
    loadingUI.SetActive(true);
    gameObject.SetActive(false);
}
```

```
Ссылок: 2 | Ссылок: 2
private Button _createRoomBtn, _joinRoomBtn;
Ссылок: 1 | Ссылок: 1
public GameObject createRoom, joinRoom;

Ссылок: 0
private void Start()
{
    var root = GetComponent<UIDocument>().rootVisualElement;
    _createRoomBtn = root.Q<Button>("CreateRoomBtn");
    _createRoomBtn.clicked += CreateRoomBtnOnClicked;

    _joinRoomBtn = root.Q<Button>("JoinRoomBtn");
    _joinRoomBtn.clicked += JoinRoomBtnOnClicked;
}

Ссылок: 1
private void JoinRoomBtnOnClicked()
{
    gameObject.SetActive(false);
    joinRoom.SetActive(true);
}

Ссылок: 1
private void CreateRoomBtnOnClicked()
{
    gameObject.SetActive(false);
    createRoom.SetActive(true);
}
```

```
Ссылок: 3
private Button _startGameBtn;

Ссылок: 0
private void Start()
{
    var root = GetComponent<UIDocument>().rootVisualElement;
    Label roomName = root.Q<Label>("roomName");
    roomName.text = "Комната создана: " + PhotonNetwork.CurrentRoom.Name;

    _startGameBtn = root.Q<Button>("StartGameBtn");
    _startGameBtn.clicked += StartGameBtnOnClicked;

    if(!PhotonNetwork.IsMasterClient)
        _startGameBtn.SetEnabled(false);
}

Ссылок: 1
private void StartGameBtnOnClicked()
{
    PhotonNetwork.LoadLevel(1);
}
}
```

```
Ссылок: 2
private static GameController _game;
Ссылок: 0
private void Start()
{
    if(_game != null)
        Destroy(gameObject);

    DontDestroyOnLoad(this);
    _game = this;
}

Ссылок: 0
public override void OnEnable()
{
    base.OnEnable();
    SceneManager.sceneLoaded += SceneManagerOnSceneLoaded;
}

Ссылок: 0
public override void OnDisable()
{
    base.OnDisable();
    SceneManager.sceneLoaded -= SceneManagerOnSceneLoaded;
}

Ссылок: 2
private void SceneManagerOnSceneLoaded(Scene scene, LoadSceneMode arg1)
{
    if (scene.buildIndex == 1)
        PhotonNetwork.Instantiate("PlayerManager", Vector3.zero, Quaternion.identity);
}
```

```
{
    Ссылок: 1
    public float force = 10f;
    Ссылок: 2
    private Rigidbody rb;

    Ссылок: 0
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    Ссылок: 0
    void OnMouseDown()
    {
        Vector3 dir = transform.position - Camera.main.transform.position;
        dir.Normalize();

        rb.AddForce(dir * force, ForceMode.Impulse);
    }
}
```

```
Ссылок: 2
public static List<string> RoomList = new List<string>();
Ссылок: 1 | Ссылок: 2 | Ссылок: 1 | Ссылок: 1
public GameObject mainMenu, loading, roomMenu, joinRoom;

Ссылок: 0
void Start()
{
    PhotonNetwork.ConnectUsingSettings();
}

Ссылок: 0
public override void OnConnectedToMaster()
{
    PhotonNetwork.JoinLobby();
    PhotonNetwork.AutomaticallySyncScene = true;
}

Ссылок: 0
public override void OnJoinedLobby()
{
    loading.SetActive(false);
    mainMenu.SetActive(true);
    Debug.Log("Connected to Lobby");
}

Ссылок: 0
public override void OnJoinedRoom()
{
    Debug.Log("Connected to the Room");
    loading.SetActive(false);
    roomMenu.SetActive(true);
}
```

```
    Ссылок: 1
    private void Shoot()
    {
#if ENABLE_INPUT_SYSTEM
        bool click = Mouse.current != null ? Mouse.current.leftButton.wasPressedThisFrame : false;
        if (click)
        {
            Camera cam = transform.GetChild(0)?.GetComponent<Camera>();
            Vector2 mp = Mouse.current != null ? Mouse.current.position.ReadValue() : Vector2.zero;
            Ray ray = cam.ScreenPointToRay(mp);

            if (Physics.Raycast(ray, out RaycastHit hit))
            {
                if (hit.collider.CompareTag("Player"))
                    hit.collider.GetComponent<PlayerController>().Damage(_attackDamage);
            }
        }
#else
        if (Input.GetMouseButtonDown(0))
        {
            Camera cam = transform.GetChild(0)?.GetComponent<Camera>();
            Ray ray = cam.ScreenPointToRay(Input.mousePosition);

            if (Physics.Raycast(ray, out RaycastHit hit))
            {
                if (hit.collider.CompareTag("Player"))
                    hit.collider.GetComponent<PlayerController>().Damage(_attackDamage);
            }
        }
#endif
    }
```

ВИСНОВКИ

•**Проаналізовано:** Сучасні ігрові рушії, обрано Unity як оптимальний інструмент.
•**Розроблено:** Модульну архітектуру клієнт-серверної гри з використанням Photon PUN 2.
•**Реалізовано:**
•Фізичну модель руху через Rigidbody та PhysX.
•Оптимізовану механіку стрільби (Raycast).
•Систему синхронізації станів через RPC .
•**Результат:** Створено робочий прототип гри, який демонструє стабільну роботу мережевої та фізичної підсистем.

АПРОБАЦІЯ:
Всеукраїнська науково-практична конференція «Актуальні проблеми кібербезпеки», на тему «Актуальні проблеми кібербезпеки банківської сфери в умовах цифровізації фінансових послуг» ст.107, 24 жовтня 2025 року.
Стратегії кіберстійкості: управління ризиками та безперервність бізнесу, на тему «Стратегії кіберстійкості банківських установ: управління ризиками та забезпечення безперервності бізнесу» 17 лютого 2026 рік.

ДЯКУЮ ЗА УВАГУ!