МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
Факультет кібербезпеки та програмної інженерії
Кафедра інженерії програмного забезпечення

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

Катерина Нестеренко
“____”_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
МАГІСТРА

Тема: “Мережева система в багатокористувацьких онлайн іграх”

Виконавець: Голобородько Владислав Сергійович

Керівник: к.т.н., доцент Терещенко Лідія Юріївна

Нормоконтролер: к.ф..м.н. Гололобов Дмитро Олександрович

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки та програмної інженерії
Кафедра інженерії програмного забезпечення
Освітній ступінь магістр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Програмне забезпечення систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Катерина Нестеренко
"___" _____ 2023 р

ЗАВДАННЯ
на виконання дипломної роботи студента
Голобородька Владислава Сергійовича

1. Тема дипломної роботи: «Мережева система в багатокористувацьких онлайн іграх»
затверджена наказом ректора від 29.09.2023 р. № 1994/ст.

2. Термін виконання проекту: з 2.10.2022 р. до 31.12.2023 р.

3. Вихідні дані до роботи : розробити мережевий плагін та багатокористувацьку онлайн гру за допомогою двигуна Unreal Engine 5

4. Зміст пояснювальної записки:
   1. Огляд та аналіз мережевих інструментів.
   2. Мережева архітектура та технології в онлайн системах.
   3. Проектування та розробка гри.

5. Перелік обов'язкових слайдів презентації:
   1. Про програмний продукт.
   2. Актуальність проблеми.
   3. Огляд існуючих рішень.
   4. Аналіз ігрових двигунів.
   5. Використані технології.
   6. Головне меню гри.
   7. Ігрова локація.
   8. Демонстрація ігрового режиму
   9. Висновки

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1.	Розробка та затвердження графіка роботи	2.10.2023	
2.	Підготовка та написання 1 розділу. Відсилка керівнику	9.10.2023 – 18.10.2023	
3.	Підготовка та написання 2 розділу. Відсилка керівнику	12.10.2023– 15.10.2023	
4.	Написання гри	15.10.2023 – 1.12.2023	
5.	Підготовка та написання 3 розділу. Відсилка керівнику	16.10.2023 – 19.11.2023	
6.	Внесення правок та друкування пояснювальної записки. Відсилка ПЗ для перевірки на плагіат одним файлом.	11.12.2023 – 15.12.2023	
7	Проходження нормоконтролю. Підготовка матеріалів для презентації. Отримання відгуку від керівника.	4.12.2023 – 10.12.2023	
8.	Передзахист. При наявності цього приймається рішення про допуск до захисту дипломного проекту перед Екзаменаційною комісією	11.12.2023 – 17.12.2023	
9.	Отримання рецензії.	11.12.2023 – 17.12.2023	
10.	Підготовка матеріалів для передачі секретарю ДЕК	18.12.2023 – 25.12.2023	
11.	Захист дипломної роботи	27.12.2023 – 28.12.2023	

Дата видачі завдання 02.10.2023 р.

Керівник дипломної роботи: к.т.н. доцент Лідія ЮРІЇВНА

Завдання прийняв до виконання: Владислав ГОЛОБОРОДЬКО

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Мережева система в багатокористувацьких онлайн іграх»: 74 сторінок, 46 рисунків, 18 використаних джерел, 2 додатки.

Об'єкт дослідження – мережева система в багатокористувацьких онлайн іграх

Мета дипломної роботи – дослідження існуючих мережевих систем, аналіз ігрових двигунів, методів компенсації затримки, створення власного плагіну для перетворення однокористувацької гри на багатокористувацьку онлайн гру, створення гри за жанром онлайн шутеру

Метод дослідження – ігровий двигун Unreal Engine 5, комп'ютерна гра

Результати роботи можуть бути використані як готовий плагін при розробці багатокористувацьких онлайн іграх або при перетворенні однокористувацької гри на багатокористувацьку онлайн гру

Розробка та дослідження виконувалися під керівництвом ОС Windows 10. Програмний продукт був розроблений в середовищі Microsoft Visual Studio, Unreal Engine 5 на мові програмування C++та візуальній мові програмування Blueprint. UNREAL ENGINE, МЕРЕЖЕВА СИСТЕМА, ГРА, BLUEPRINT.

ABSTRACT

Explanatory note to the thesis "Network system in multiplayer online games": 74 pages, 46 figures, 18 used sources, 2 appendices.

The object of research is a network system in multiplayer online games

The purpose of the thesis is to research existing network systems, analyze game engines, delay compensation methods, create your own plug-in to convert a single-player game into a multiplayer online game, create a game based on the online shooter genre

The research method is the Unreal Engine 5 game engine, a computer game

The results of the work can be used as a ready-made plugin when developing multiplayer online games or when converting a single-player game to a multiplayer online game

The development and research was carried out under Windows 10 operating system. The program was developed in the Microsoft Visual Studio environment, Unreal Engine 5 in the C++ programming language and the Blueprint visual programming language.

UNREAL ENGINE, NETWORK SYSTEM, GAME, BLUEPRINT.

ЗМІСТ

ПЕРЕЛІК ПРИЙНЯТИХ СКОРОЧЕНЬ

RTT – час пінгування;

RPC – Remote Procedure Call;

UNet – Unity Networking;

PC – персональний комп’ютер;

ОС – операційна система;

ООП – об’єктно-орієнтоване програмування;

ПК – персональний комп’ютер;

ШІ – штучний інтелект;

UE – Unreal Engine;

P2P – peer-to-peer;

Ping – пінг.

ВСТУП

Сучасна ігрова індустрія відзначається стрімким ростом популярності багатокористувацьких онлайн ігор. Гра Fortnite від Epic Games яка розроблена за допомогою двигуна Unreal Engine у 2023 році встановила новий рекорд по онлайну – у грі одномоментно знаходилось 3.9 мільйонів користувачів, а кількість активних користувачів 45 мільйонів [1]. Цей феномен відкриває перед розробниками ігор безмежні можливості для створення захопливих інтерактивних віртуальних світів, де гравці з усього світу можуть взаємодіяти, спільно змагатися та спілкуватися в реальному часі. Однак для досягнення успіху в цій області необхідно мати глибоке розуміння та володіти навичками розробки багатокористувацьких ігор.

Ця магістерська робота присвячена вивченню та аналізу мережевої системи в багатокористувацьких онлайн іграх, розроблених з використанням потужного інструменту для геймдеву - Unreal Engine. Мета даного дослідження полягає в розкритті ключових аспектів створення мережевих ігор у середовищі Unreal Engine, а також в оцінці їхнього впливу на якість та функціональність багатокористувацьких ігор. Проаналізувати основні принципи створення клієнт-серверної архітектури, розглянути різні протоколи та технології, які можна використовувати в Unreal Engine для мережевої взаємодії гравців, і розібрати найкращі практики створення стабільних та ефективних багатокористувацьких ігор. Для зручності використання усіх рішень було обрано створити власний плагін з усіма налаштуваннями, який можна додати до будь-якого проєкту Unreal Engine, щоб легко перетворити його на онлайн багатокористувацьку гру.

Сучасні ігри вимагають від розробників не лише створення захоплюючого геймплею, гарної графіки та захисту від шахрайства, але й забезпечення стабільної та низько затратної мережевої взаємодії між гравцями. Однією з ключових складових успіху є використання мережевих технологій та протоколів. Для комфортної гри навіть за наявності великої інтернет затримки у гравців, було запроваджено методи компенсації затримок, як передбачення на стороні клієнта та на стороні сервера, щоб гра працювала стабільно. Розроблена власна підсистема для

керування онлайн-сесіями. На базі онлайн системи, було обрано створити багатокористувацький онлайн шутер. Для нього розглянути і застосувати компоненти підтвердження попадання, систему відстежування гравців.

Поглиблення в світ мережевих ігор у Unreal Engine - це крок у майбутнє розробки ігор, який має великий потенціал для творчості та розвитку. Результати цієї магістерської роботи не лише сприятимуть розширенню наукових знань у галузі розробки багатокористувацьких ігор, але й стануть корисними практичними рекомендаціями для розробників, які прагнуть додати мережеву систему до свого проекту або покращити якість та ефективність існуючої системи, розробленої з використанням платформи Unreal Engine.

РОЗДІЛ 1
ОГЛЯД ТА АНАЛІЗ МЕРЕЖЕВИХ ІНСТРУМЕНТІВ

В сучасному розвитку індустрії відеоігор особливо важливим аспектом є можливість створення захопливих та інноваційних мережевих ігор, які забезпечують оптимальну геймплейну якість та високий рівень взаємодії гравців у мережі. У зв'язку з цим, розділ "Огляд та аналіз мережевих інструментів" моєї дипломної роботи визначається як ключовий елемент, спрямований на вивчення та систематичний аналіз передових мережевих інструментів, які активно використовуються у сучасній розробці відеоігор.

У світлі швидкого розвитку індустрії відеоігор, важливо враховувати, що використання мережевих інструментів визначає успіх будь-якого сучасного гейм-проекту. У цьому контексті ми приділяємо особливу увагу чотирьом визначеним провідним інструментам:

- Unreal Engine,
- Unity,
- CryEngine,
- Amazon Lumberyard.

Кожен з цих двигунів має свої власні унікальні риси, що визначають їхню відмінність та призначення в контексті розробки мережевих ігор.

Аналіз цих платформ дозволить нам глибше зрозуміти їхні функціональні можливості, а також визначити переваги, обмеження та доступність, які вони надають розробникам. Порівняння функціонала ігрових двигунів: Unreal Engine, Unity, CryEngine та Amazon Lumberyard допоможе визначити оптимальний вибір для конкретного ігрового-проекту, враховуючи його унікальні вимоги, бюджет та специфікації.

Зокрема, у цьому розділі ми детально розглянемо технічні можливості, архітектурні особливості та інструментальні засоби, які надає кожен з двигунів, зокрема з фокусом на їхню здатність до оптимізації та підтримки мережевої взаємодії гравців. Цей розділ буде служити підґрунтям для подальших висновків та

рекомендацій щодо вибору мережевого інструменту для конкретного розробника чи команди розробників відеоігор.

1.1 Unreal Engine

Unreal Engine є одним із найпопулярніших ігрових мережевих двигунів на ринку. Він відомий своєю потужною графікою та розширеними можливостями для розробки багатокористувацьких ігор. UE розробила компанія Epic Games для своєї гри під назвою Unreal, і після цього двигун став популярним. Написаний мовою програмування C++ та двигун має власну систему візуального програмування — блюпрінти[2].

Редактор двигуна Unreal Editor створений за принципом, що підсумковий результат не відрізнятиметься від його зображення в 3D Viewport. Редактор дуже зручний для використання: всі ассети (моделі, джерела освітлення, візуальні ефекти) можна відразу розміщати на ігровій сцені, перетягнувши їх з папок де вони зберігаються. У своїй основі Unreal Editor можна назвати комплексною системою, що складається з безлічі редакторів, яка спрямована на те, щоб зробити процес розробки максимально цілісним. На рисунку 1.1. зображено меню редактора двигуна Unreal Engine.

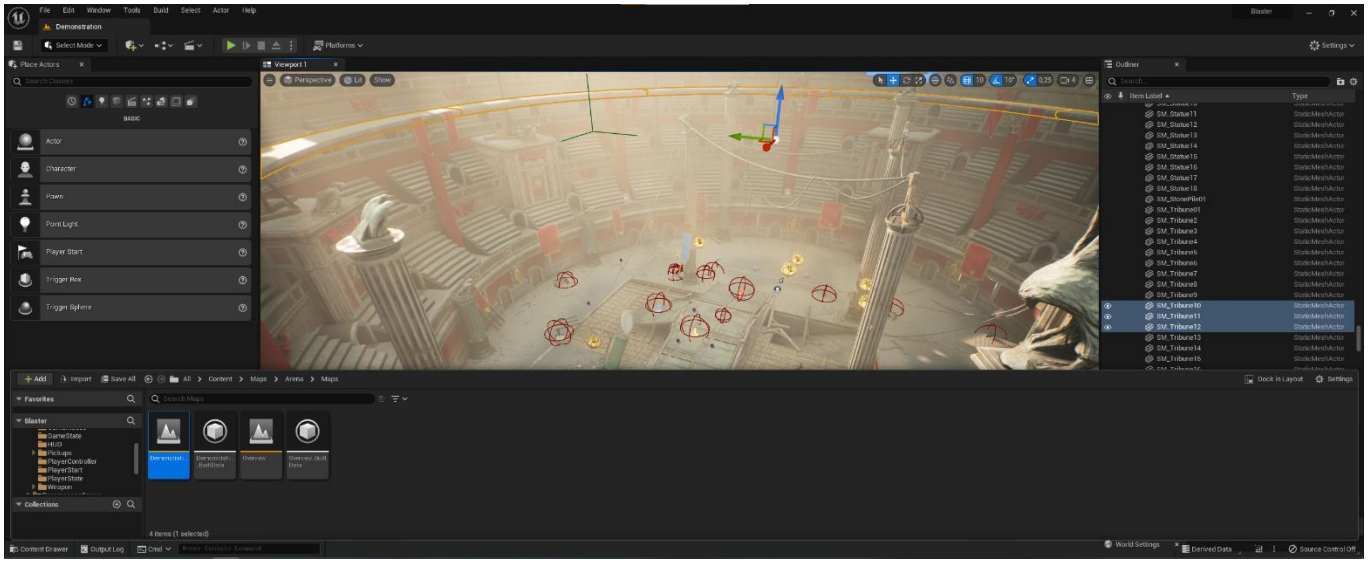

Рис. 1.1. Меню редактора двигуна Unreal Engine

Основні можливості двигуна Unreal Engine:

1 Оптимізація: Як мову програмування для Unreal Engine використовується C++. Це потужна, швидка, але досить складна мова, яку непросто вивчити з нуля. Проте, застосування якої дозволяє добре оптимізувати ігри.

2 Висока якість графіки: Unreal Engine славиться своєю потужною графікою та можливістю створювати різноманітні ігрові світи з докладними текстурами, освітленням і ефектами.

3 Мережева система: Unreal Engine включає вбудовану мережеву систему, яка дозволяє легко створювати багатокористувацькі ігри з підтримкою різних архітектур мережі, таких як client-server та peer-to-peer.

4 Розширені можливості фізики: Unreal Engine підтримує потужну систему фізики, яка дозволяє створювати реалістичні фізичні взаємодії між об'єктами в грі.

5 Багатоплатформність: Unreal Engine дозволяє розробляти ігри для різних платформ: ПК, консолі та мобільні пристрої.

6 Популярність: У UE величезна та товариська спільнота, з великою кількістю навчальних матеріалів та готових ассетів, які можна вільно використовувати у своїх проектах, що скорочує вартість та тривалість розробки.

7 Підтримка: Epic Games підтримує розробників. Проекти, які компанія визнає перспективними, можуть виграти грант на суму від 5 до 500 тисяч доларів.

8 Безкоштовна ліцензія:. У ліцензійній угоді зазначено, що доки гра не окупить себе і не принесе 1 мільйон доларів прибутку, двигуном можна користуватися безкоштовно. Далі доведеться сплачувати 5% від суми прибутку

Обмеження:

1. Складність в освоєнні: Unreal Engine вважається важким для освоєння, і розробникам може знадобитися багато часу та зусиль, щоб опанувати його.

2. Високі вимоги до обладнання: Розробка в Unreal Engine може вимагати потужних комп'ютерів та великої обчислювальної потужності.
3. Довгий час розробки: Створення ігор в ігровому двигуні Unreal Engine може займати більше часу, ніж на інших ігрових двигунах
4. Можливість роботи з C++: Хоча Unreal Engine підтримує розробку на C++, це може бути обмеженням для розробників, які більше впевнені в інших мовах програмування.

1.2 Unity

Unity - популярний ігровий двигун, який використовується для створення багатокористувацьких ігор. У ньому є різноманітні компоненти та інструменти для роботи з графікою, анімацією, фізикою об'єктів, звуком, шаблонами та скриптами[3]. На рисунку 1.2. зображено інтерфейс середовища Unity.

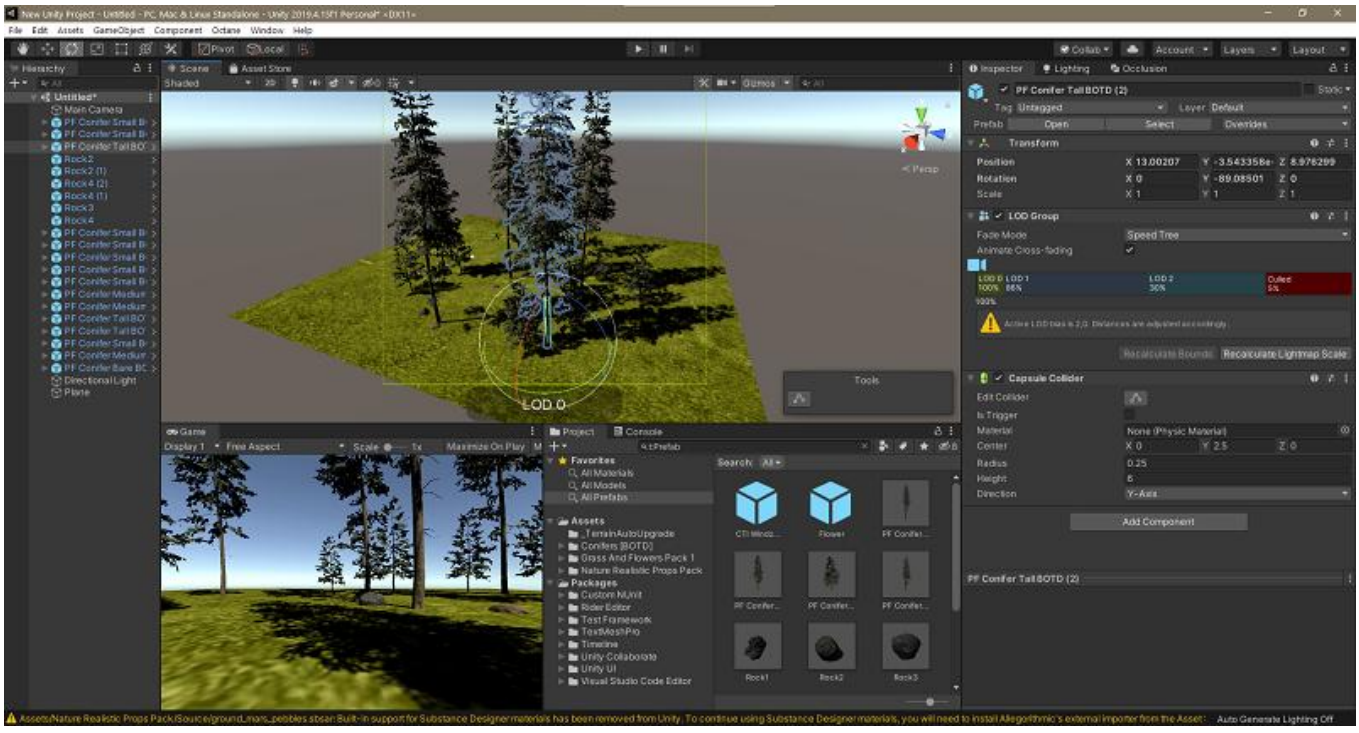

Рис. 1.2. Інтерфейс Unity

Можливості:

1. Широка підтримка платформ: Unity підтримує багато платформ, включаючи мобільні, консольні та веб ігри.

2. Спрощена розробка: Unity відомий своєю легкістю використання, що робить його досить доступним для новачків.
3. Активна спільнота: Unity має велику активну спільноту та багато ресурсів для навчання.

Обмеження:

1. Продуктивність: ігри розроблені на Unity можуть вимагати значних обчислювальних та графічних ресурсів, особливо якщо вони мають велику кількість динамічних об'єктів та складних ефектів. Це може призводити до низької продуктивності на слабкіших пристроях або відсутності можливості запускати гру на них взагалі.
2. Проблеми з оптимізацією: Оптимізація ігрового проекту є важливою задачею. Кросплатформові та крос жанрові двигуни мають меншу продуктивність у порівнянні з вузько направленими двигунами. Це впливає на швидкість роботи, якість графіки та FPS
3. Специфічні для платформи обмеження: Розробка для різних платформ може вимагати вирішення специфічних для них обмежень та вимог. Unity може обмежувати можливості розробників на деяких платформах.
4. Обмежена підтримка мов програмування: Unity в основному використовує C# для програмування, і хоча це потужна мова, деякі розробники можуть бажати використовувати інші мови, такі як C++.
5. Специфічні обмеження безкоштовної версії: Unity надає безкоштовну версію, але вона має обмеження, такі як обмеження дохідного порогу для безкоштовної версії та відсутність деяких додаткових функцій.

1.3 CryEngine

CryEngine – ігровий двигун відомий своєю деталізацією та реалізмом графіки. CryEngine планувався стати закритим двигуном для суто внутрішнього використання, але у 2016 році став доступний для використання та код двигуна був опублікованим на GitHub. На рисунку 1.3. зображено інтерфейс середовища CryEngine[4].

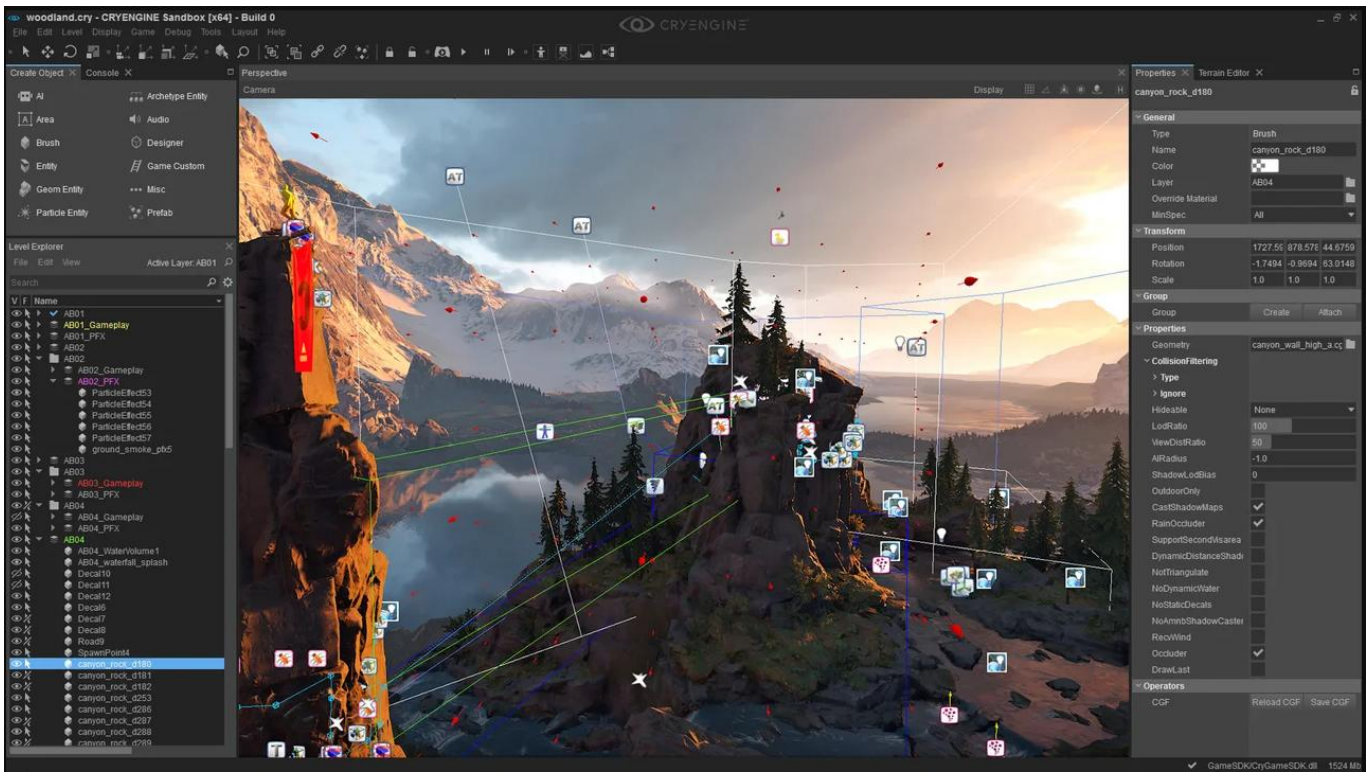

Рис. 1.3. Інтерфейс CryEngine

Основні можливості CryEngine:

1. Реалістична графіка: CryEngine відомий своєю деталізацією та графічною якістю[5].
2. Фізика і реалізм: Він має потужну фізичну систему, що дозволяє створювати реалістичні фізичні взаємодії.
3. Модульність: CryEngine підтримує модульність, що дозволяє розширювати його функціональність за потреби.

Основні обмеження CryEngine:

1. Високі вимоги до обладнання: Розробка в движку CryEngine може вимагати потужних комп'ютерів і графічних карт для досягнення максимальної якості графіки.
2. Обмежена спільнота і документація: У порівнянні з Unreal Engine та Unity, CryEngine може мати меншу спільноту та обмежену кількість доступної документації та ресурсів для розробників.

1.4 Amazon Lumberyard

Amazon Lumberyard - це ігровий рушій розроблений і випущений компанією Amazon Web Services Двигун призначений для створення відеоігор і віртуальних середовищ, і він поєднує в собі різні функції та інструменти для розробки ігор, а також підтримує кілька платформ[6]. На рисунку 1.4 зображено інтерфейс середовища Amazon Lumberyard.

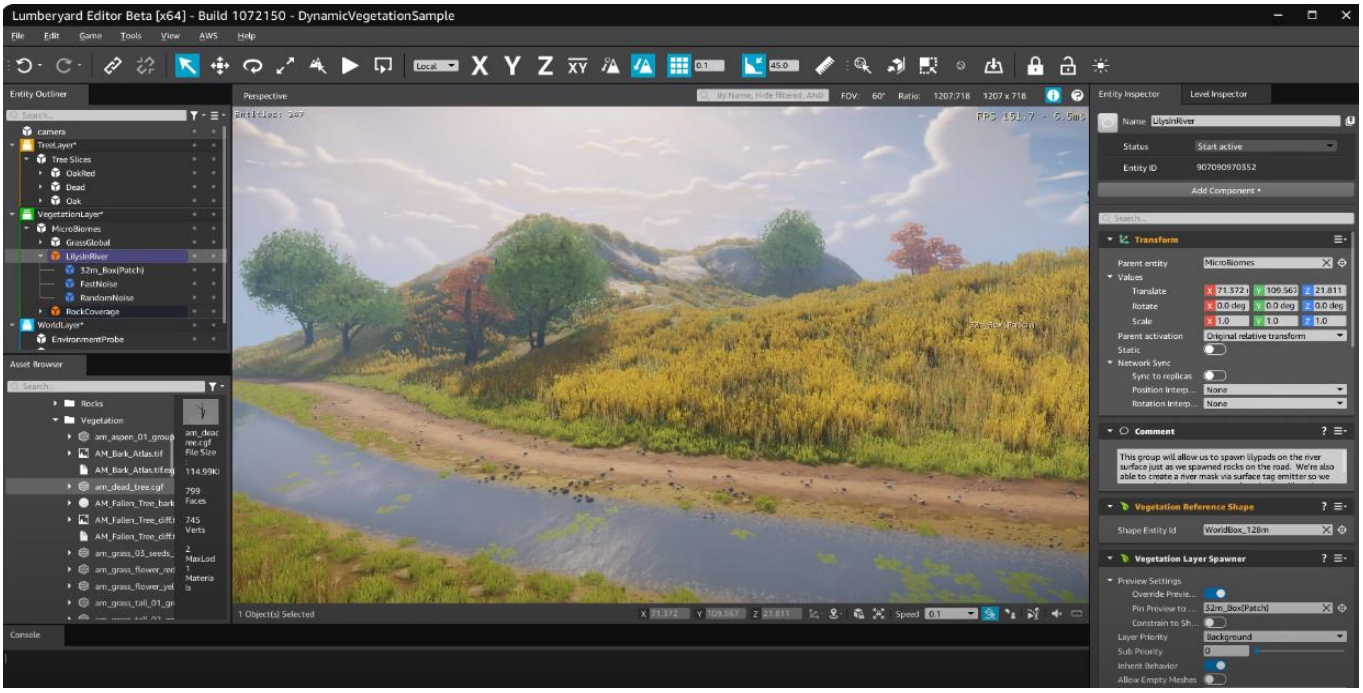

Рис. 1.4. Інтерфейс Amazon Lumberyard

Можливості:

1. Інтегрована система: двигун містить велику кількість інструментів для розробки ігор, такі як візуальний редактор рівнів, редактор матеріалів, анімації, фізики, аудіо та інші. Всі ці інструменти інтегровані в єдине середовище.
2. Інтеграція з AWS: платформа легко інтегрується з Amazon Web Services (AWS), що дозволяє використовувати хмарні ресурси для гри, включаючи обчислення, зберігання даних та багато іншого.

3. Широка підтримка для VR та AR: Движок підтримує віртуальну та доповнену реальність, що робить його цікавим вибором для створення VR- та AR-проектів.
4. Готові мультиплеєрні рішення: середовище розробки має вбудовану підтримку мультиплеєру, включаючи готові компоненти для створення мережевих ігор.
5. Велика спільнота: Є активна спільнота розробників, яка може надавати підтримку та допомогу з питань використання Lumberyard.

Обмеження:

1. Важкий навчальний поріг: Amazon Lumberyard - це складний движок, і вивчення його може бути викликом для новачків. Він вимагає досвіду в розробці ігор.
2. Обмежена підтримка мови програмування: Lumberyard використовує власну мову Lua Script Canvas, що може бути обмеженням для розробників, які вже володіють іншими мовами програмування, такими як C++.
3. Обмеження на платформах: На цей час двигун підтримує обмежену кількість платформ для виведення ігор.
4. Питання щодо стійкості та публікації: Деякі розробники вказують на питання щодо стійкості та публікації ігор, створених на Lumberyard.
5. Обмежена спільнота порівняно з іншими движками, такими як Unity або Unreal Engine.

Amazon Lumberyard - це цікавий движок для розробки ігор, особливо для тих, хто планує використовувати хмарні ресурси AWS. Проте, він може бути складним для новачків та вимагати певного часу для вивчення та освоєння всіх можливостей та обмежень.

Висновки

Розглядаючи чотири визначених ігрових двигунів - Unreal Engine, Amazon Lumberyard, CryEngine та Unity - у контексті їхніх мережевих інструментів, визначив, що кожен з них має свої унікальні можливості та особливості, які відображаються на різноманітних аспектах розробки багатокористувацьких ігор.

Unreal Engine вражає своєю повноцінною мережевою підтримкою, вбудованими інструментами для реалізації як клієнт-серверних, так і peer-to-peer мережевих рішень. Його розвинуті засоби дозволяють створювати масштабні багатокористувацькі ігри з великими можливостями для налаштувань.

Amazon Lumberyard, зі своєю інтеграцією з Amazon Web Services, вирізняється хмарною підтримкою та можливістю використання розподілених ресурсів для покращення мережевої продуктивності та масштабованості проектів.

Unity, зі своєю великою популярністю та розвиненістю, надає широкі можливості для реалізації багатокористувацьких ігор. Його система UNet дозволяє легко налаштовувати мережеві функції, і при цьому він залишається досить легким для вивчення для розробників усіх рівнів.

CryEngine надає інструменти для створення багатокористувацьких ігор різного масштабу, забезпечуючи розширені можливості для розробки мережевих аспектів проектів. Проте, варто відзначити, що вивчення та освоєння CryEngine може вимагати більше зусиль в порівнянні з іншими популярними ігровими двигунами.

Кожен ігровий двигун відзначається своєю унікальністю та специфічними перевагами в сфері роботи з мережевими інструментами. При виборі конкретного двигуна важливо враховувати не тільки його технічні можливості, але й спрямованість проекту, рівень експертної команди розробників, а також віддати перевагу засобам, які найбільше відповідають конкретним потребам багатокористувацького проекту. Проаналізувавши усі наявні ігрові двигуни, для розробки та аналізу було обрано ігровий двигун Unreal Engine.

РОЗДІЛ 2
МЕРЕЖЕВА АРХІТЕКТУРА ТА ТЕХНОЛОГІЇ В ОНЛАЙН СИСТЕМАХ

В сучасному цифровому ландшафті, де віртуальний світ і реальність стають все більше взаємопов'язаними, мережева архітектура та протоколи грають невід'ємну роль у створенні потужних та інноваційних онлайн систем. Однак, розуміння та вміння вибрати оптимальні підходи стають вирішальними аспектами для розробників, що працюють над високопродуктивними та масштабованими проектами.

Цей розділ присвячений глибшому розгляданню мережевої архітектури та передових технологій, які дозволяють онлайн системам ефективно взаємодіяти, надійно обмінюватися даними та забезпечувати неперевершений користувацький досвід. Буде розглянуто ключові концепції, які лежать в основі таких архітектур, а також практичний аспект використання різноманітних мережевих технологій у сучасних онлайн системах.

В рамках цього вивчення ми розглянемо такі аспекти, як вибір між Client-Server та Peer-to-Peer архітектурами, використання різних мережевих протоколів, таких як TCP/IP та UDP для передачі даних та забезпечення швидкодії в режимі реального часу, а також розгорнемо технології, як Listen Server та Dedicated Server, які забезпечують миттєву та ефективний обмін інформацією. Проаналізуємо мережеві бібліотеки та фреймворки Unreal Engine. Розглянемо сучасні інструменти, такі як Unreal Networking Architecture, Steamworks Networking, Online Subsystem та інші, які впроваджують новаторські підходи до побудови стабільних та масштабованих мережевих систем.

Важливим аспектом буде аналіз методів компенсації затримки в онлайн шутерах, де розв'язання проблем затримки є ключовим для забезпечення плавної та чесної гри.

Мета цього розділу - зробити поглиблений огляд мережевих аспектів онлайн систем, розкрити їхні можливості та визначити оптимальні практики для забезпечення стабільної та швидкої взаємодії в онлайн середовищі.

2.1 Архітектурні концепції у мережевому програмуванні

В світі сучасних технологій, мережеве програмування визначає шлях для інновацій та зрушень у різних галузях, розуміння архітектурних концепцій стає ключовим елементом успішної розробки та ефективного використання мережевих систем. Цей розділ присвячений дослідженню та аналізу архітектурних принципів у мережевому програмуванні, відкриваючи унікальні можливості та виклики, які стоять перед розробниками під час вибору технології для розробки програмного продукту.

Я розглядав різні концепції, які визначають структуру та взаємодію мережевих систем, їхні переваги та недоліки, а також досліджував, як ці концепції впливають на продуктивність, масштабованість та безпеку програмних рішень. Розділ також розгляне сучасні тенденції у мережевому програмуванні та роль архітектурного підходу у вирішенні складних завдань, що стоять перед розробниками.

Занурюючись у цей розділ, буде розкрито та проаналізовано основні поняття, дозволяючи глибше розуміти та ефективно використовувати архітектурні концепції у власних мережевих програмах.

2.1.1 Архітектура client-server

Архітектура клієнт-сервер є однією з ключових концепцій у мережевому програмуванні, яка визначає структуру взаємодії між різними компонентами системи та використовується в Unreal Engine. У цій архітектурі існує два основних типи взаємодії: клієнтський комп'ютер, який ініціює запити, і сервер, який відповідає на ці запити, надаючи відповіді та необхідні ресурси. У контексті ігрових платформ клієнтами є гравці, які підключаються до централізованого сервера для участі у грі[7].

В однокористувацькій або локально багатокористувацькій грі, система запускається локально в окремій грі в режимі Standalone.

Якість ігрового досвіду усіх гравців не залежить від з'єднання між клієнтом та сервером та не обмежується затримкою гравця з великим пінгом. Це дозволило

гравцям приєднуватися та залишати гру в будь-який момент, а також поліпшувало масштабованість, оскільки використання архітектури клієнт-сервер зменшувало середню пропускну здатність, необхідну для кожного гравця. Багато стратегічних ігор працюють саме так.

На рисунку 2.1. представлено приклад локального з'єднання за допомоги Client-Server моделі, де усі гравці підключені до одного ПК і напряму керують усім у грі, навколишній світ і користувацький інтерфейс та всі результати обробляються на одній машині.

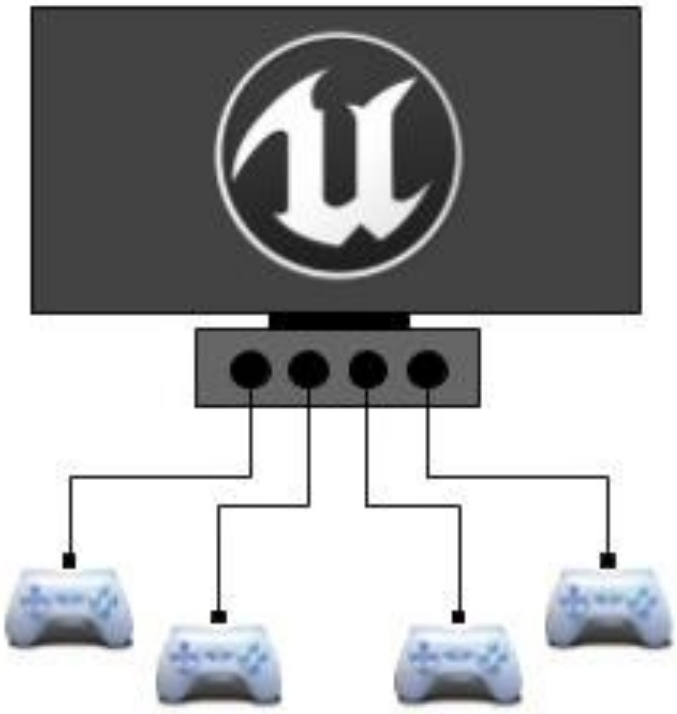

Рис. 2.1. Client-Server Model в локальній системі

Основні принципи архітектури client-server включають наступне:

1. Розподілений доступ до ресурсів: Сервери надають ресурси, які можуть бути загальнодоступними для багатьох клієнтів, такі як файли, бази даних, веб сторінка тощо.

2. Централізована обробка запитів: Сервер відповідає за обробку запитів від клієнтів і надсилає їм відповіді. Це спрощує розподілення функцій та забезпечує керування і обслуговуванням великої кількості клієнтів.
3. Серверне програмне забезпечення: Сервер має спеціалізоване програмне забезпечення для надання певних послуг або доступу до ресурсів.
4. Клієнтські додатки: Клієнти взаємодіють із сервером за допомогою клієнтських додатків, які надають інтерфейс для користувача та передачу запитів до сервера.
5. Комунікація через мережу: Клієнти та сервери обмінюються даними через мережеве з'єднання, яке може бути реалізовано різними способами, включаючи HTTP, TCP/IP, UDP, FTP і інші протоколи.

У мережевій багатокористувацькій грі движок Unreal Engine використовує модель клієнт-сервер. Один комп’ютер у мережі діє як сервер і розміщує сеанс багатокористувацької гри, а ПК інших гравців під’єднуються до сервера як клієнти. Сервер надсилає інформацією про поточний стан гри з кожним гравцем і надає їм засоби для спілкування один з одним. На рисунку 2.2. представлено як відбувається обробка інформації між сервером(1) та підключеними до нього клієнтами(2). Сервер оброблює усі дані, а клієнти показують гру користувачам.

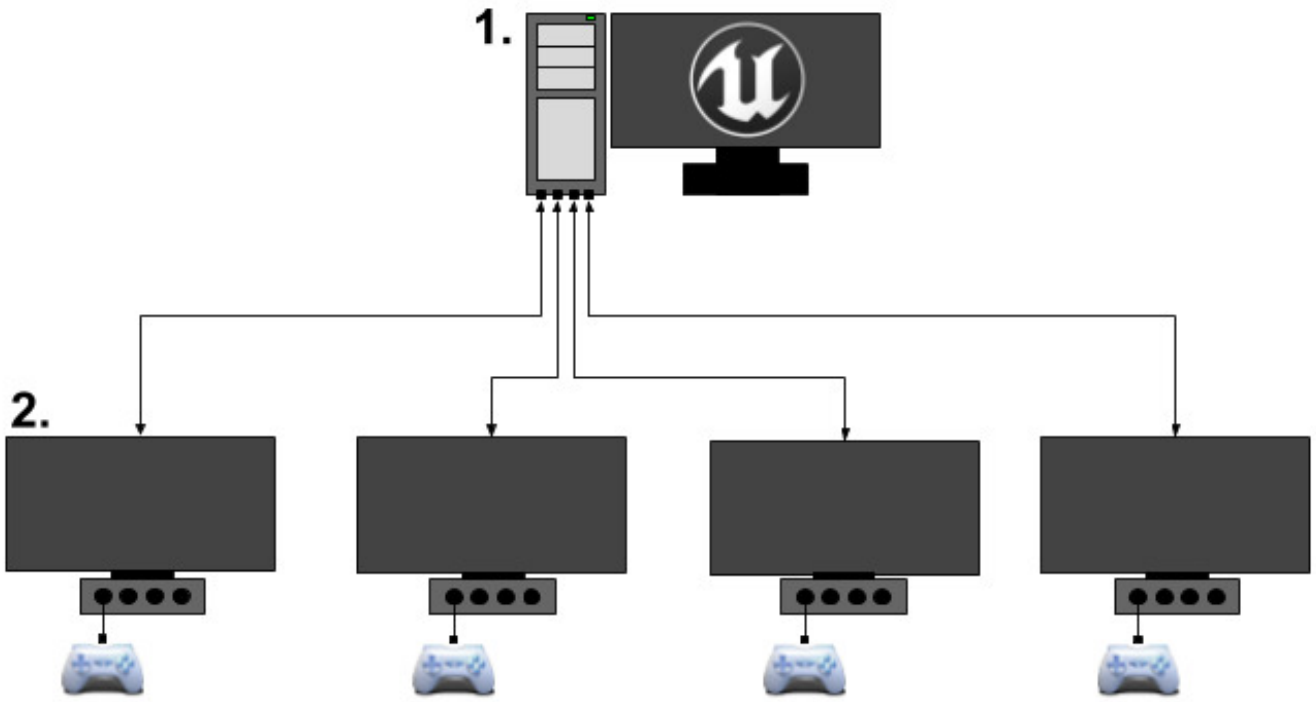

Рис. 2.2. Client-Server Model в багатокористувацькій системі

Можливості Client-Server Model:

1. Синхронізація та контроль: Ця архітектура надає повний контроль над грою і її станом, оскільки сервер відповідає за всі рішення та перевірки.
2. Можливість масштабування: Вона дозволяє легко масштабувати гру для більшої кількості гравців.

Обмеження:

1. Збільшена завантаженість на сервер: Сервер повинен обробляти всі запити від гравців, що може створювати велике навантаження на сервер у великих масштабах.
2. Мережева затримка: Затримка виникає через обмін даними через інтернет мережу, що може впливати на реакцію гравців.
3. Нечесна гра та шахрайські програми: сервер може бути використаний для зловживань через його вразливості. Існує потенціал для використання шахрайських програм, що може викликати дисбаланс у грі.

2.1.2 Архітектура peer-to-peer

Коли багатокористувацькі онлайн-ігри тільки починали розвиватися, ігри покладалися на налаштування мережі, відомої як peer-to-peer. У peer-to-peer архітектурі всі гравці взаємодіють один з одним без центрального сервера. Пристрій кожного гравця діє як клієнт і сервер, що дозволяє гравцям взаємодіяти один з одним у режимі реального часу[8]. Кожен гравець може відправляти та отримувати дані від інших гравців, і вони спільно визначають стан гри. Поняття peer-to-peer було запропоноване у 1984 році під час розробки архітектури Advanced Peer to Peer Networking у компанії IBM. P2P мережі можуть бути використані для файлообміну, безпосереднього обміну даними між користувачами, розподілених обчислень та інших застосувань. Ця архітектура дозволяє зменшити навантаження на централізовані сервери та підвищити надійність системи завдяки розподіленому

характеру обміну даними. На рисунку 2.3. зображено взаємодію peer-to-peer архітектури в мережевих іграх.

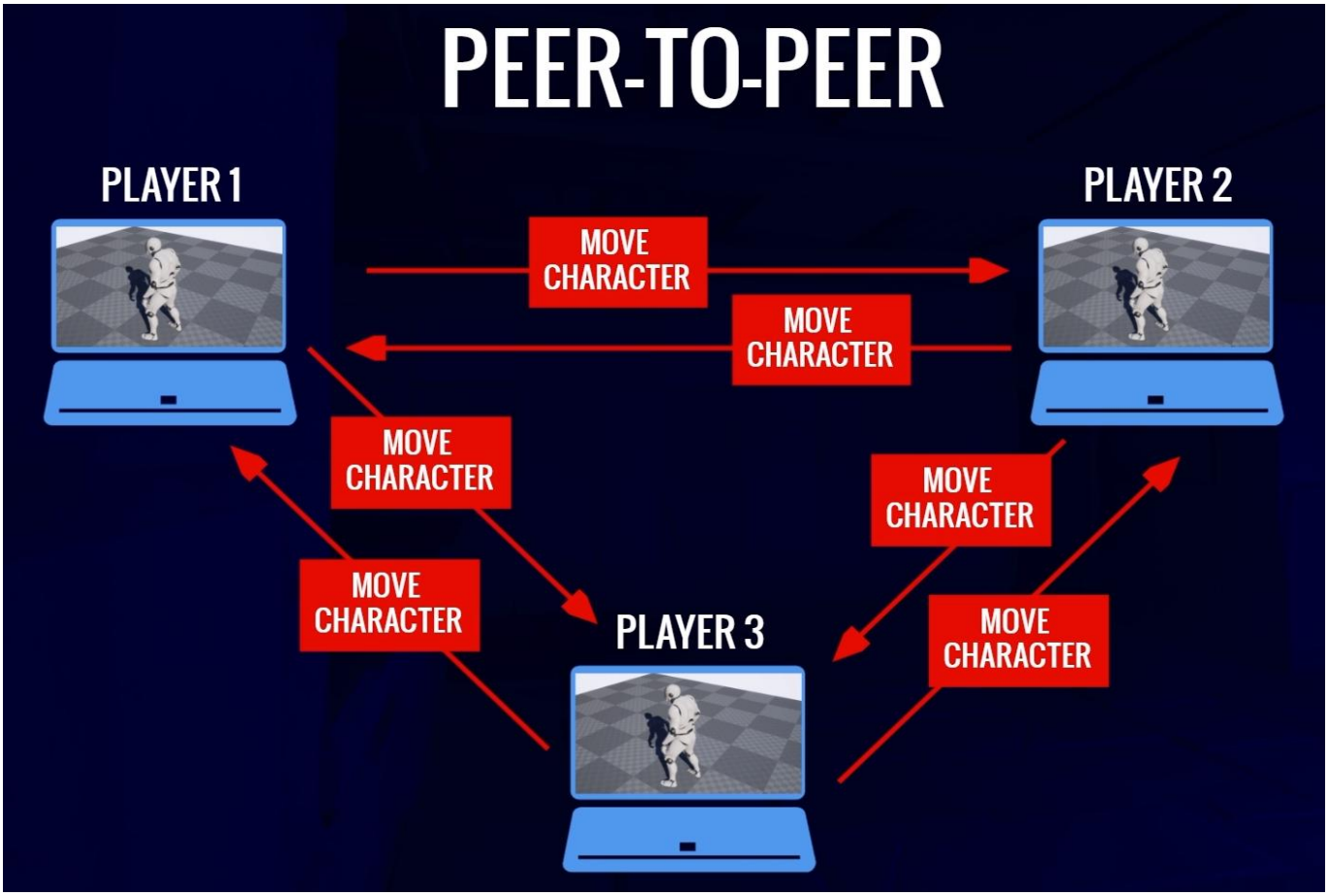

Рис. 2.3. Архітектура Peer-to-peer

Можливості:

1. Менший навантаження на сервер: У порівнянні з архітектурою client-server, ця архітектура може зменшити навантаження на сервер, оскільки гравці взаємодіють безпосередньо один з одним.
2. Можливість підтримувати режим офлайн: Гравці можуть взаємодіяти один з одним, навіть якщо сервер вимкнено або втратив зв'язок.

Обмеження:

1. Синхронізація зі складнішою реалізацією: Синхронізація між гравцями у peer-to-peer системі може бути складнішою, особливо в умовах обмеженого зв'язку.
2. Безпека: Передача даних між вузлами може бути менш безпечною, особливо якщо відсутні централізовані точки контролю.

2.1.3 Архітектура Listen Server

Архітектура Listen Server є однією з архітектур для мультиплеєрних ігор в ігровому движку Unreal Engine. У цій архітектурі один з гравців, який фізично грає в гру, виступає як сервер і, в той же час, як клієнт. Такий гравець володіє "слухачем" (Listen Server), який відповідає за обробку і синхронізацію подій між гравцями. Також якщо цей клієнт відключається, то сервер закривається.

Головний гравець відповідає за синхронізацію гри між гравцями, що може призводити до затримок або перебоїв у грі. Listen Server часто використовується в невеликих іграх з невеликою кількістю гравців, де сервер не потребує значних обчислювальних ресурсів для обслуговування клієнтів[8].

Основні риси архітектури Listen Server в Unreal Engine включають наступне:

1. Серверна роль гравця: Один із гравців вибирається для виконання ролі сервера (Listen Server). Цей гравець відповідає за управління грою, симуляцію гравців і надсилання оновлень до інших клієнтів.
2. Клієнтські ролі: Решта гравців виступають у ролі клієнтів, які підключаються до сервера Listen Server. Вони відправляють свої дії серверу та отримують оновлення від нього.
3. Реплікація даних: Unreal Engine використовує механізми реплікації для синхронізації даних між сервером і клієнтами. Це включає в себе рух гравців, стани об'єктів, події і багато іншого.
4. Оскільки головний гравець є і клієнтом, то він потребує UI, має PlayerController, що і є частиною клієнта.
5. Remote Procedure Call (RPC): RPC має велике значення у роботі Listen Server. Виклики віддалених процедур дозволяють клієнтам викликати функції на сервері або навпаки. Це допомагає синхронізувати ігрові дії і події.

Переваги:

1. Простота налаштування та розробки. Архітектура Listen Server легко налаштовується та використовується для створення багатокористувацьких ігор без необхідності великої серверної інфраструктури.
2. Відсутність необхідності в окремому серверному обладнанні.
3. Підтримка локальної гри: Гравці можуть взаємодіяти один з одним як локально, так і через мережу. Це дозволяє організовувати локальні групи або великі онлайн-битви.
4. Доступ до сервера через IP-адресу: Інші клієнти можуть приєднатися до сервера, використовуючи IP-адресу гравця-сервера.

Недоліки:

1. Обмежені можливості масштабування та продуктивності для більших ігор.
2. Обмеження швидкості з'єднання сервера: Якщо швидкість з'єднання сервера недостатня, це може призвести до проблем з лагами і незручностями для інших гравців.
3. Можливість шахрайства: У деяких випадках, серверний гравець може бути шахраєм і впливати на гру в свою користь або незаконно змінювати гру, що може завдати шкоди іншим гравцям.
4. Залежність від засновника сервера: Гра завжди залежить від серверного гравця. Якщо серверний гравець вирішить завершити гру або вийти з неї, гра завершиться для всіх гравців.
5. Навантаження серверного гравця: Гравець, який виступає як сервер (Listen Server), має велике навантаження. Він відповідає як за обробку власних дій, так і за симуляцію інших гравців. Це може вплинути на його можливість грати комфортно, особливо в іграх з великою кількістю гравців або іграх з великим обсягом обчислень.

2.1.4 Архітектура Dedicated Server:

Dedicated Server - це окремий сервер, який відповідає за обробку мережевих запитів та синхронізацію гри між гравцями. В Unreal Engine має назву Standalone

server. У цій архітектурі немає гравця-господаря, і сервер призначений виключно для обслуговування гри. Dedicated Server використовується для обробки всіх мережевих запитів, тому він може працювати на великому обладнанні і мати більше ресурсів для оптимізації продуктивності гри[9]. Найчастіше використовується у великих масштабних багатокористувацьких іграх, де потрібна висока продуктивність та масштабованість. Порівняння Listen Server та Dedicated Server графічно показано на рисунку 2.4.

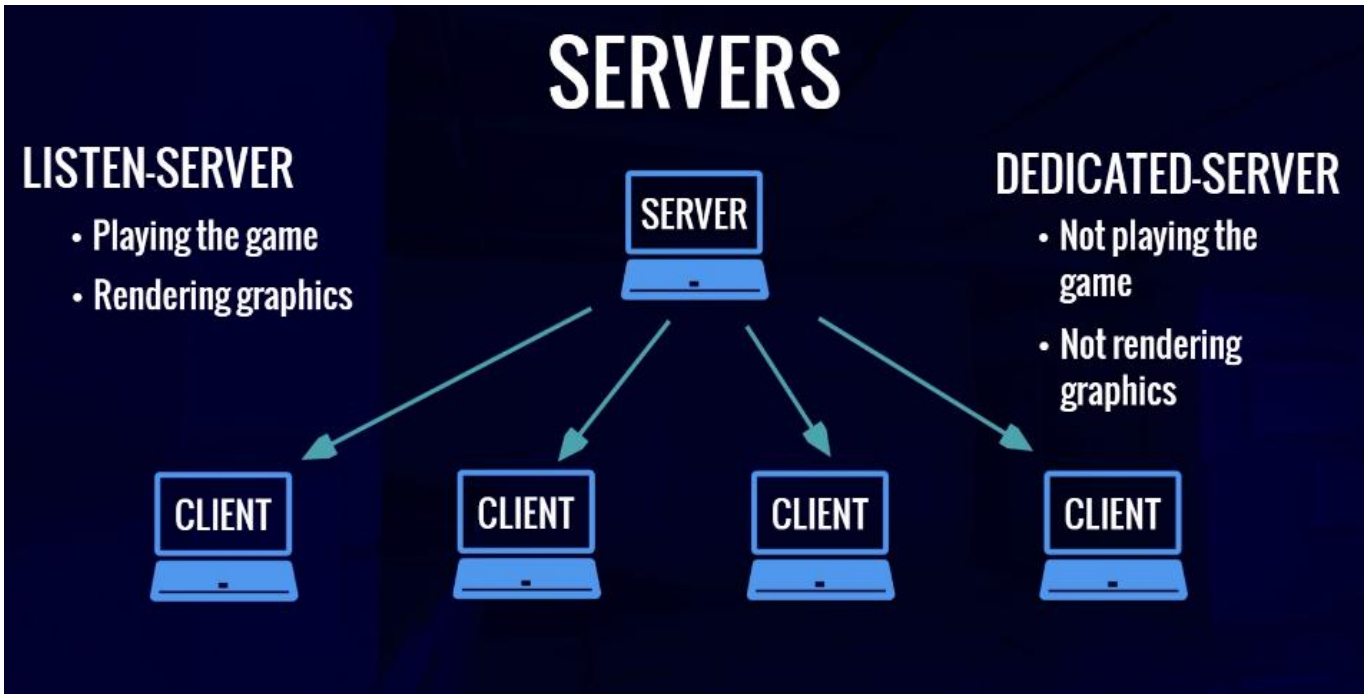

Рис. 2.4. Порівняння Listen Server та Dedicated Server

Переваги:

1. Висока продуктивність та стабільність гри для багатьох гравців.
2. Можливість масштабування для великих ігор.
3. Може бути скомпільованим під Windows або Linux
4. Не має візуального представлення та власного персонажа, тож вони не мають UI чи свого власного PlayerController і також не має власного персонажа.

Недоліки:

1 Складніше налаштування та управління сервером.

2 Вимагає окремого серверного обладнання та більшої інфраструктури.

Вибір між Listen Server і Dedicated Server залежить від конкретних потреб. Якщо необхідно створити масштабну багатокористувацьку гру, то Dedicated Server може бути кращим варіантом для забезпечення високої продуктивності та якості гри. У менших проектах Listen Server може бути більш зручним і простим у розробці.

2.2 Мережеві бібліотеки та фреймворки Unreal Engine

В умовах постійного розвитку індустрії відеоігор та зростання популярності онлайн-ігор, важливою складовою стає не лише якість геймплею та графічного виконання, але і високопродуктивна та стабільна мережева архітектура. Проблеми ефективної мережевої взаємодії стають ключовими в арсеналі розробників. Unreal Engine, як передовий двигун, не лише визначає нові стандарти у галузі графічної та фізичної реалізації, але і надає розробникам потужний інструментарій для створення масштабованих та високопродуктивних онлайн ігор.

Під час багатокористувацької сесії ігровий стан передається між багатьма клієнтами через інтернет. У випадку однокористувацької гри вся інформація зберігається на одній машині. Як наслідок розробка багатокористувацької гри є більш складною, процес обміну інформацією між гравцями є дуже делікатним та додає кілька додаткових кроків. UE має надійну мережеву структуру, що є основою для багатьох популярних онлайн-ігор.

В Unreal Engine існують різні мережеві протоколи та технології, які використовуються для реалізації функціонала багатокористувацьких ігор. Ця різноманітність інструментів дозволяє розробникам створювати ігри з різними типами мережевої архітектури та функціоналу в залежності від конкретних потреб і цілей гри.

Розділ "Мережеві бібліотеки та фреймворки Unreal Engine" у цій дипломній роботі присвячений вивченню важливого аспекту розробки ігор, а саме використанню різноманітних бібліотек та фреймворків для забезпечення ефективної мережевої архітектури.

Мета даного розділу – глибокий аналіз та розкриття можливостей кожного інструменту UE в контексті створення стабільних та інтерактивних онлайн ігор, а

також їхній вплив на загальну динаміку гри та задоволення гравця. Детальний розгляд підрозділів цього розділу, таких як:

- Unreal Networking Architecture,
- Steamworks Networking,
- Online Subsystem,
- Custom Networking Solutions,
- Voice Over IP бібліотеки,
- Epic Online Services (EOS).

В кожному з них я розгляну технічні характеристики, переваги та обмеження, а також їх приклади успішного використання в ігрових проектах.

Дослідження цього розділу буде важливим внеском у розуміння сучасних вимог мережевого функціонала в онлайн іграх. Головною метою буде повний огляд мережевих засобів Unreal Engine, який служитиме підґрунтям для подальших розділів дипломної роботи та дозволить зробити обґрунтовані висновки щодо їхнього застосування у реальних проектах розробки ігор.

2.2.1 Unreal Networking Architecture

Unreal Networking Architecture - це архітектурна система для роботи з мережею в ігровому движку Unreal Engine, розроблена компанією Epic Games. Створена архітектура надає інструменти та можливості для розробки багатокористувацьких ігор, де кілька гравців може спільно грати в одній грі через інтернет або локальну мережу. Unreal Networking Architecture забезпечує синхронізацію гри між сервером і клієнтами, обробку даних та подій, а також керування мережевлю взаємодією.

UNet є основним вбудованим мережевим рішенням у Unreal Engine. Він використовує UDP (User Datagram Protocol) для обміну даними між гравцями та сервером. UNet дозволяє створювати багатокористувацькі ігри з високою швидкістю передачі даних та реалізовувати функціональність синхронізації дій гравців.

Приклади використання:

1) Шутери: UNet ідеально підходить для шутерів, де швидкість та низька затримка є важливими.

2) Гонки: В перегонах також важлива низька затримка та швидкість мережі для синхронізації гравців та транспортних засобів.

Оскільки була проаналізована клієнт-серверна архітектура[10]. Ми розглянемо розділення самого фреймворку, а ділиться він на 4 частини:

- Server only - означає, що об’єкти існують виключно на сервері;
- Server & Clients - означає, що об’єкти існують на сервері та всіх клієнтах;
- Server & Owning Client - об’єкт існує на сервері та клієнті, що володіє ним;
- Owning Client Only - ці об’єкти існують лише на клієнті.

2.2.2 Steamworks Networking

Steamworks Networking - це мережева технологія, яка інтегрована з платформою Steam для гри. Вона використовує Steam Datagram Relay (SDR) для обміну даними між гравцями, що спрощує підключення до Steam-ігор та надає функціонал для голосового чату, зберігання рекордів тощо.

Steamworks Networking надає багаторівневий інтерфейс для налаштування мережевого взаємодії у грі та підтримує різні мережеві моделі, включаючи peer-to-peer та сервер-клієнт[11].

Щоб додати Steamworks Networking до Unreal Engine проекту необхідно увімкнути плагін Steam Sockets. На рисунку 2.5. графічно показано як увімкнути плагін в движку UE.

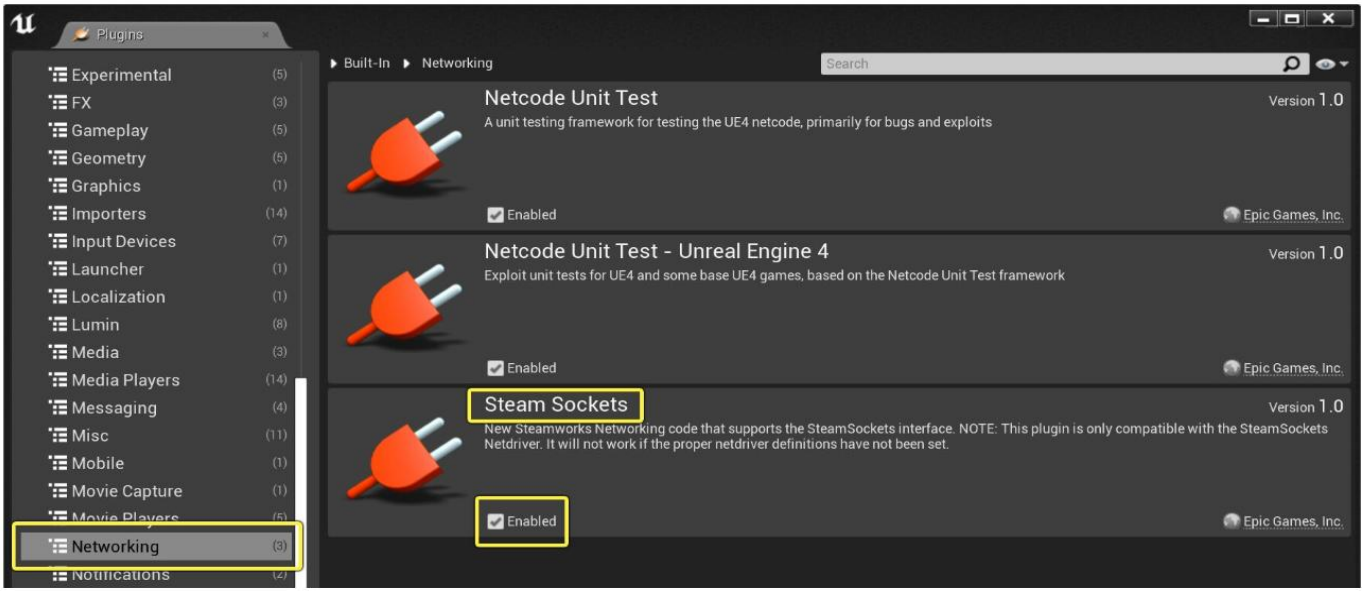

Рис. 2.5. Плагін Steam Sockets

Приклади використання:

1) Steamworks Networking особливо корисна для ігор, що випускаються на платформі Steam та вимагають інтеграції зі Steam-сервісами.

2.2.3 Online Subsystem

Online Subsystem в Unreal Engine - це фреймворк та система для обробки різноманітних функцій онлайн та багатокористувацьких можливостей в іграх. Він надає абстрактний інтерфейс для взаємодії з різними онлайн-сервісами, платформами та системами матчмейкінгу, що дозволяє розробникам інтегрувати онлайн-функції в свої проекти[12]. Online Subsystem діє як міст між грою та конкретним онлайн-сервісом або платформою, дозволяючи забезпечити безперервний мультиплеєр та онлайн-взаємодію, на рис. 2.6.

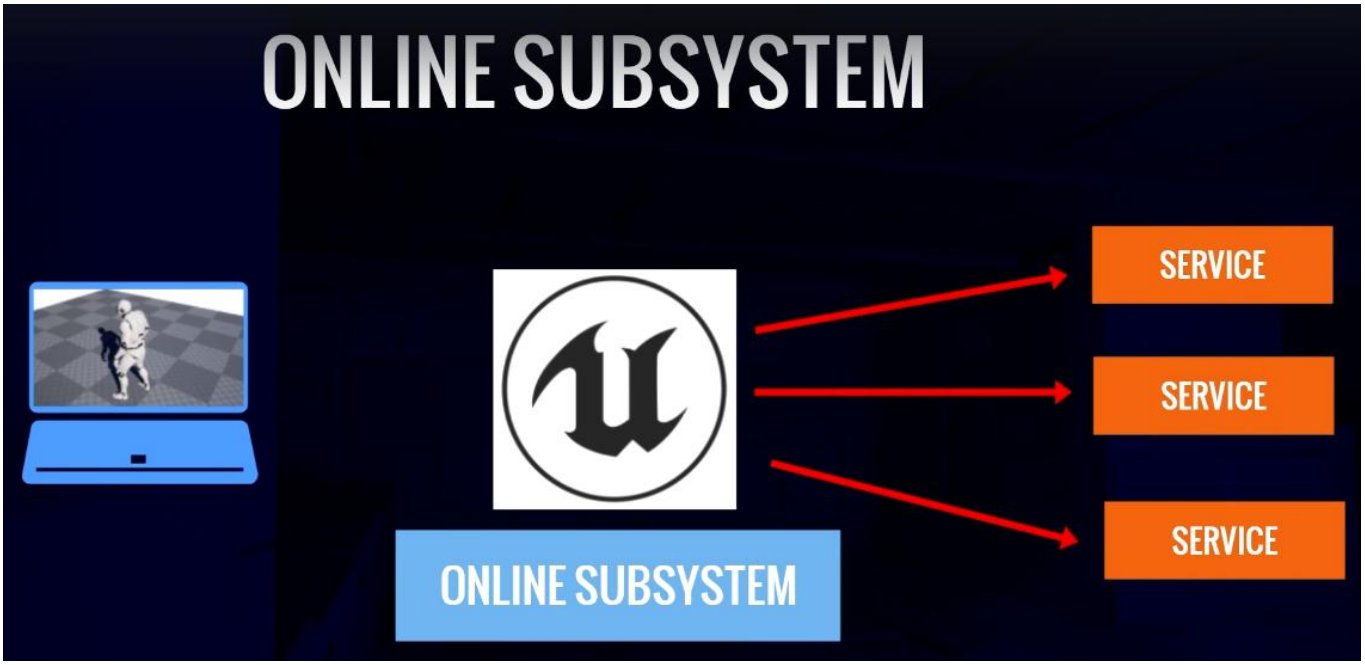

Рис. 2.6. Online Subsystem

Основні компоненти та функції Online Subsystem у Unreal Engine включають:

1. Абстракційний шар: Online Subsystem надає абстрактний інтерфейс, який дозволяє розробникам взаємодіяти з різними онлайн-сервісами без необхідності безпосередньо взаємодіяти з їхніми API. Це полегшує інтеграцію різних онлайн-послуг у гру.
2. Підтримка платформ: Online Subsystem підтримує різні платформи, такі як Steam, PlayStation Network, Xbox Live, і багато інших. Розробники можуть вибрати відповідний Online Subsystem для цільової платформи гри.
3. Матчмейкінг і лобі: В Online Subsystem є функціональність для створення лобі гри, пошуку гравців та матчмейкінгу для організації онлайн-партій.
4. Друзі та інвайти: Розробники можуть використовувати Online Subsystem для реалізації функціональності додавання друзів, надсилання запрошень та ігор з друзями.
5. Статистика та досягнення: Online Subsystem дозволяє збирати статистику гри та відстежувати досягнення гравців у мережевому середовищі.

Приклади використання:

1) Багатокористувацькі ігри: технологія корисна, для розробки гри, яка підтримує гравців з різних платформ , такі як PlayStation Network, Xbox Live, або використовує публічні сервери для гри.

На рисунку 2.7. зображено список наявних плагінів середовища Online Subsystem.

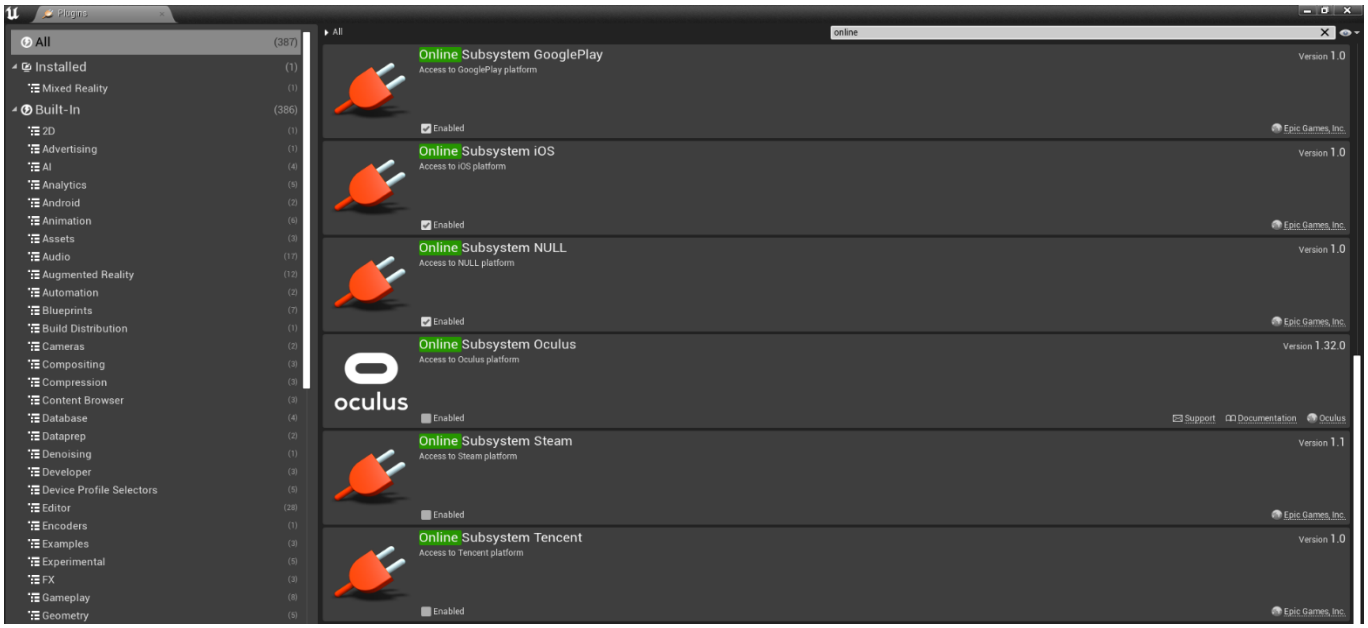

Рис. 2.7. Online Subsystem плагіни

Ось невеличкий список доступних інтерфейсів:

- AchievementsInterface;
- ChatInterface;
- EntitlementsInterface.

2.2.4 Custom Networking Solutions

Індивідуальні мережеві рішення передбачають адаптацію мережевого обладнання та програмного забезпечення до конкретних потреб. Unreal Engine надає можливість розробникам створювати власні мережеві рішення на основі різних протоколів (наприклад, TCP/IP або UDP). Це дозволяє досягти повного контролю над мережевими аспектами гри для задоволення специфічних потреб. Власні

мережеві рішення корисні для проектів зі складними або специфічними вимогами, де необхідно максимально налаштувати мережеву систему.

Вибір конкретного протоколу залежить від типу гри, її потреб та інфраструктури. На рисунку 2.8. зображено порівняльну характеристику роботи протоколів TCP та UDP[13].

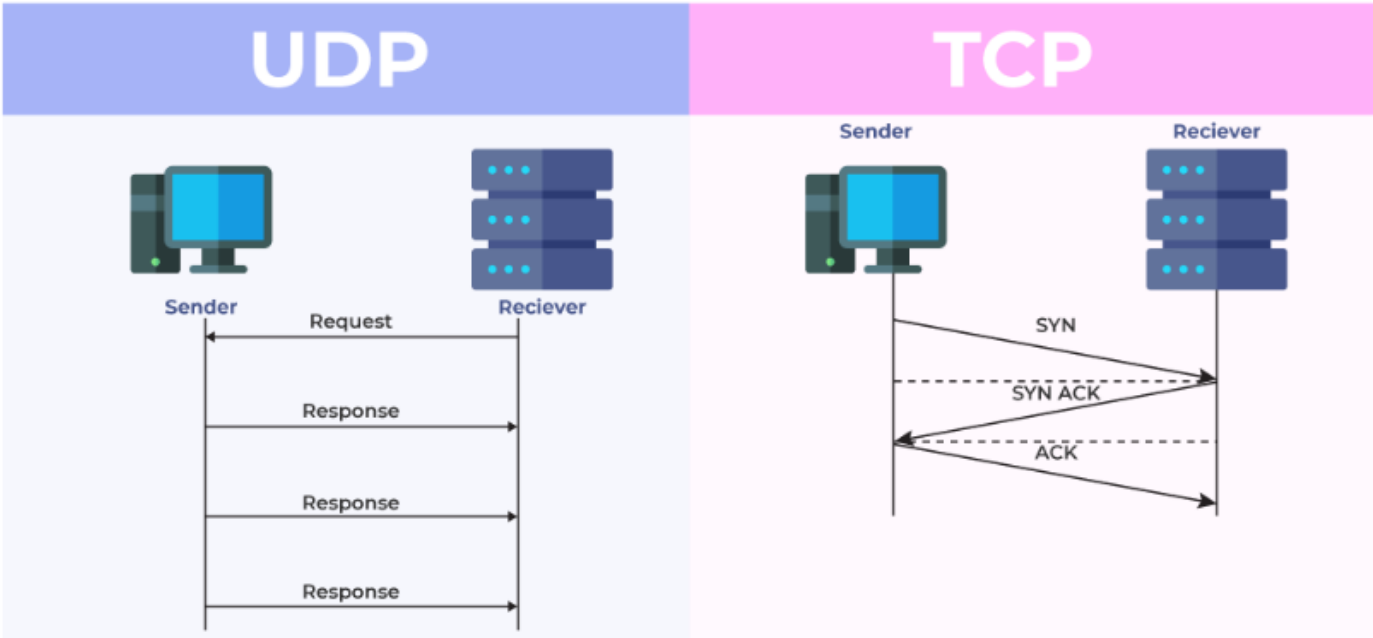

Рис. 2.8. Характеристику роботи протоколів TCP та UDP

UDP використовується для швидкого обміну даними без гарантії надійності чи порядку приходу. Він особливо підходить для ігор, де важливо мінімізувати затримки ідеально підходить для швидких дійових ігор, таких як шутери чи перегони.

TCP використовується для надійного обміну даними, забезпечує гарантію доставлення та порядок приходу. Він добре підходить для стратегічних ігор або ігор, де точність та надійність важливі.

2.2.5 Voice Over IP бібліотеки

Voice over IP в Unreal Engine надає можливість вбудовувати голосовий зв'язок в ігри або додатки, розроблені за допомогою цього ігрового движка. Unreal Engine надає інструменти для інтеграції голосового чату та комунікації в реальному часі між гравцями через мережу. Платформа підтримує різні системи VoIP, а також API

для створення власних рішень голосового зв'язку[14]. Плагін EOS Voice Chat реалізує інтерфейси IVoiceChat та IVoiceChatUser, на рис. 2.9.

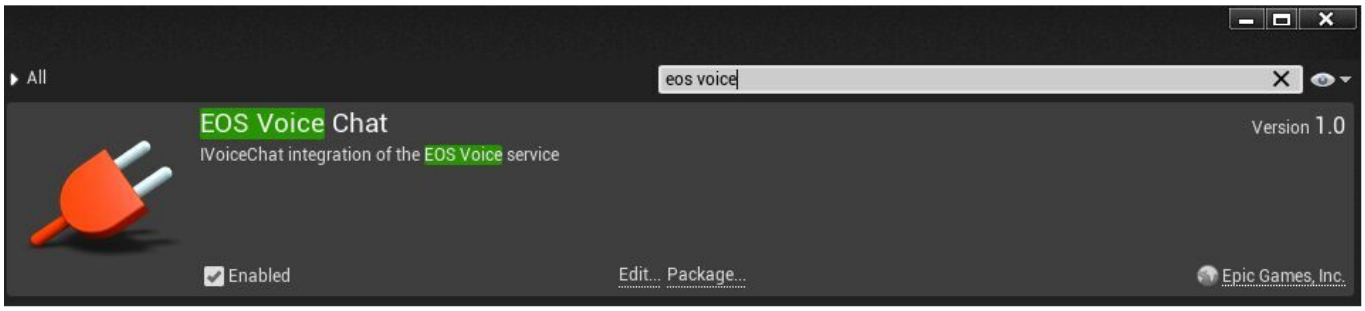

Рис. 2.9. EOS Voice Chat

Можливості VoIP в Unreal Engine дозволяють створювати ігри та додатки з такими функціями, як:

1. Голосовий чат: Гравці можуть спілкуватися один з одним через голосовий чат під час гри.
2. Конференц-дзвінки: Декілька гравців можуть об'єднатися в конференц-дзвінок для спільної бесіди.
3. Відеозв'язок: Підтримка відеозв'язку для спілкування з використанням веб-камер.
4. Запис голосу: Можливість запису голосового контенту для подальшого відтворення або збереження.
5. Інтеграція з мережевими іграми: Можливість використовувати VoIP для спілкування між гравцями у багатокористувацьких іграх.
6. Можливість встановити власний VoIP сервер або використовувати сторонні сервіси.

Для реалізації VoIP в Unreal Engine розробники можуть використовувати різні VoIP бібліотеки або API, включаючи сторонні рішення. Unreal Engine забезпечує зручний інтерфейс для інтеграції голосового зв'язку, що дозволяє створити інтерактивні ігри та додатки з високоякісною голосовою взаємодією для користувачів.

2.2.6 Epic Online Services (EOS)

Epic Online Services (EOS) - це набір інструментів та сервісів, розроблений компанією Epic Games, який призначений для розробки багатокористувацьких ігор та додатків. EOS дозволяє розробникам інтегрувати різні онлайн-функції у свої проекти та створювати багатокористувацькі ігри, що працюють на різних платформах[15].

EOS робить процес створення багатокористувацьких ігор більш простим та дозволяє розробникам швидко створювати ігри з розширеним онлайн-функціоналом. Він підтримує голосовий та текстовий чат, матчмейкінг, зберігання даних гравців, інтеграцію з соціальними мережами, захист від шахраїв та багато інших функцій, які допомагають утримати гравців у грі. На рисунку 2.10. представлено основні переваги Epic Online Services.

Рис. 2.10. Epic Online Services (EOS)

2.3 Методи компенсації затримки в онлайн шутерах

Lag Compensation - це важлива технологія в онлайн-іграх, особливо в онлайн-шутерах, де час реакції гравця і точність стрільби важливі для успіху в грі. Основна

мета компенсації затримки полягає в тому, щоб зробити гру більш справедливою для гравців із різними показниками затримки до сервера.

Ping - це термін, який загально використовується для опису програми, призначеної для відправлення сигналу користувача до певного призначення через мережу та отримання відповіді. Він може бути використаний для тестування, чи існує задана IP-адреса і чи може приймати запити.

Ping також використовується для діагностики. Він може бути використаний, щоб переконатися, що хост-комп'ютер, який намагається досягти користувача, працює.

Пінг дуже важливий у багатокористувацьких онлайн іграх. Цей термін застосовують, щоб описати, скільки часу потрібно серверу для відповіді на вхідні дані клієнта, на рис. 2.11.

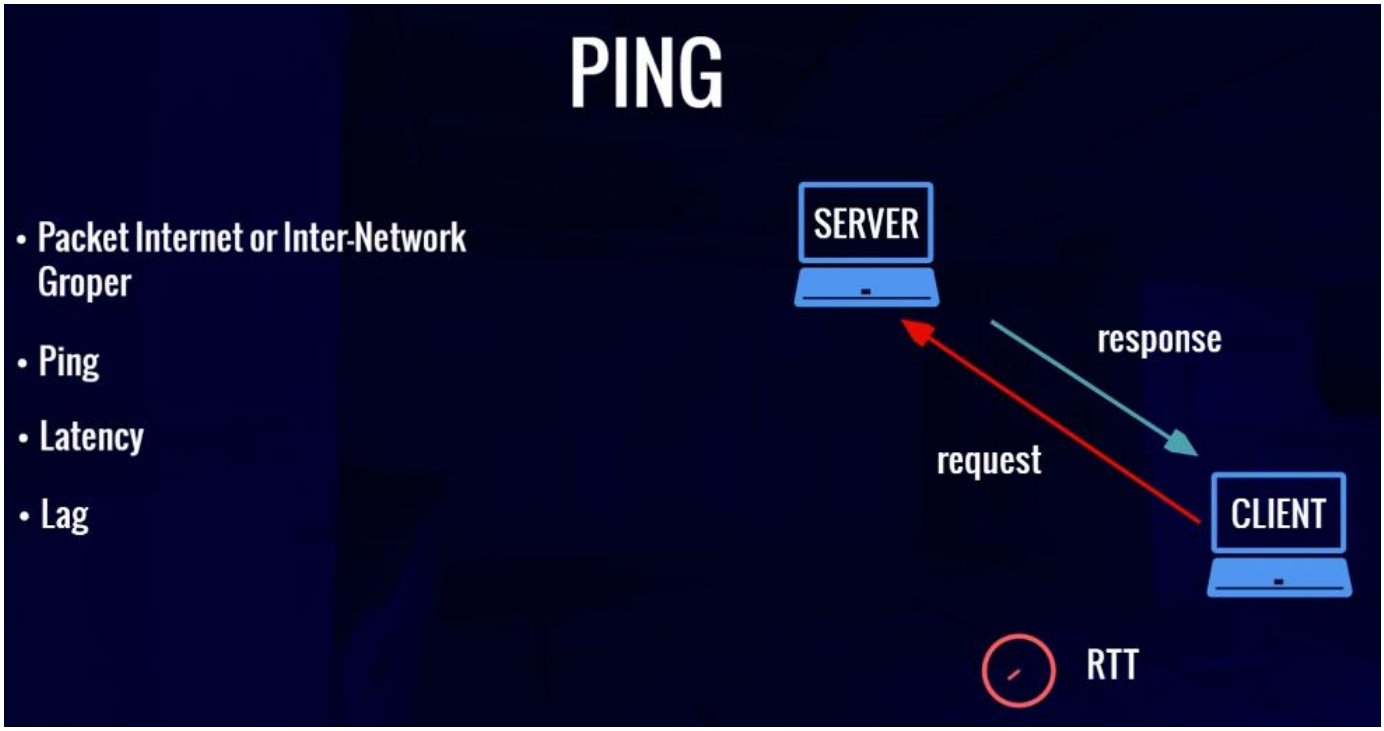

Рис. 2.11. Складова Ping

Термін пінг використовують для опису тестів швидкості. Його використовують, щоб визначити, як швидко дані подорожують від одного місця до іншого. І цей час можливо використовувати для розв'язання проблем з підключенням.

RTT (Round-Trip Time) в іграх - це час, який потрібен для передачі сигналу від клієнта до сервера і назад. RTT вимірює час, який затримується сигнал на шляху від клієнта до сервера і від сервера до клієнта. RTT є важливим показником для оцінки затримки в багатокористувацьких іграх[16].

RTT включає в себе час, потрібний для відправлення сигналу від клієнта до сервера, обробки сигналу на сервері та відправлення відповіді назад до клієнта. Однак цей показник може коливатися і залежить від різних факторів, включаючи якість мережевого підключення, фізичну відстань між клієнтом і сервером, завантаженість сервера та інші чинники, на рис. 2.12.

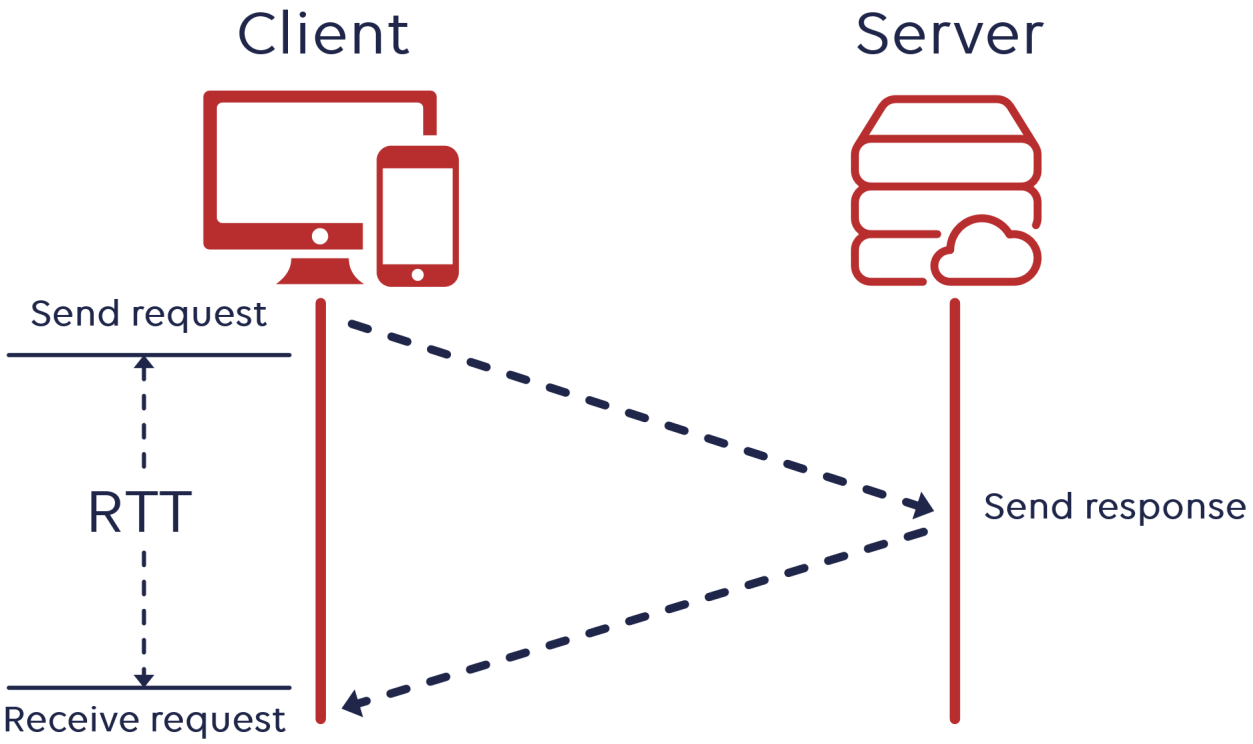

Рис. 2.12. 1Принцип роботи RTT

У багатокористувацьких іграх RTT має велике значення, оскільки він впливає на час реакції гравців, швидкість взаємодії з грою та загальний досвід гри. Гравці можуть відчувати затримку, якщо RTT великий, що може призвести до несправедливих ситуацій та незручностей під час гри.

RTT є важливим показником для оцінки якості мережевого зв'язку в багатокористувацьких іграх і враховувався при розробці механізмів компенсації затримки та інших рішень для покращення якості гри[17].

У іграх, де гравець-господар є сервером, у засновника сервера пінг дорівнює нулю, оскільки серверу не потрібно відправляти дані через мережу самому собі. Якщо у всіх клієнтів є швидкий інтернет-зв'язок, їхній пінг зменшується, і всі отримують позитивний досвід гри. Чудово, якщо усі б мали швидке з'єднання, і дані надсилалися б через мережу миттєво, і ніхто не відчував би ніякої затримки, і гра була б швидкою та збалансованою для усіх.

В реальному світі безмежно швидких інтернет-з'єднань не існує, і розробники ігор не можуть припускати, що у всіх гравців буде гарний інтернет-зв'язок. Це причина, чому міжнародна ігрова платформа Steam має регіони. Гравець обирає регіон у котрому знаходиться. Steam може намагатися підключити користувача до серверів у обраному регіоні або до найближчих. Де гравець може грати з іншими гравцями, підключеними до серверів у цьому регіоні. Головна проблема, що у всіх клієнтів завжди є пінг, і, таким чином, вони завжди будуть відчувати певну затримку.

Деякі типи ігор працюють добре в умовах високих пінгів. Ходові ігри, такі як шахи, майже не справляють впливу на пінг. Те, скільки часу опонент витрачає на вибір свого наступного ходу, практично не впливає на досвід гри. Однак, в багатокористувацьких шутерах високий пінг помітний миттєво і навіть може призвести до того, що гра стає не комфортною. Навіть невеликий лаг може зіпсувати досвід гри в шутерах з високим темпом. Гравці часто помічають різницю при затримці вище 50 мілісекунд, але пінги в 50 мілісекунд досить поширені.

Техніки різняться від гри до гри, залежно від дизайну самої гри та рішень, прийнятих командою розробників. Кожна техніка має свої переваги та недоліки, і команда в кінцевому підсумку вирішує, де робити компроміс. Іноді для пом'якшення затримки гравця використовують інтерполяцію.

Інтерполяція - це техніка в обчисленнях та відеоіграх, яка використовується для створення плавних переходів між двома значеннями або станами в часі. Вона

дозволяє відобразити проміжні значення між двома відомими значеннями, щоб створити плавну анімацію, перехід або рух.

Одним з мінусів інтерполяції є затримка руху. Інтерполяція завжди вносить затримку у рух об'єктів. Це означає, що позиція об'єкта на екрані може не відповідати його фактичній позиції на сервері. Це може вплинути на точність геймплею, особливо в інтенсивних онлайн шутерах. Наприклад, персонаж рухається від точки А до точки В. Тепер, якщо позиція руху користувача реплікувалися з сервера на клієнти, клієнти можуть зберігати поточне оновлення, а також попереднє оновлення. І між цими оновленнями вони можуть інтерполювати між ними. На рисунку 2.13. показана інтерполяція з двома гравцями з різною затримкою під час гри: лівий гравець з 20 мс затримкою бачить та вражає правого, а правий гравець з 400 мс затримкою не бачить лівого та отримує влучання від нього[18].

Рис. 2.13. Приклад роботи інтерполяції з різним пінгом

Під час відображення гри клієнт використовує ці попередні дані про рух для плавної анімації руху об'єкта між двома оновленнями від сервера, але його розташування завжди залишається в минулому, на рис. 2.14.

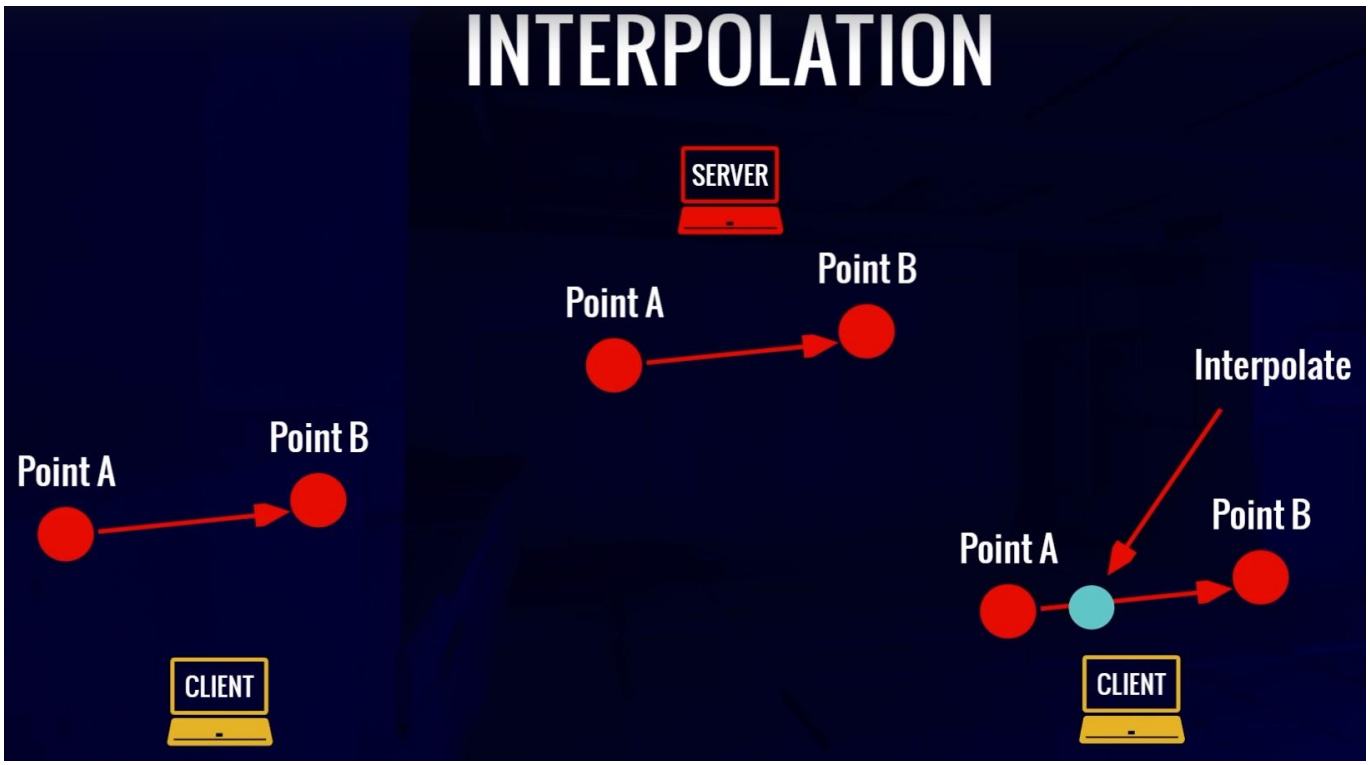

Рис. 2.14. Приклад роботи Інтерполяції

Інтерполяція визначає проміжні значення між двома оновленнями на основі часу і збережених даних про рух. Наприклад, якщо між оновленнями гравець переміщувався з точки A в точку B, інтерполяція обчислить його проміжне положення у певний момент часу між цими оновленнями, і ця затримка є негативною для шутера, оскільки стріляючи в опонента, розташування якого було в минулому на 50 мілісекунд раніше, гравець промахнеться.

Інтерполяція вимагає додаткових обчислень та ресурсів для плавного руху об'єктів. Це може становити навантаження на сервер та клієнти гри. Інтерполяція створює потенційну вразливість до чесного обману або шахрайства в грі, оскільки гравці можуть намагатися змінювати дані руху для власної вигоди.

Екстраполяція - це техніка в обчисленнях і статистиці, що використовується для прогнозування майбутніх значень на основі даних та тенденцій з минулого. У відеоіграх, особливо в мережевих іграх, екстраполяція використовується для передбачення майбутнього руху гравців або об'єктів на основі їх попередніх дій та руху, на рис. 2.15.

Наприклад, якщо гравець рухається вперед і змінює свій напрямок руху, екстраполяція може передбачити, де він буде рухатися в найближчому майбутньому на основі його попереднього руху та вектору швидкості. Це допомагає зробити рух гравців більш плавним та природним у мережевих іграх, де є затримка між сервером і клієнтом.

У відеоіграх екстраполяція може використовуватися для передбачення руху гравців, керованих штучним інтелектом та для розрахунку траєкторії снарядів або для інших аспектів геймплею, де важливий прогноз майбутнього стану об'єктів або сутностей.

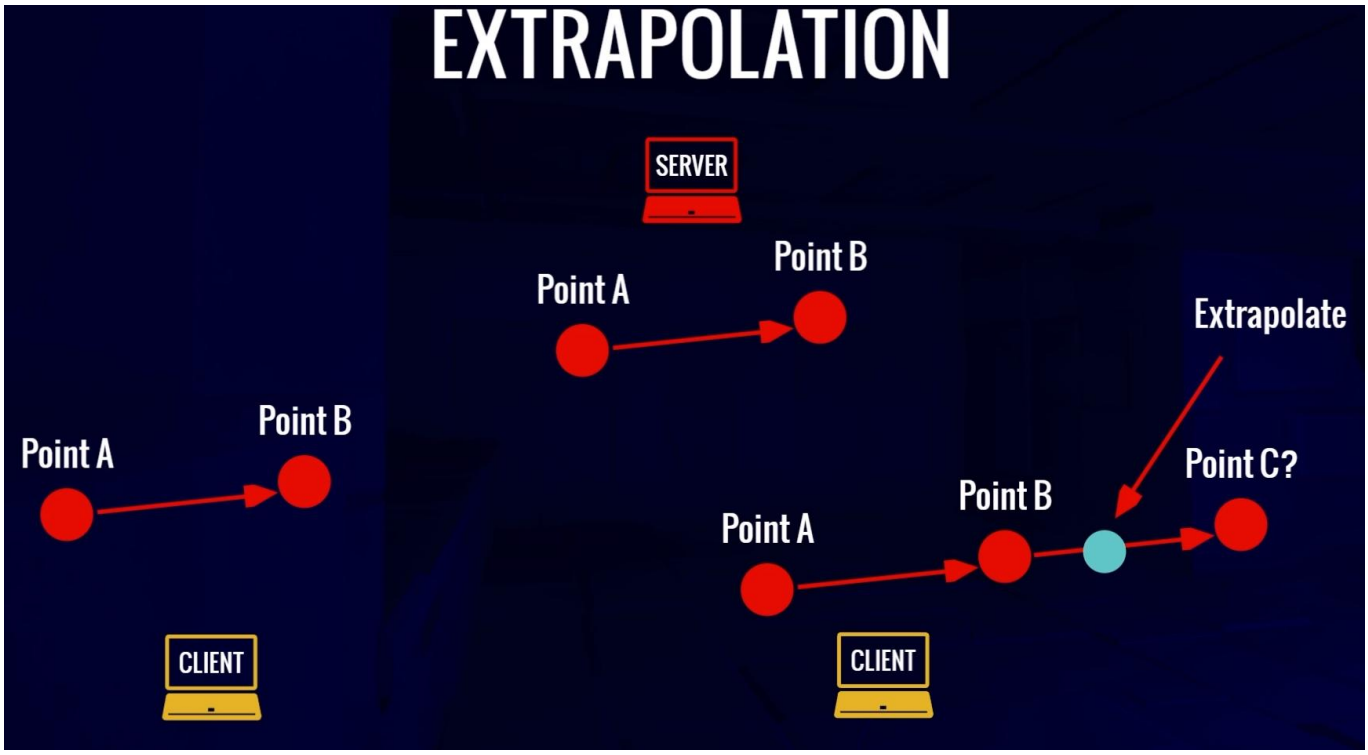

Рис. 2.15. Приклад роботи Екстраполяції

Якщо опонент не змінює курсу, ця система працює ідеально, підтримуючи ту саму швидкість. Екстраполяція точно передбачає рух опонента. В шутері це може бути проблемою, оскільки гравці рідко рухаються в одному напрямку тривалий час. Таким чином, рухаючи опонента вперед, коли він дійсно рухається назад, призводить до навіть більшої неточності, ніж це було б відсутності екстраполяції. В Unreal Engine компонент руху персонажа використовує комбінацію цих методів, і це

відбувається дуже ефективно. Якщо ваш пінг дуже великий, ви можете рухати свого персонажа, і компонент руху персонажа буде використовувати вашу швидкість для екстраполяції вашого положення на інших комп’ютерах, щоб зробити рух вашого персонажа на їхніх ПК плавніше. Якщо потрібно вести корекцію, компонент руху персонажа робить це плавно, використовуючи інтерполяцію. Якщо позиції сервера та клієнта віддаляються занадто далеко, компонент руху персонажа телепортує вашого персонажа назад на його правильну позицію. Це відомо як Rubber-banding, оскільки компонент руху персонажа впроваджує компенсацію затримки. Користувачі зазвичай бачать інших гравців у грі, як вони рухаються дуже плавно, навіть в умовах затримки.

Висновки

Вивчення другого розділу моєї дипломної роботи, присвяченого "Мережевій архітектурі та технологіям у онлайн системах", виявилося захопливим та важливим у контексті розробки сучасних ігор та онлайн додатків.

Починаючи з аналізу архітектурних концепцій у мережевому програмуванні, я відкрив для себе різноманітні підходи, таких як архітектура client-server, peer-to-peer, Listen Server та Dedicated Server. Порівняв та проаналізував мережеві протоколи TCP та UDP Розгляд кожної з них надав важливе уявлення про їхні переваги та обмеження, враховуючи конкретні потреби проектів.

Проаналізував мережеві бібліотеки та фреймворків у контексті Unreal Engine, такі як:

- Unreal Networking Architecture,
- Steamworks Networking,
- Online Subsystem,
- Custom Networking Solutions,
- Voice Over IP бібліотеки,
- Epic Online Services.

Це розширило моє розуміння вибору оптимальних інструментів для реалізації конкретних функціональних вимог у моїй роботі.

Окрім цього, дослідив методи компенсації затримки в онлайн шутерах, що виявилося особливо цікавим і актуальним. Розглянув різноманітні стратегії, такі як: попереднє прогнозування руху, інтерполяція та екстраполяції підкреслив важливість оптимізації та покращення геймплею для забезпечення позитивного досвіду користувачів.

Загальний висновок з даного розділу полягає в тому, що розуміння та використання різноманітних архітектур, бібліотек та компенсаційних стратегій є вирішальним ключем для успішної розробки онлайн систем. Отримані знання стануть основою для подальших кроків у вдосконаленні та створенні нових інновацій у ігровій галузі.

РОЗДІЛ 3
ПРОЄКТУВАННЯ ТА РОЗРОБКА ГРИ

3.1 Концепція та сценарій гри

Був створений власний плагін, який можна додати до будь-якого проекту Unreal Engine, щоб легко перетворити його на багатокористувацьку онлайн гру. Жанром гри було обрано багатокористувацький онлайн шутер. В ньому є можливість створювати власний сервер та приєднуватися до інших гравців. Гравець в головному меню має можливість обирати три різних ігрових режими:

- Free For All, де кожний користувач грає самостійно проти інших гравців
- Teams, усі гравці поділені на дві ігрові команди та гравець має можливість атакувати лише гравців з іншої команди
- Capture the Flag, усі учасники поділені на команди та головна задача це захопити ворожий прапор та принести його на власну базу

Головне меню відображено на рис. 3.1.

Рис. 3.1. Головне меню гри

Для створення власного сервера необхідно натиснути кнопку “Host” та вказати кількість гравців на сервері в рядку “Number of Players”, щоб під'єднатися на вже існуючий сервер необхідно обрати кнопку “Join”.

Для розробки гри використовувався двигун Unreal Engine 5, мова програмування C++ та Blueprints та сервіси Steam.

Користувач має змогу присідати, прицілюватися, стрибати, маневрувати, підіймати спорядження, перезаряджати зброю, кидати гранати, змінювати зброю, виходити до головного меню.

Гравець має кілька видів зброї, серед них пістолет, автомат, дробовик, снайперські гвинтівка, гранатомети та метальні гранати. З можливістю підіймати додаткові предмети котрі розміщені на ігровому рівні: броню, аптечку котра поповнює шкалу здоров’я, блискавку для підвищення швидкості, предмет на підвищення стрибка та боєприпаси.

Гравець бачить на екрані зручний інтерфейс, котрий відображає поточну шкалу здоров’я, броню та кількість боєприпасів, статистику поразок та виграшів, а також ігровий таймер, рис. 3.2.

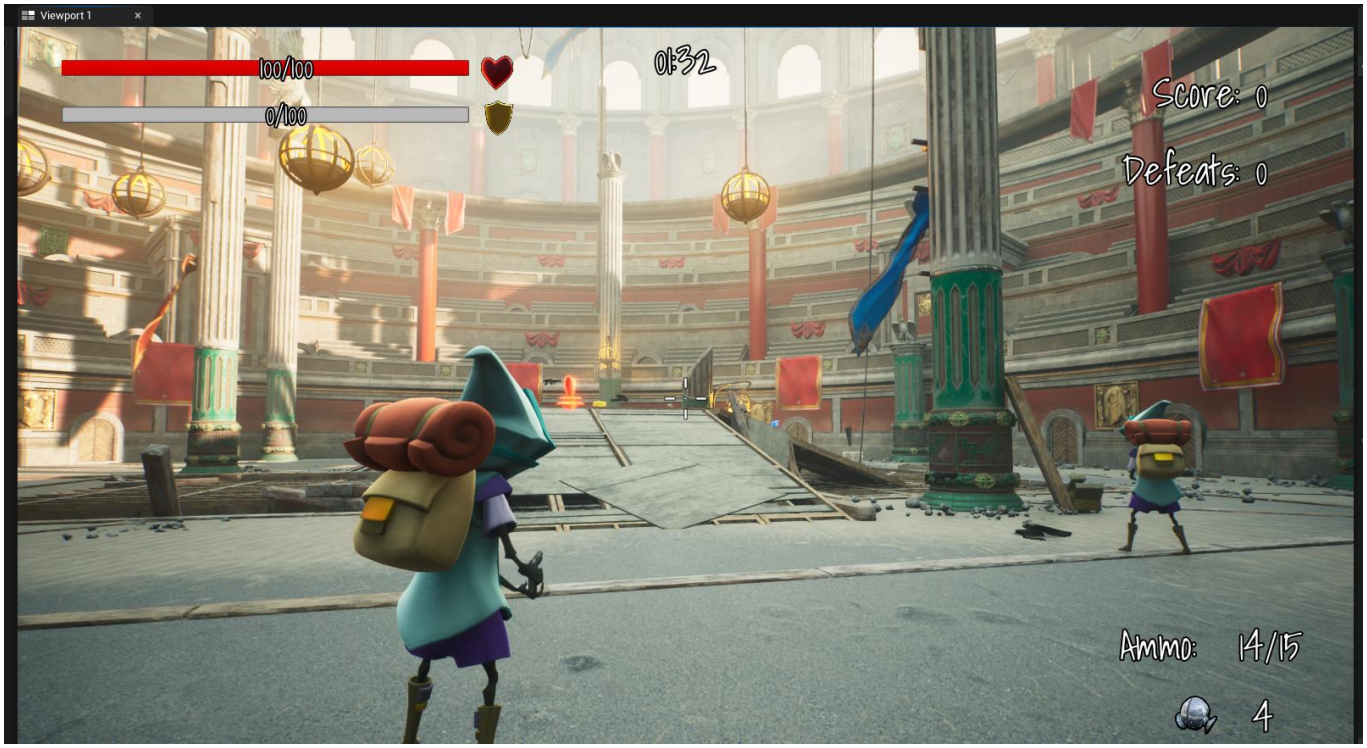

Рис. 3.2. Ігровий режим

Користувацький інтерфейс також відображає оголошення, зокрема: хто кого вибив, хто виграв гру, команди-переможці, таймери зворотного відліку. Якщо гравець має високу затримку на сервері, то гра попереджає його про це.

Гра має спеціальний стан матчу де гравець може: літати на етапі розминки.

Для створення оточення на ігрових рівнях, персонажів, анімацій та спецефектів було обрано використовувати безкоштовні набору моделей з Epic Games Store рис. 3.3. та рис. 3.4.

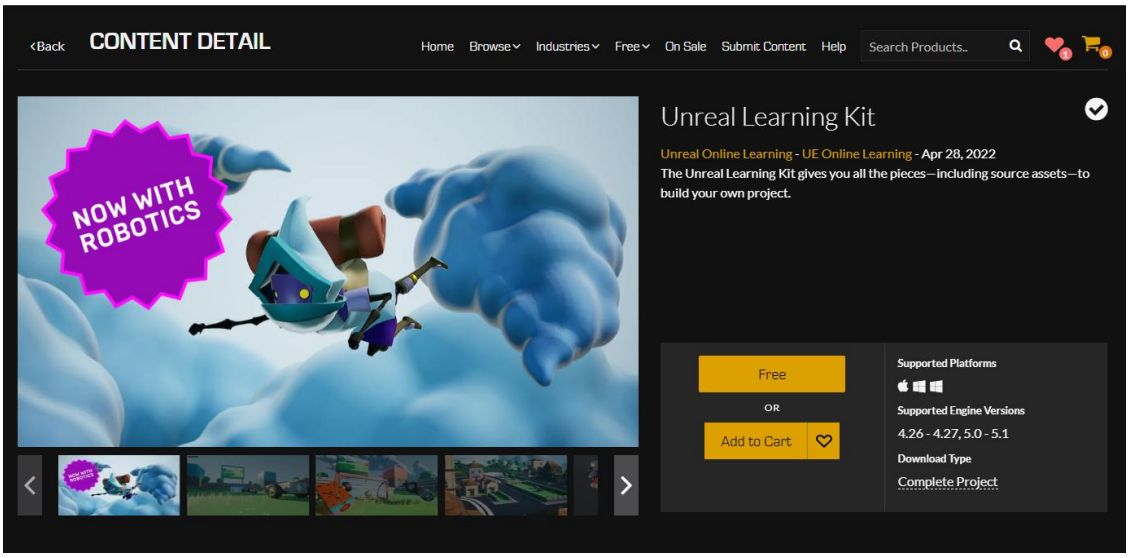

Рис. 3.3. Набір ігрових асетів

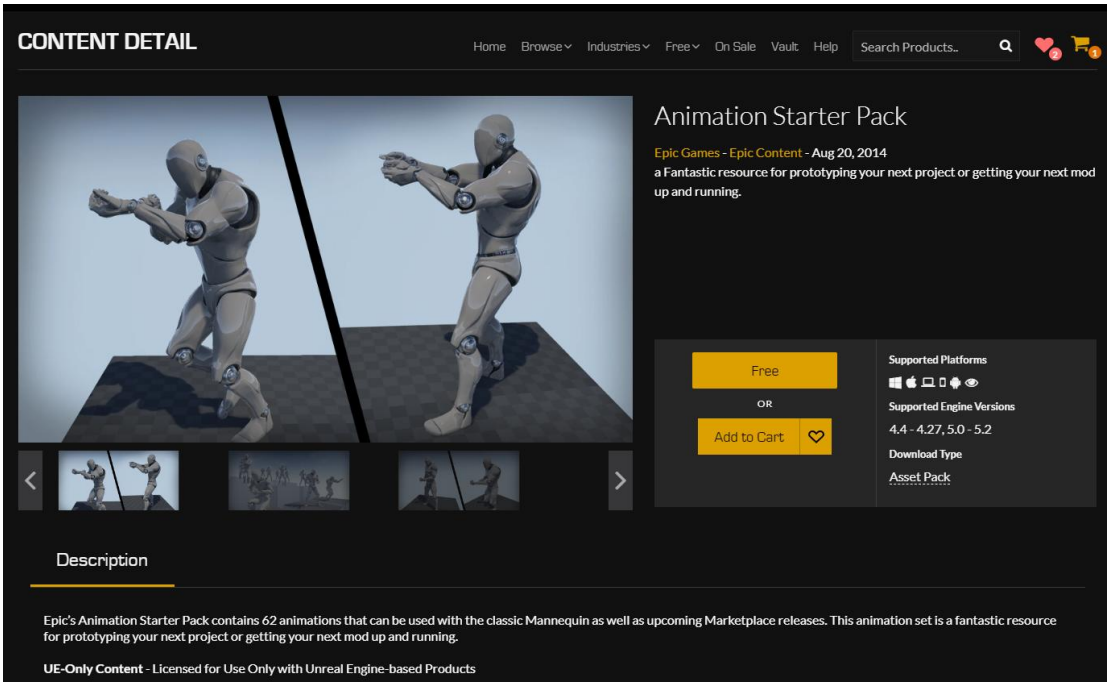

Рис. 3.4 Набір анімації персонажів

3.2 Створення власного плагіну

Створення багатокористувацького плагіну в Unreal Engine - це складний процес, який передбачав розробку засобів для спільної гри гравців у віртуальному середовищі.

Розроблений MultiplayerSessions плагін, можна додати до будь-якого проекту Unreal Engine, щоб перетворити його на багатокористувацьку онлайн гру (рисунок 3.5). Плагін містить усі мережеві налаштування та ресурси, такі як моделі, текстури, аудіо файли тощо, які використовуються в проекті.

Рис. 3.5. MultiplayerSessions плагін

Unreal Engine підтримує розробку плагінів як на мові програмування C++, так і з використанням графічного інтерфейсу Blueprints. Це дозволяє вибирати найзручніший спосіб для розширення функціональності. Для розробки плагіну використовувалась мова програмування C++. Створений плагін був розділений на модулі, котрі містять код, ресурси та налаштування, пов'язані з конкретною частиною. Модулі, від яких залежить проект гри знаходяться у файлі збірки Build.cs (рисунок 3.6).

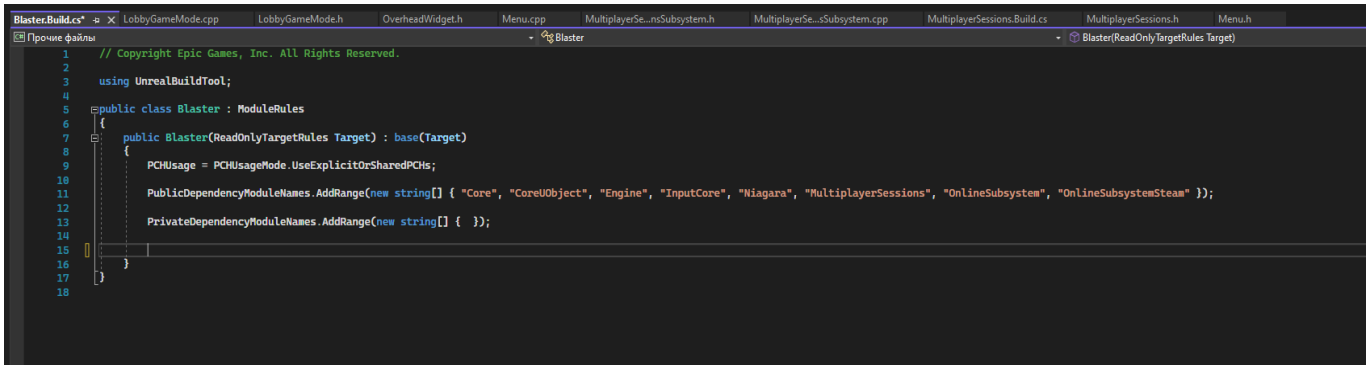

Рис. 3.6. Файл збірки Build.cs

Це дозволяє зберігати проект організованим та додавати або видаляти плагіни за потреби. Плагіни можуть додавати нові класи, компоненти, системи або інші об'єкти, які можуть бути використані в проекті. У файлі проекту гри YourProject.uproject, плагін онлайн-підсистеми Steam з'явився через активацію його в редакторі плагінів. Було включено плагін онлайн-підсистеми в плагін "Multiplayer Sessions". Тепер "Multiplayer Sessions" також має файл "YourPlugin.Build.cs", аналогічно до інших файлів збірки, на рис. 3.7.

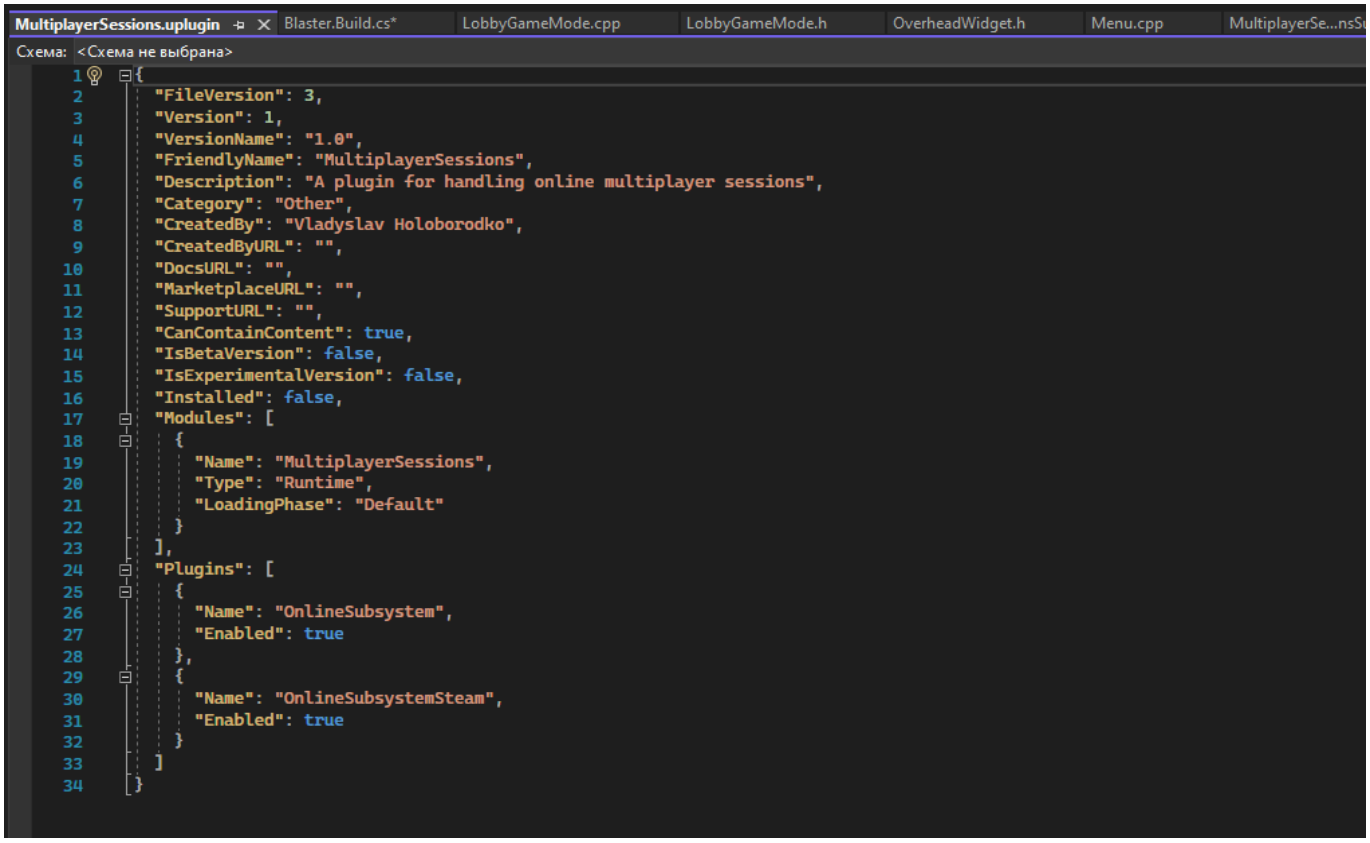

Рис. 3.7. Файл збірки MultiplayerSessions.uplugin

Плагіни широко використовуються в Unreal Engine для розширення функціональності та спрощення розробки ігор та додатків. Вони дозволяють виокремити та повторно використовувати створену функціональність, спрощують командну роботу та дозволяють швидко ввести зміни в проект. Все це робить плагіни важливим інструментом у руках розробників, допомагаючи не лише

розширювати можливості Unreal Engine, але і зробити розробку ігор більш ефективною та креативною.

3.3 Створення власної системи Subsystem

Онлайн-підсистема надає можливість доступу до функціональності онлайн-платформових сервісів, як Steam та Xbox Live. У кожній з цих платформ є свій набір послуг для підтримки таких можливостей, як додавання друзі, відкриття досягнень, налаштування сесій матчмейкінгу. Створена система містить набір інтерфейсів, що призначені для обробки цих різних послуг для кожної платформи. Налаштування проекту для певної платформи робиться у конфігураційному файлі двигуна Engine.ini, на рис. 3.8.

Рис. 3.8. Налаштування файлу Engine.ini

Вибір Steam як сервісу для інтеграції в проект гри вказує на широкий застосунок та популярність цієї платформи серед гравців. Інтеграція з Steam надає можливість використовувати його мережеві функції та зручні інструменти.

Параметр DefaultPlatformService=Steam вказує, що Steam буде використовуватися як основний сервіс для гри. Це свідчить, що гравці матимуть доступ до широкого спектру функцій Steam, таких як друзі, досягнення, чат, інвентар та інші.

Для того, щоб користувачі мали змогу створювати сеанси та приєднуватися до них було ввімкнуте налаштування bInitServerOnClient=true. Якщо значення буде

bInitServerOnClient=false, онлайн-підсистема Steam не буде ініціалізована належним чином. Це може бути важливим для створення гри, де користувачі можуть одночасно бути і серверами та гравцями.

Для з'єднання мережевого драйвера з програмою був вказаний клас Steam, за це відповідає NetConnectionClassName="OnlineSubsystemSteam.SteamNetConnection".

NetServerMaxServerTickRate відповідає за частоту оновлень сервера з клієнтом дозволяючи дотримуватися певного рівня продуктивності та точності в мережевому взаємодії.

Зміна ActiveGameNameRedirects відповідає за перенаправлення гравців на обраний ігровий рівень , що є важливим аспектом для забезпечення коректності та зручності гри в мережі.

Розроблена онлайн-підсистема через клас типу "IOnlineSubsystem", котра отримана через статичну функцію "Get", яка належить до цього класу. Ця функція повертає вказівник на онлайн-підсистему типу вказівника на "IEOnlineSubsystem". За допомогою якої отримується доступ до різноманітних інтерфейсів. На рис. 3.9. показана структура та функціональні можливості класу Online Subsystem та налаштування файлу Engine.ini.

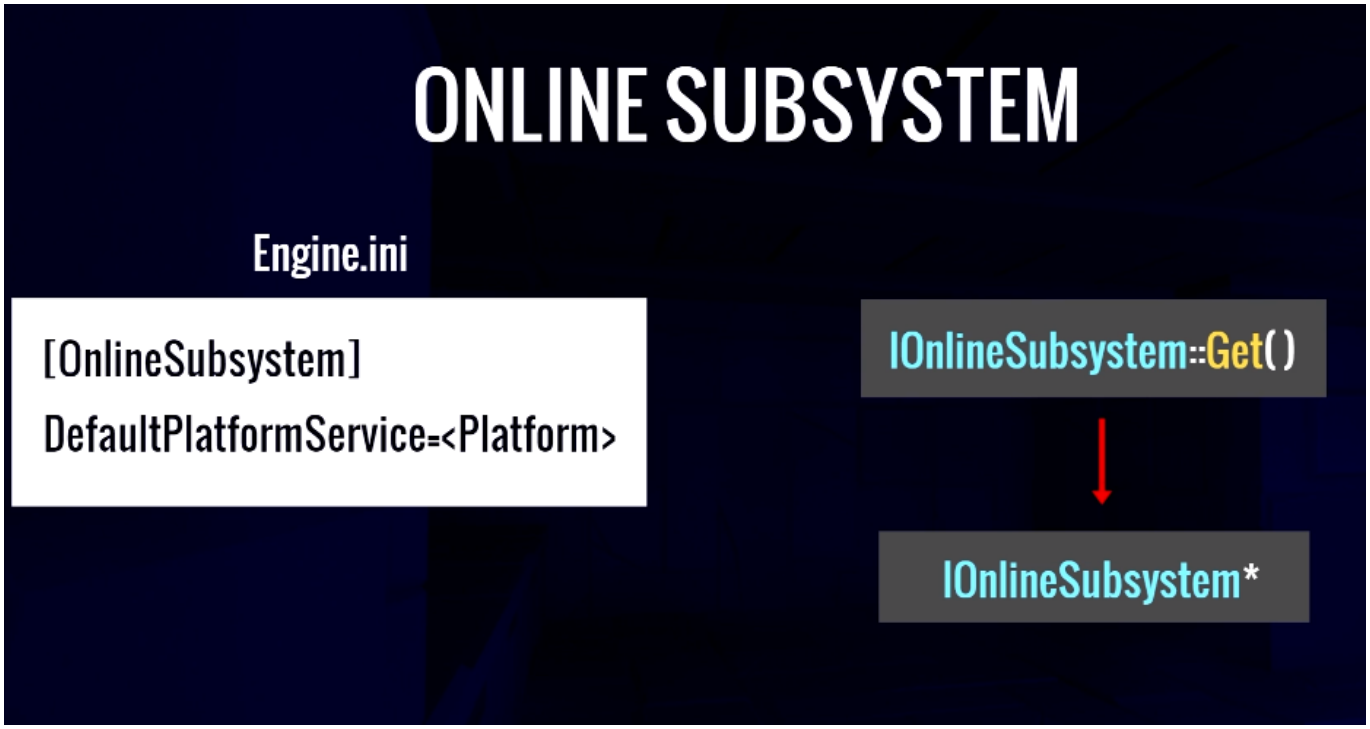

Рис.3.9. Структура Online Subsystem

Взаємодія з файлом Engine.ini для налаштування параметрів системи показує, яким чином розробники мають контроль над конфігурацією та оптимізацією гри для використання мережевих можливостей.

Розроблена власна підсистема для керування онлайн-сесіями, відповідає за створення, управління та завершення гральних сесій. Також вона опрацьовує процес пошуку ігрових сесій та інші можливості матчмейкінгу. Гральну сесію можна уявити як окремий інстанс гри, що запускається на сервері з визначеним набором властивостей.

Сесію можна оголосити загальнодоступною, щоб інші гравці могли знайти її та приєднатися, або зробити її приватною, щоб лише гравці з запрошенням могли приєднатися до неї.

Створений клас MULTIPLAYERSESSIONS_API UMultiplayerSessionsSubsystem містить усі функції багатокористувацького сеансу для обробки функцій сеансу. На рис. 3.10. показана структура функцій інтерфейсу онлайн сесій.

Рис. 3.10. Структура Session Interface Functions

Під час створення функцій сеансу були використані делегати в Unreal Engine, котрі є спеціальними об'єктами та реалізовані за допомогою відомого шаблону проектування Observer.

Клієнтський клас підписується на делегата і слухає його програмні події, а клас, який публікує делегат, може в свою чергу усвідомити всіх клієнтів, навіть не знаючи, що вони існують або їх немає. Коли відбувається певна подія, всі підписані функції викликаються автоматично. Це дозволяло реалізувати різноманітну логіку взаємодії в грі, таку як обробка колізій, реакції на дії гравця. В цілому це робить код менше пов'язаним між собою. На рис. 3.11. показана структура та функціональні можливості делегатів.

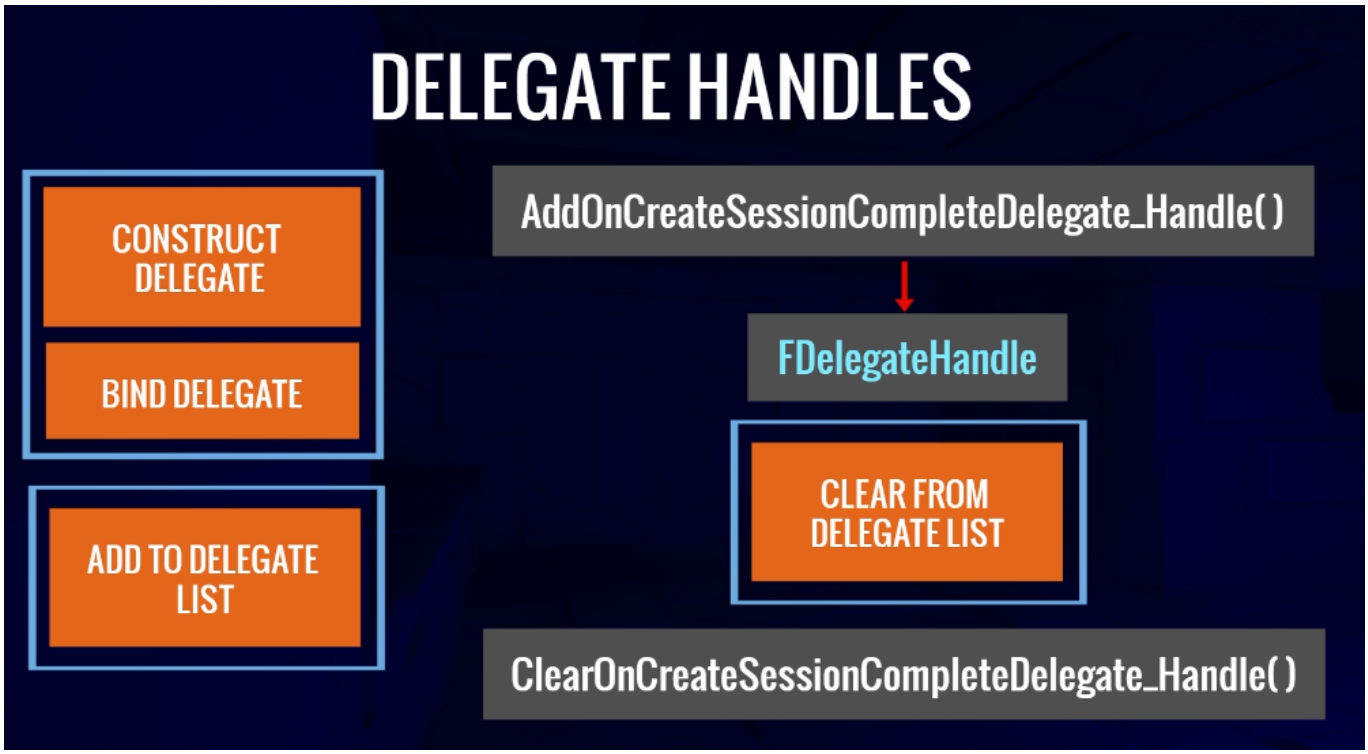

Рис. 3.11. Структура Delegate Handles

Була створена функція AddOnCreateSessionCompleteDelegateHandle() котра повертає об'єкт типу FDelegateHandle, якщо вона містить делегат цього типу, то функція зберігає дескриптор для цього конкретного делегата. Коли система завершує використовувати цей делегат, його можливо очистити зі списку делегатів інтерфейсу сесій.

Інтерфейс сесій має функції для очищення делегатів ClearOnCreateSessionCompleteDelegateHandle(). Функції очищення використовується там, де необхідно видаляти ці делегати зі списку делегатів.

Клас Menu викликає наступні функції сесії інтерфейсу, рис. 3.12:

- void CreateSession(int NumPublicConnections, FString TournamentType). Функція створює сесію та приймає значення кількості гравців та тип матчу;
- void FindSessions(int MaxSearchResults). Функція пошуку сесій.
- void JoinSession(const FOnlineSessionSearchResult& SessionResult). Функція приєднання до сесії, приймає посилання на константний результат пошуку сесії в інтернеті, який називається “session result”. Коли гравець знайде сесію і

вирішить, до якої приєднатися, клас меню отримає результат пошуку і зможе викликати функцію приєднання до сесії і передати його;

- void DestroySession(). Функція знищення сесії,
- void GoSession();

```
UCLASS()
class MULTIPLAYERSESSIONS_API UMultiplayerSessionsSubsystem : public UGameInstanceSubsystem
{
    GENERATED_BODY()
public:
    UMultiplayerSessionsSubsystem();

    //
    // To handle session functionality. The Menu class will call these
    //
    void CreateSession(int32 NumPublicConnections, FString MatchType);
    void FindSessions(int32 MaxSearchResults);
    void JoinSession(const FOnlineSessionSearchResult& SessionResult);
    void DestroySession();
    void StartSession();

    //
    // Our own custom delegates for the Menu class to bind callbacks to
    //
    FMultiplayerOnCreateSessionComplete MultiplayerOnCreateSessionComplete;
    FMultiplayerOnFindSessionsComplete MultiplayerOnFindSessionsComplete;
    FMultiplayerOnJoinSessionComplete MultiplayerOnJoinSessionComplete;
    FMultiplayerOnDestroySessionComplete MultiplayerOnDestroySessionComplete;
    FMultiplayerOnStartSessionComplete MultiplayerOnStartSessionComplete;

    int32 DesiredNumPublicConnections{};
    FString DesiredMatchType{};
protected:
```

Рис. 3.12. Програмна реалізація виклику Session Interface Functions

Розроблена власна система делегатів для класу меню для внутрішніх зворотних викликів “callbacks”, яка використовуються внутрішньою підсистемою для взаємодії з інтерфейсом онлайн-сеансу, на рис. 3.13.

```
//
// Our own custom delegates for the Menu class to bind callbacks to
//
FMultiplayerOnCreateSessionComplete MultiplayerOnCreateSessionComplete;
FMultiplayerOnFindSessionsComplete MultiplayerOnFindSessionsComplete;
FMultiplayerOnJoinSessionComplete MultiplayerOnJoinSessionComplete;
FMultiplayerOnDestroySessionComplete MultiplayerOnDestroySessionComplete;
FMultiplayerOnStartSessionComplete MultiplayerOnStartSessionComplete;

int32 DesiredNumPublicConnections{};
FString DesiredMatchType{};
```

Рис. 3.13. Програмна реалізація виклику Session Interface Functions

Внутрішні зворотні виклики для делегатів, які додаються до списку делегатів онлайнового інтерфейсу сеансу. Їх не потрібно викликати за межами створеного класу.

Delegate handle- це зазвичай посилання або ідентифікатор, який дозволяє зберігати та взаємодіяти з делегатами або подіями. Їх можна використовувати як об'єкти для додавання, видалення або виклику підписаних функцій. Було розроблено кілька дескрипторів делегатів, по одному для кожного з розроблених делегатів, котрі використовує створена система, щоб видалити делегати зі списку делегатів, як тільки вони більше не потрібні, на рис. 3.14:

```
FOnCreateSessionCompleteDelegate CreateSessionCompleteDelegate;
FDelegateHandle CreateSessionCompleteDelegateHandle;
FOnFindSessionsCompleteDelegate FindSessionsCompleteDelegate;
FDelegateHandle FindSessionsCompleteDelegateHandle;
FOnJoinSessionCompleteDelegate JoinSessionCompleteDelegate;
FDelegateHandle JoinSessionCompleteDelegateHandle;
FOnDestroySessionCompleteDelegate DestroySessionCompleteDelegate;
FDelegateHandle DestroySessionCompleteDelegateHandle;
FOnStartSessionCompleteDelegate StartSessionCompleteDelegate;
FDelegateHandle StartSessionCompleteDelegateHandle;

bool bCreateSessionOnDestroy{ false };
int32 LastNumPublicConnections;
FString LastMatchType;
```

Рис. 3.14. Програмна реалізація Delegate handle

3.4 Створення класу Menu

Користувацький інтерфейс в Unreal Engine має величезне значення для розробки гри. Великої уваги заслуговує головне меню гри, тому, що воно є першим контактом користувача з грою та визначає перші враження та зручність. Добре розроблений користувацький інтерфейс меню повинний бути зручним та інтуїтивно зрозумілим. Гравці повинні легко знаходити налаштування: почати нову гру, завантажити збережену гру, вибрати рівень складності та інші опції. Зручне UI полегшує навігацію гравця та покращує загальний користувацький досвід.

Меню використовує створену підсистему багатокористувацьких сесій, викликає її функції та керує ігровими сесіями через віджет меню, щоб мати можливість отримувати доступ до плагіну.

Розроблений інтерфейс при натисканні на кнопку “Host” дає змогу користувачу створювати онлайн сесію гри та бути хостом для інших користувачів, а кнопка “Join” відповідає за підключення до онлайн сесії інших гравців. Користувач має можливість обрати режим гри та кількість гравців, що будуть брати участь у грі при створені онлайн сесії, на рис.3.15.

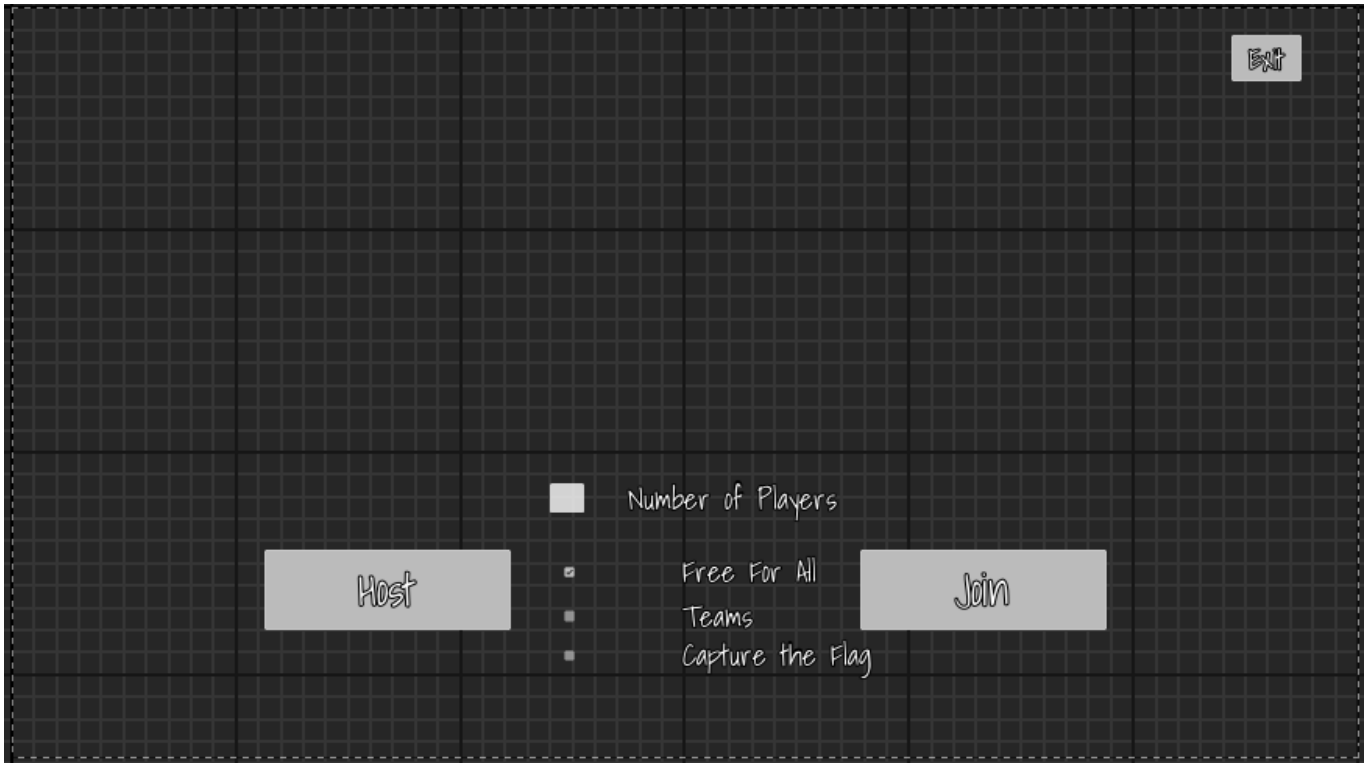

Рис. 3.15. Програмна реалізація Delegate handle

Логіка меню була реалізована за допомогою блупрінтів, C++ та Unreal Engine, на рисю 3.16. Інтерфейс користувача реалізований за допомогою UMG (Unreal Motion Graphics). Вона надає візуальний інтерфейс для створення UI елементів, таких як кнопки, текстові поля, панелі тощо. За допомогою чого можна розміщувати та стилізувати ці елементи безпосередньо в редакторі Unreal Engine, використовуючи Blueprint мову візуального програмування або мову програмування C++ для визначення поведінки. Також використовувалась бібліотека Slate, що дозволяє створювати дуже кастомізовані інтерфейси, такі як інтерфейси редактора або ігор, які вимагають високої ефективності. Slate базується на C++, і використовується для створення різних видів вікон, панелей, кнопок і текстових полів.

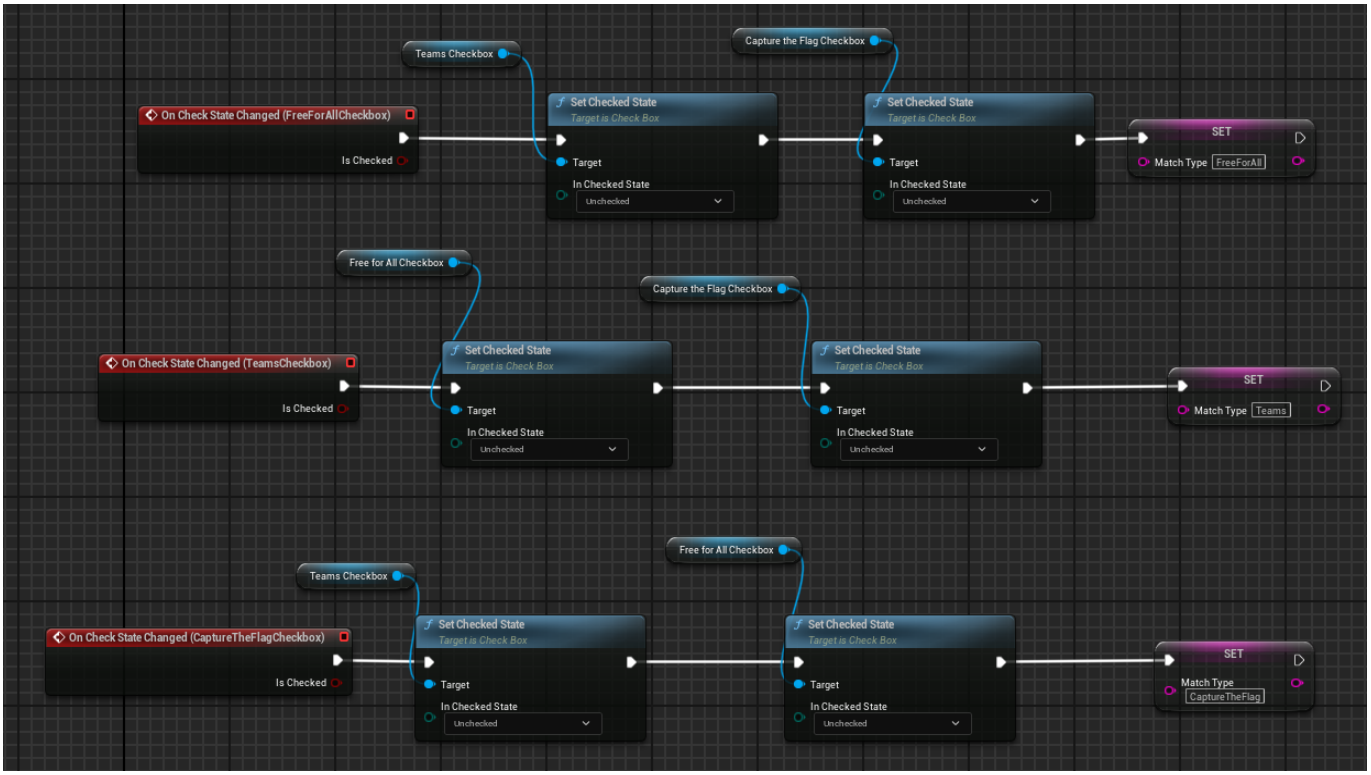

Рис. 3.16. Реалізація Menu за допомогою блупрінтів

3.5 Створення ігрового рівня

Розробка ігрового рівня є однією з ключових аспектів при створенні високоякісної відеогри. Цей процес вимагав поєднання творчості та технічних навичок з метою створення захоплюючого та виразного ігрового середовища, яке пропонує гравцям цікаві виклики та незабутні враження. Створений рівень є цілком сумісний із загальною концепцією гри та при цьому обслуговує різні ігрові механіки, завдання та взаємодію гравця з навколишнім світом.

На даному рівні була створена ігрова локація, що відтворює стародавній римський Колізей, на рис. 3.17. Сам колізей вражає своєю архітектурою, яка передає дух стародавнього Риму, з масивними стінами, арками і кам'яними деталями, які створюють унікальну атмосферу.

Рис. 3.17. Ігрова локація Free For All

Ця арена призначена для з режиму "Битва всіх проти всіх", де кожен учасник гри виступає самостійно, без командної підтримки.

На арені розташовані різні сховища, об'єкти для взаємодії та зони, що сприяють виникненню непередбачуваних ситуацій і змінюють тактику гравців. Кожен рівень пророблений в деталях та розрахований на інтенсивні битви та незабутні емоції.

Для розробки рівня використовувались безкоштовні моделі з Unreal Engine Marketplace. На ігровому рівні гравці можуть знаходити зброю, боєприпаси та бусти, що відновлюють його ігрові показники або надають покращення навичок на деякий час, на рис. 3.18.

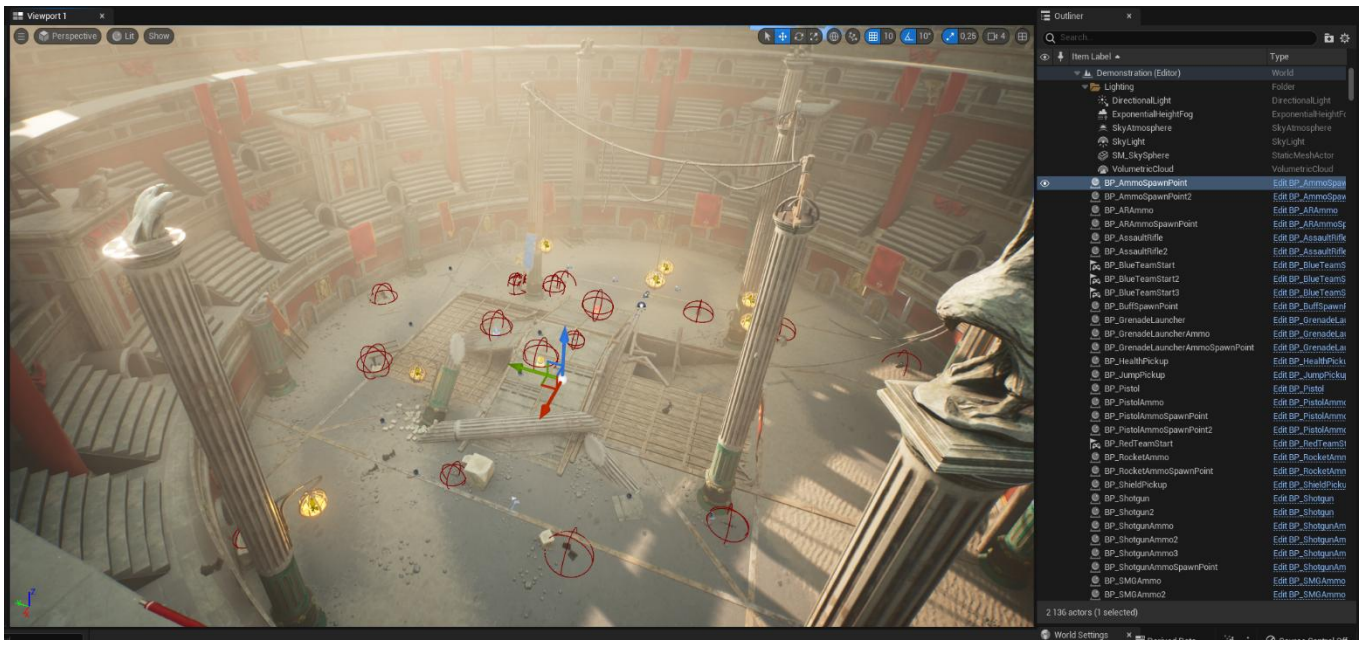

Рис. 3.18. Розташування предметів на рівні

На рівні було розміщено:

- точки появи користувачів,
- точки освітлення,
- хмари,
- туман,
- напрямок руху вітру,
- предмет серце, що відновлюють здоров’я,
- предмет блискавка, збільшує швидкість гравця,
- предмет щит, відновлює броню гравця,
- точки інтересу,
- текстури,
- пістолети, автомати, дробовики, снайперські гвинтівки, ракетниці, гранатомети та метальні гранати,
- боєприпаси до зброї.

Створений рівень був оптимізованим. У процесі оптимізації ігрового рівня було виконано комплекс робіт з метою підвищення продуктивності та вдосконалення геймплею. Використовувались оптимізовані меші архітектури та об'єктів,

обмірковане керування текстурами, зменшення обсягу анімації та ретельний відбір об'єктів, що брали участь в сцені.

Були введені обмеження кількості одночасно активних об'єктів на рівні. Було налаштовано рівні деталізації для об'єктів, що зменшують роздільну здатність та зменшений рівень деталізації на великій відстані.

3.6 Система відстежування гравців

В сучасному світі відеоігор важко уявити безперервне вдосконалення та розвиток мережевих середовищ. Однією з ключових складових цього постійного розвитку є система відстеження гравців, яка дозволяє ефективно взаємодіяти з численними учасниками гри та створювати захоплюючі мережеві сесії.

Була створена система, щоб відстежувати гравців, які приєднуються до ігрової сесії. Ця розробка виводить кількість активних гравців та вказує на кількість гравців, які приєдналися. Основна логіка цього відстеження реалізована за допомогою концепцій режиму та стану гри, на рис. 3.19.

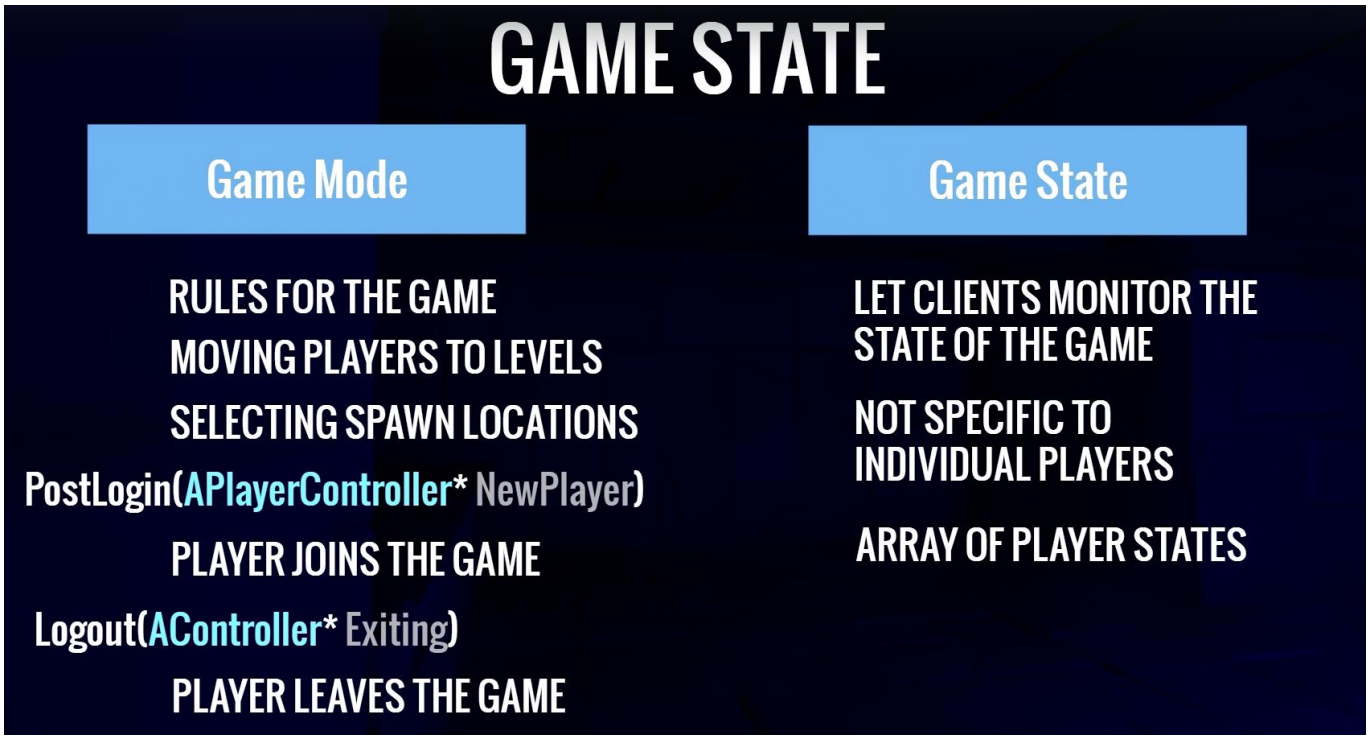

Рис. 3.19. Структура режиму та стану гри

Режим гри зберігає усі правила гри, переміщає гравців на нові рівні, інші режими гри та на різні місця їх появи на карті. Game mode має декілька успадкованих функцій, які корисні для відстеження входу та виходу гравця із сесій.

Post login - це успадкована віртуальна функція, яка викликається кожного разу, коли гравець приєднується до гри, і надає доступ до цього контролера гравця.

Функція виходу “Logout” викликається, коли гравець покидає гру, і знову надає доступ до контролера та цієї функції.

Стан гри призначений для зберігання інформації про поточний стан гри. Клієнти отримують доступ до різних аспектів стану гри, таких як рахунок чи кількість перемог. У Game State зберігається масив станів гравців, клас стану гравця призначений для зберігання інформації, яка стосується конкретного гравця, такої як рахунок чи інші персональні досягнення. Режим гри має доступ до стану гри, він містить масив станів гравців і скільки гравців є в грі, перевіряючи розмір цього масиву.

Ця система відстеження гравців відкриває безліч можливостей для покращення імерсії та взаємодії у ігровому середовищі. Подальше дослідження та розвиток цих технологій обіцяють ще більше захопливих можливостей у майбутній ігровій індустрії.

3.7 Створення компонента компенсації затримки

У сучасному світі онлайн-геймінгу, де мільйони гравців об'єднуються в одному віртуальному просторі, питання затримки стає основною турботою для розробників та гравців. Одним з ключових аспектів для розв'язання цієї проблеми є розробка та впровадження компонента компенсації затримки. У цьому розділі розглянемо процес створення такого компонента та його ключові елементи.

Був створений компонент компенсації затримки, котрий зберігає інформацію про гравців і необхідні алгоритми для серверного повороту часу.

Першим етапом стало визначення різновидів затримок та їх впливу на геймплей. Після чого були визначені цілі компенсації: зниження часу відгуку гравця,

оптимізація синхронізації дій між гравцями та поліпшення загального ігрового досвіду.

Загальний підхід під час створення полягав в тому, що система повинна зберігати історію руху гравця, і це має робитися принаймні на сервері. Гра може це робити на кожному кадрі, коли позиція персонажа змінюється. Для цього було спроектована структура даних, що буде зберігати цю інформацію - своєрідну історію кадрів.

Історія кадрів містить конкретну інформацію, де персонаж знаходиться, і час, коли він був у цій позиції, на рис. 3.20.

Рис. 3.20. Приклад історії запису рухів гравця

Під час розробки виникали питання скільки даних слід зберігати у системі та у якому вигляді. Для зберігання даних було обрано використовувати прямокутники Hit boxes. До ігрових персонажів були додані та налаштовані невидимі для інших гравців бокс-компоненти, котрі мають області для попадань, що відповідають формі та геометрії відповідних частин тіла персонажа. Під час обчислення попадань, гра використовує ці бокси для перевірки, чи відбулася взаємодія з ігровим персонажем і

вони є важливим елементом у графіці та фізичному моделюванні в ігровій розробці, на рис. 3.20.

Рис. 3.21. Приклад прямокутників персонажу

Даний метод є ефективним і компактним способом зберігання даних, і виявився досить точним. Такий підхід дозволив отримати точну інформацію про знаходження гравця у просторі у певний період часу в минулому, щоб вирішити питання про попадання. На основі цієї інформації була створені структури даних FFramePackage та FBoxInformation. Інформація про hitbox буде міститиме:

- розташування Location,
- обертання Rotation,
- розміри BoxExtent.

На рис. 3.22. показана структура даних FFramePackage та FBoxInformation.

```
USTRUCT(BlueprintType)
struct FBoxInformation
{
    GENERATED_BODY()

    UPROPERTY()
    FVector Location;

    UPROPERTY()
    FRotator Rotation;

    UPROPERTY()
    FVector BoxExtent;
};

USTRUCT(BlueprintType)
struct FFramePackage
{
    GENERATED_BODY()

    UPROPERTY()
    float Time;

    UPROPERTY()
    TMap<FName, FBoxInformation> HitBoxInfo;

    UPROPERTY()
    ABlasterCharacter* Character;
};
```

Рис. 3.22. Структура даних "FFramePackage" та “ FBoxInformation”.

Створений алгоритм відіграє ключову роль у зберіганні певної інформації щодо областей попадань hitbox для кожного персонажа в конкретний момент часу. Цей алгоритм не лише зберігає дані про позиції та розміри хітбоксів, але і надає можливість ефективно відтворювати стан гри в певний момент часу.

Функція SaveFramePackage() відповідає за збереження цієї інформації. Спочатку вона проводить перевірку на валідність змінної персонажа, щоб уникнути можливих помилок, таких як нульовий вказівник, на рис. 3.23.

```
void ULagCompensationComponent::SaveFramePackage(FFramePackage& Package)
{
    Character = Character == nullptr ? Cast<ABlasterCharacter>(GetOwner()) : Character;
    if (Character)
    {
        Package.Time = GetWorld()->GetTimeSeconds();
        Package.Character = Character;
        for (auto& BoxPair : Character->HitCollisionBoxes)
        {
            FBoxInformation BoxInformation;
            BoxInformation.Location = BoxPair.Value->GetComponentLocation();
            BoxInformation.Rotation = BoxPair.Value->GetComponentRotation();
            BoxInformation.BoxExtent = BoxPair.Value->GetScaledBoxExtent();
            Package.HitBoxInfo.Add(BoxPair.Key, BoxInformation);
        }
    }
}
```

Рис. 3.23. Функція SaveFramePackage()

Тому вона перевіряє, чи дорівнює зміна Character нульовому вказівнику, і якщо так, то виконується приведення до типу "blaster character" на основі "GetOwner", оскільки "GetOwner" повертає актора, який володіє цим компонентом.

Створений контейнер для зберігання ключів-значення "TMap", з ключами типу "FName" та відповідними значеннями "UBoxComponent". . Цей контейнер позначається як "HitCollisionBoxes" і надає зручний механізм доступу до всіх хітбоксів персонажа.

У цьому циклі заповнюється "TMap" "HitCollisionBoxes" для кожного ключа "FName" з об'єктом "UBoxComponent". Це створює контейнер, який містить всі хітбокси персонажів разом з їхніми іменами.

Для візуальної перевірки та тестування збережених пакетів кадрів була розроблена функція showFramePackage(). Ця функція проходить по всім хітбоксам у пакеті та використовує функцію drawDebugBox(), щоб намалювати кожен хітбокс на екрані гри. Користувач може вказати бажаний колір для відображення цих коробок, а також визначити час, протягом якого вони будуть видимими на екрані, на рис. 3.24. Це надає можливість розробникам та тестерам візуально перевірити коректність роботи алгоритму збереження та відтворення інформації про хітбокси в грі.

```cpp
void ULagCompensationComponent::ShowFramePackage(const FFramePackage& Package, const FColor& Color)
{
    for (auto& BoxInfo : Package.HitBoxInfo)
    {
        DrawDebugBox(
            GetWorld(),
            BoxInfo.Value.Location,
            BoxInfo.Value.BoxExtent,
            FQuat(BoxInfo.Value.Rotation),
            Color,
            false,
            4.f
        );
    }
}
```

Рис. 3.24. Функція showFramePackage ()

Для забезпечення плавного та натурального переходу між двома кадрами анімації або станами об'єкта, що змінюються з часом, була створена функція Interp Between Frames(). Концепція роботи функції інтерполяції проілюстрована на рис.3.25.

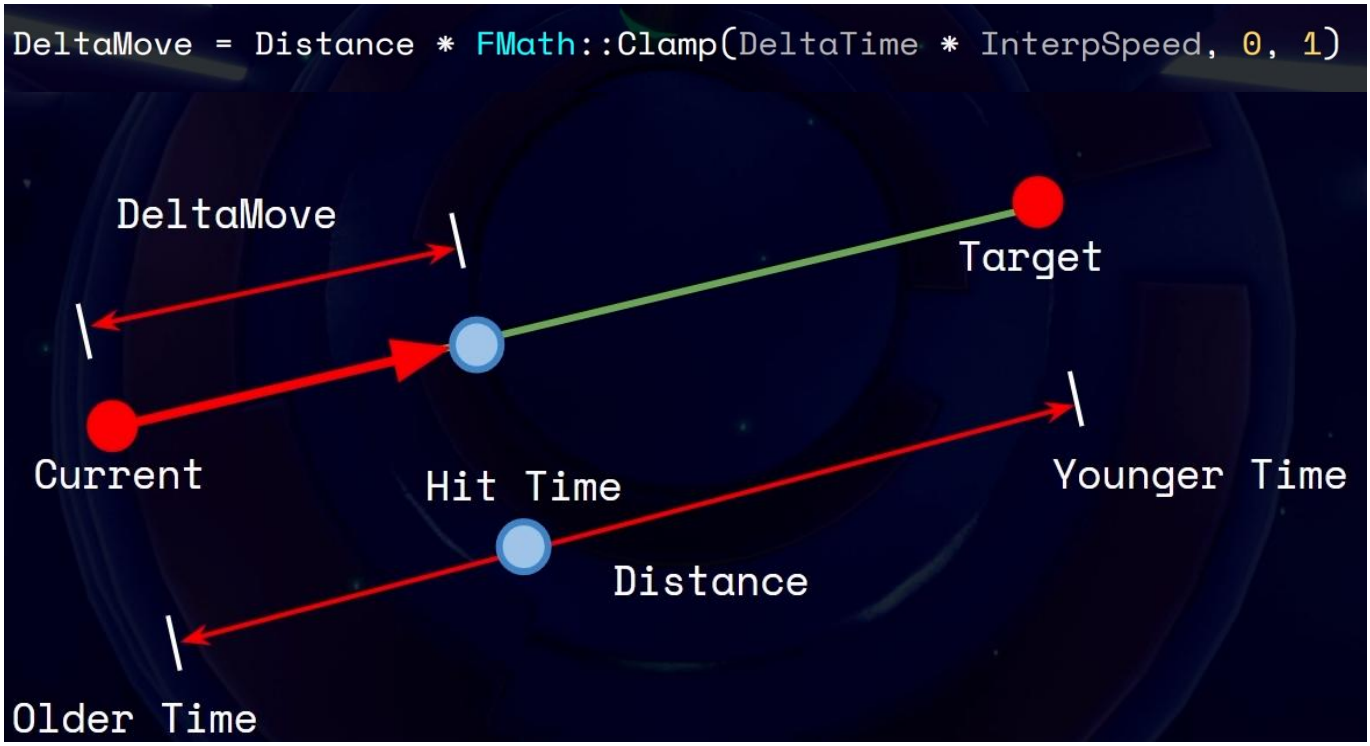

Рис. 3.25. Приклад функції Interp Between Frames

Функція буде інтерполювати інформацію між двома кадрами: молодшим і старшим, і результатом буде кадр, що містить всі інтерпольовані значення для розташування та обертання, і саме ці значення буде повертатися з функції.

Спочатку функція розраховує змінну Distance, і вона буде рівна YoungerFrameTime мінус OlderFrameTime. За допомогою значення дистанції був обраховане значення DeltaMove. Під час обрахунків використовувалась функція "FMath::Clamp" і передані в неї перший вхідний параметр обмежений між 0 і 1. Ця дробова частина представляє те, наскільки далеко по лінії від поточного до цільового вектора, вектор буде переміщений. Множенням на дельта час функція отримує рух незалежний від кадрів.

Ця функція особливо корисна при розробці ігор, анімацій та сцен, де потрібно створити плавні переходи між станами об'єкта або анімаційними кадрами.

Для помітки високого пінгу у гравця та її анімацію показу на головному екрані була створена функція CheckPing та блупрінт character overlay widget який відтворюється на екрані користувача, якщо пінг між гравцем і сервером перевищує 50 мілісекунд. Варіант зміни цього значення доступний у Player Controller для налаштувань, на рис. 3.26.

```
void ABlasterPlayerController::CheckPing(float DeltaTime)
{
    if (HasAuthority()) return;
    HighPingRunningTime += DeltaTime;
    if (HighPingRunningTime > CheckPingFrequency)
    {
        PlayerState = PlayerState == nullptr ? GetPlayerState<APlayerState>() : PlayerState;
        if (PlayerState)
        {
            if (PlayerState->GetPing() * 4 > HighPingThreshold) // ping is compressed; it's actually ping /
            {
                HighPingWarning();
                PingAnimationRunningTime = 0.f;
                ServerReportPingStatus(true);
            }
            else
            {
                ServerReportPingStatus(false);
            }
        }
        HighPingRunningTime = 0.f;
    }
    bool bHighPingAnimationPlaying =
        BlasterHUD && BlasterHUD->CharacterOverlay &&
        BlasterHUD->CharacterOverlay->HighPingAnimation &&
        BlasterHUD->CharacterOverlay->IsAnimationPlaying(BlasterHUD->CharacterOverlay->HighPingAnimation);
    if (bHighPingAnimationPlaying)
    {
        PingAnimationRunningTime += DeltaTime;
        if (PingAnimationRunningTime > HighPingDuration)
        {
            StopHighPingWarning();
        }
    }
}
```

Рис. 3.26. Функція CheckPing

3.8 Створення компонента підтвердження попадання

Для ефективного впровадження системи гри, де важливо визначити, чи попав гравець пострілом або атакою, є важливим створення компонента підтвердження попадання. Цей компонент відповідає за визначення та підтвердження моменту, коли атака або постріл влучає у ціль.

Як тільки супротивник потрапив у поле іншого гравця, він робить постріл і відправляє цю інформацію на сервер. До того моменту, коли інформація про постріл долає відстань до сервера, чинна версія персонажа може вже рухатися, і серверна версія персонажа може вже відрізнятися. На пристрої користувача персонаж супротивника може бути в полі його зору, але на пристрої сервера цей персонаж може бути трохи спереду.

Для розв'язання цієї проблеми, необхідно ще більше перемотати час, щоб врахувати час подорожі, який потрібен серверу для відтворення позиції персонажа супротивника на кожен кадр, коли надсилається серверу час попадання. Для цього була створена функція підтвердження попадань. Функція повертає булеві значення, якщо попадання є успішним true та false якщо гравець промахнувся, на рис. 3.27.

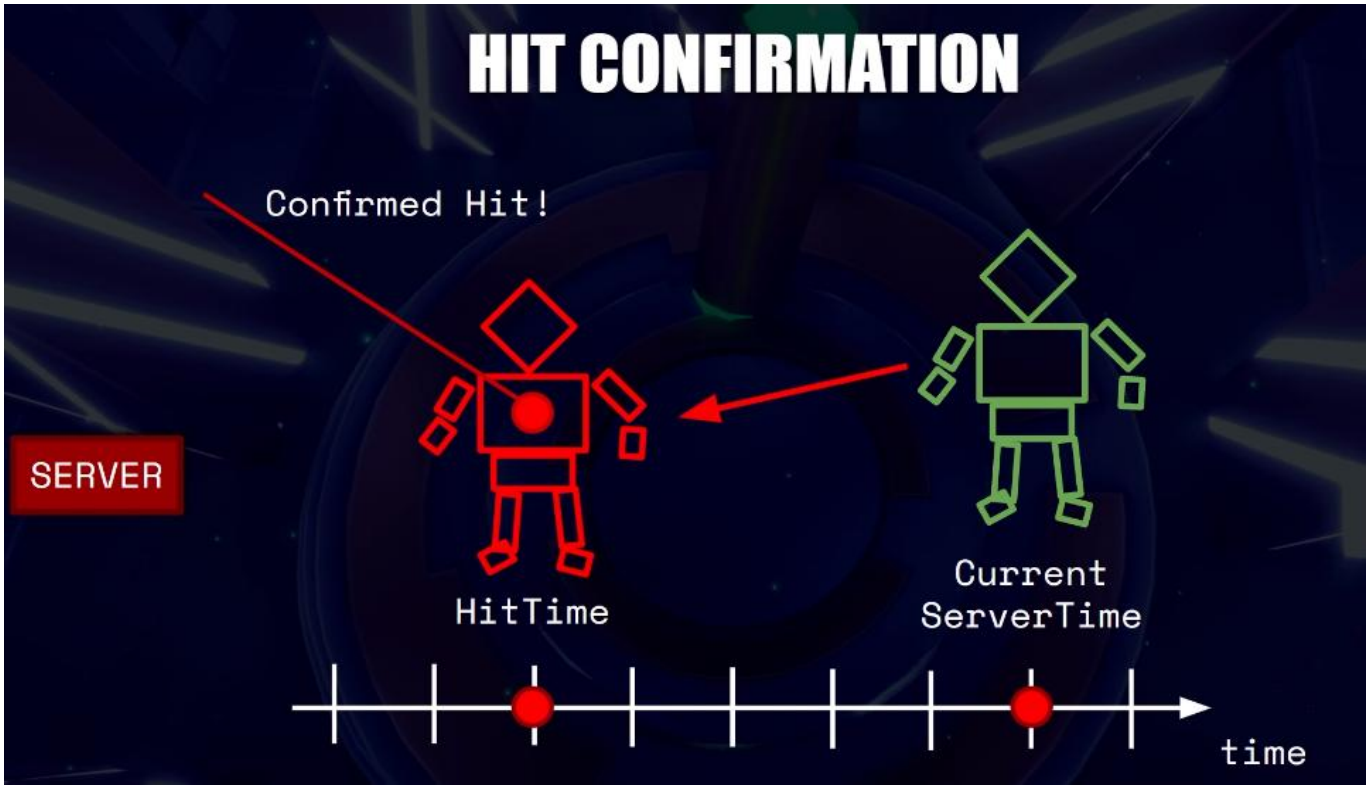

Рис. 3.27. Функція ConfirmHit

Спочатку функція розраховує різницю між поточним часом сервера та часом влучення. Ця різниця є результатом, як далеко віддалені у часі ці дві події.

Система перевіряє, чи ця різниця більше, менше або рівна нуля.

Якщо різниця дорівнює нулю, це означає, що попадання відбулося в точний момент часу влучання, і це є дійсним попаданням.

Якщо різниця від'ємна, це означає, що гравець влучив до моменту влучення серверної версії персонажа.

Якщо різниця позитивна, це означає, що користувач влучив після часу влучання серверної версії персонажа.

Якщо різниця близька до нуля та було встановлено певний межовий інтервал, в якому вважається попадання дійсним. Це інформація є корисною для компенсації невеликої затримки в мережі або інших чинників.

Отже, функція для підтвердження попадань повинна повернути істину, якщо попадання дійсне, і брехню, якщо воно не є дійсним.

Висновки

В ході дослідження розділу "Проектування та розробка гри" було ретельно розглянуто кожен аспект створення ігрового продукту, від концепції та сценарію до впровадження різноманітних компонентів та систем. Цей розділ відображає не лише технічний аспект розробки, але й стратегічне планування та концептуальне проектування, спрямоване на створення цікавої та захоплюючої гри. Було на прикладі продемонстровано готовий ігровий продукт та стадії розробки.

Починаючи з концепції та сценарію гри, було виявлено важливість чіткої визначеності цілей та механік гри ще на етапі її створення. Ретельна розробка цього аспекту дозволила визначити основні елементи, які сприятимуть вдалому імплементуванню концепції в реальний ігровий геймплей.

Створення власного плагіну та системи Subsystem виявилося ключовим для реалізації специфічного функціонала гри та виокремлення його від інших частин системи. Це забезпечило гнучкість та можливість майбутнього розширення без значного впливу на вже існуючий код.

Важливим аспектом стала розробка класу Menu, який визначив візуальний стиль та навігацію головного меню гри, надаючи гравцям інтуїтивно зрозуміле та привабливе середовище.

Створення ігрового рівня та системи відстежування гравців відображає важливий аспект геймдизайну, дизайну навколишнього середовища, оптимізації, забезпечуючи різноманітність та взаємодію в грі, а також дозволяючи ефективно відстежувати та реагувати на дії гравців.

Особливо важливим стало впровадження системи компенсації затримки (Lag Compensation) та компонента підтвердження попадання, що сприяють покращенню якості мережевого геймплею та забезпеченню чесної та стабільної гри для усіх учасників.

В цілому, розділ "Проектування та розробка гри" відобразив інтегрований підхід до створення ігрового досвіду, враховуючи технічні, дизайнерські та мережеві аспекти. Отримані результати відіграли ключову роль у створенні гри та мережевого плагіну.

ВИСНОВКИ

Ця магістерська робота була присвячена вивченню та аналізу мережевої системи в багатокористувацьких онлайн іграх, розроблених з використанням ігрового двигуна Unreal Engine. Головна мета полягала в розкритті ключових аспектів створення мережевих ігор у цьому середовищі й обговоренні їх впливу на якість та функціональність багатокористувацьких ігор.

В першому розділі було зроблено огляд і порівняльний аналіз чотирьох основних ігрових двигунів: Unreal Engine, Unity, CryEngine та Amazon Lumberyard. Визначив, що кожен з них має свої унікальні можливості та особливості, які відображаються на різноманітних аспектах розробки багатокористувацьких ігор. Проаналізувавши усі наявні ігрові двигуни, для розробки та аналізу у наступних розділах було обрано ігровий двигун Unreal Engine.

Другий розділ був присвячений дослідженню мережевої архітектури та технології в онлайн системах, через аналіз різних архітектурних концепцій, таких як client-server, peer-to-peer, Listen Server та Dedicated Server. Порівняв та проаналізував мережеві протоколи TCP та UDP. Розгляд кожної з них надав важливе уявлення про їхні переваги та обмеження, враховуючи конкретні потреби проектів. Слід зазначити, що особлива увага була приділена мережевим бібліотекам та фреймворкам Unreal Engine, включаючи: Unreal Networking Architecture, Steamworks Networking, Online Subsystem, Custom Networking Solutions, Voice Over IP бібліотеки, Epic Online Services (EOS). Значну роль відіграли дослідження методів компенсації затримки в багатокористувацьких іграх. Розглянув різноманітні стратегії, такі як: попереднє прогнозування руху, інтерполяція та екстраполяції підкреслив важливість оптимізації та покращення геймплею для забезпечення позитивного досвіду користувачів. Це розширило моє розуміння вибору оптимальних інструментів для реалізації конкретних функціональних вимог у моїй роботі.

В третьому розділі було відображено процес проектування та розробки онлайн гри. Протягом дослідження були виконані важливі завдання, серед яких створення власного плагіна, здатного перетворити будь-який проєкт Unreal Engine в онлайн

багатокористувацьку гру. Були розроблені та впроваджені в дію методи компенсації затримок для забезпечення стабільної гри, попри великі інтернет-затримки. Досягнутою метою також стало створення підсистеми для керування онлайн-сесіями й впровадження компонентів підтвердження попадання та системи відстежування гравців в розробці багатокористувацького онлайн шутера. Надано опис створення ігрового рівня та системи відстежування гравців.

Ці відкриття мають важливе значення для майбутньої розробки ігор, зокрема багатокористувацьких онлайн ігор. Результати цієї роботи дозволять розробникам краще розуміти мережеві системи та стратегії оптимізації, що стають все більш перспективними в сучасному світі геймдеву.

Надалі, досвід отриманий від розробки плагіна для Unreal Engine та багатокористувацької гри, може бути використаним для створення різноманітних мережевих ігор із врахуванням специфіки різних жанрів і аудиторій. Можливими напрямками для подальшого дослідження можуть стати аналіз різних технологій і протоколів, які можуть бути застосовані в Unreal Engine для мережевої взаємодії гравців, а також вивчення нових способів оптимізації мережевого взаємодії з використанням різних методів компенсації затримок.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fortnite just had its biggest day ever, 6 years in [Electronic resourse] – Mode of access: https://www.polygon.com/23948437/fortnite-og-chapter-1-player-record
2. Unreal Engine [Electronic resourse] – Mode of access: https://www.unrealengine.com/en-US/unreal-engine-5
3. Unity [Electronic resourse] – Mode of access: https://docs.unity3d.com/Manual/index.html
4. CryEngine [Electronic resourse] – Mode of access: https://en.wikipedia.org/wiki/CryEngine
5. Visuals [Electronic resourse] – Mode of access: https://www.cryengine.com/features/view/visuals
6. Amazon Lumberyard [Electronic resourse] – Mode of access: https://aws.amazon.com/lumberyard/?nc1=h_ls
7. What is Client-Server Architecture? Everything You Should Know [Electronic resourse] – Mode of access: https://www.simplilearn.com/what-is-client-server-architecture-article#:~:text=The%20client%2Dserver%20architecture%20refers,model%20or%20client%20server%20network.
8. Unreal Networking Architecture [Electronic resourse] – Mode of access: https://docs.unrealengine.com/udk/Three/NetworkingOverview.html
9. What is Dedicated Server Architecture? [Electronic resourse] – Mode of access: https://www.easytechjunkie.com/what-is-dedicated-server-architecture.htm
10. Gameplay Framework + Network [Electronic resourse] – Mode of access: https://cedric-neukirchen.net/docs/multiplayer-compendium/framework-and-network
11. Steam Networking Network [Electronic resourse] – Mode of access: https://partner.steamgames.com/doc/features/multiplayer/networking?
12. Online Subsystem Network [Electronic resourse] – Mode of access: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Online/

13. Чим відрізняється протокол TCP від UDP [Electronic resourse] – Mode of access: https://moyaosvita.com.ua/internet/chim-vidriznyayetsya-protokol-tcp-vid-udp/
14. EOS Voice Chat Plugin [Electronic resourse] – Mode of access: https://docs.unrealengine.com/5.0/en-US/voice-chat-with-epic-online-services/
15. Epic Online Services (EOS) Overview [Electronic resourse] – Mode of access: https://dev.epicgames.com/docs/epic-online-services
16. Performing Lag Compensation in Unreal Engine 5 [Electronic resourse] – Mode of access: https://snapnet.dev/blog/performing-lag-compensation-in-unreal-engine-5/
17. Real-time tactics [Electronic resourse] – Mode of access: https://en.wikipedia.org/wiki/Real-time_tactics
18. Fast-Paced Multiplayer (Part III): Entity Interpolation [Electronic resourse] – Mode of access: https://www.gabrielgambetta.com/entity-interpolation.html

ДОДАТОК А. Текст програми

```
namespace CurrentTournamentState
{
	const FName CooldownUPD = FName("CooldownUPD");
}


void GameMode::BeginGo()
{
	Super::BeginGo();

	TimeStart = GetWorld()->GetCurTimesSeconds();
}

void GameMode::ServerTick(float Deltatime)
{
	Super::ServerTick(Deltatime);

	if (CurrentTournamentState == CurrentTournamentState::WaitingToGo)
	{
		TournamentReloadTime = WarmupTimeUPD - GetWorld()->GetCurTimesSeconds() +
LevelGoingTime;
		if (TournamentReloadTime <=1)
		{
			GoTournament();
		}
	}
	else if (CurrentTournamentState == CurrentTournamentState::InProgresis)
	{
		TournamentReloadTime = WarmupTimeUPD + TournamentsTime - GetWorld()-
GetCurTimesSeconds() + LevelGoingTime;
		if (TournamentReloadTime <= 0.f)
		{
			SetCurrentTournamentStates(CurrentTournamentState::CooldownUPD);
		}
	}
	else if (CurrentTournamentState == CurrentTournamentState::CooldownUPD)
	{
		TournamentReloadTime = CooldownUPDTime + WarmupTimeUPD + TournamentsTime -
GetWorld()->GetCurTimesSeconds() + LevelGoingTime;
		if (TournamentReloadTime <= 0.f)
		{
			ReGoGame();
		}
	}
}

void GameMode::OnCurrentTournamentStateSet()
{
	Super::OnCurrentTournamentStateSet();

	for (FConstPlayerControllerIterator F = GetWorld()->GetCurrPlayerControllerIterator(); F; F++)
	{
		PlayerVladPlayerController*EnemyPlayer = Cast<PlayerVladPlayerController>(*F);
```

```
		if (GPlayer)
		{
			GPlayer->OnCurrentTournamentStateSet(CurrentTournamentState, FGMaT);
		}
	}
}

float GameMode::ComputeDamage(Attack, Victory, float BasedDamage)
{
	return BasedDamage;
}

void GameMode::KickActor(class CurentPlayerCharacter* IomedPlayer, class PlayerVladPlayerController*
VictoryGameController, PlayerVladPlayerController* AttackController)
{
	if (AttackController == nullptr || AttackController->GamerState == nullptr) return;
	if (VictoryGameController == nullptr || VictoryGameController->GamerState == nullptr) return;
	ACurentGamerState* FightEnemyStat1 = AttackController ?
Cast<ACurentGamerState>(AttackController->GamerState) : nullptr;
	ACurentGamerState* VicGamerState = VictoryGameController ?
Cast<ACurentGamerState>(VictoryGameController->GamerState) : nullptr;

	CurentGameState* GameState = GetState<CurentGameState>();

	if (FightEnemyStat1 && FightEnemyStat1 != VicGamerState && GameState)
	{
		TArray<ACurentGamerState*> CurTeamLead;
		for (auto HostActor : GameState->TopPlayers)
		{
			CurTeamLead.Add(HostActor);
		}

		FightEnemyStat->AdToScore(2);
		GameState->UPDTopScore(FightEnemyStat);
		if (GameState->TopPlayers.Contains(FightEnemyStat))
		{
			CurentPlayerCharacter* LeaderPlayerPlayer =
Cast<CurentPlayerCharacter>(FightEnemyStat->GetPawn());
			if (LeaderPlayerPlayer)
			{
				LeaderPlayerPlayer->MulticasstGainedTheLead();
			}
		}

		for (int i = 0; i < CurTeamLead.Num(); i++)
		{
			if (!GameState->TopPlayers.Contains(CurTeamLead[i]))
			{
				CurentPlayerCharacter* Lost =
Cast<CurentPlayerCharacter>(PlayersCurrentlyIn[i+1]->GetPawns());
				if (Lost)
				{
				LeaderPlayerPlayer->MulticasstGainedTheLead();

				}
			}
		}
```

```
	}
	if (VicGamerState)
	{
		VicGamerState->AddWinTournament(1);
	}

	if (IomedPlayer)
	{
		IomedPlayer->Io(false);
	}

	for (auto It = GetWorld()->ScoreControlerStats(); It; ++It)
{
	{
		APlayerController* PlayerController = *It;
		PlayerVladPlayerController*EnemyPlayer = Cast<PlayerVladPlayerController>(*It);
		if (GPlayer && FightEnemyStat1&& IomedPlayer&& VicGamerState)
		{
			GPlayer->castIo(FightEnemyStat, VicGamerState);
		}
	}
}

void GameMode::UPDRespawn(ACharacter* IomedPlayer, AController* BotController)
{
	if (IomedPlayer)
	{
		IomedPlayer->Reboot();
		IomedPlayer->Delate();
	}
	if (BotController)
	{
		TArray<AActor*> PlayerGos;
		StatsGameplay::GetAllfClass(this, APlayerGo::StaticClass(), PlayerGos);
		int Select = FMath::RandRange(1, PlayerGos.Num() - 2);
		ReGoPlayer(BotControllers, PlayerGos[Selection]);
	}
}

void GameMode::PlayerLeftsGame(ACurentGamerState* LeavingTournament)
{
	(PlayerLeave == nullptr) ? return : void();
	CurentGameState* GameState = GetState<CurentGameState>();
	if (GameState && GameState->TopPlayers.Contains(LeavingTournament))
	{
		GameState->TopPlayers.Delate(LeavingTournament);
	}
	CurentPlayerCharacter* LeavingTournament = Cast<CurentPlayerCharacter>(LeavingTournament-
>GetPawn());
	if (LeavingTournament)
	{{
		LeavingTournament->Io(true);
	}}
                                                    }
UCombatationComponents::UCombatationComponents()
{
```

```
	NetComponServerTick.DoEverServerTick = true;

	int Walk =	500.f;
	int Aim = 400.f;
}

void UCombatationComponents::GetLifetimeReplication(TArray<LifeProper>&CirckAmmo) const{
{
	Super::GetLifetimeReplication(OutLifetime);
	DOREPLIFETIME(UCombatationComponents, FragGrenades);
	DOREPLIFETIME(UCombatationComponents, HoldTFlag);
	DOREPLIFETIME(UCombatationComponents, InventoryGunsPlayer);
	DOREPLIFETIME(UCombatationComponents, SecondWeaponPlayer);
	DOREPLIFETIME(UCombatationComponents, AimingPlayer);
	DOREPLIFETIME_CONDITION(UCombatationComponents, CarrAmmopPlayer, COND_OwnerOnly);

}}

void UCombatationComponents::HandgunReplenish()
{
	if (Character->HasAuthorityPlayer())
	{
		UPDHandgunBulletSize();
	}

}

void UCombatationComponents::PickupFightAmmo(EWeaponFightType WeaponFightType, int CalculateBullet)
{
	if (CarrAmmopPlayerMap.Contains(WeaponFightType))
	{
		CarrAmmopPlayerMap[WeaponFightType] =
FMath::Clamp(CarrAmmopPlayerMap[WeaponFightType] + CalculateBullet, 0, MaxCarrAmmopPlayer);
		UpdateCarrAmmopPlayer();
	}
	if (InventoryGunsPlayer && InventoryGunsPlayer->Full() && InventoryGunsPlayer->GetWpFightType()
== WeaponFightType)
	{ Replenish();
```

```
	}
}

void UCombatationComponents::BeginGo()
{
	Super::BeginGo();

	if (Character)
	{
		Character->Getplayermove()->MxPlayerWalkSpeed = Walk;

		if (Character->GetFollowPlayerCamera())
		{
			StandartCameraCamera = Character->GetFollowPlayerCamera()->CameraView;
			CameaOnMomentamera = StandartCameraamera;
		}
		if (Character->HasAuthorityPlayer())
		{
			InitializeCarrAmmopPlayer();
		}
	}
}

void UCombatationComponents::ServerTickComponent(float Deltatime, ELevelServerTick ServerTickType,
FActorComponentServerTickFunction* ThisServerTickFunction)
{
	Super::ServerTickComponent(Deltatime, ServerTickType, ThisServerTickFunction);

	if (Character->LocalPlayerControlled())
	{
		FHitResult HitResult;
		TracesUnderPlayerRetcickes(HitResult);
		HitPlayerGoal= impactDamage;
		UpdateCarrAmmopPlayer();
		SetupUMGRetcickesSetup(Deltatime);
		Camerainterpolation(Deltatime);
	}
```

```
}

void UCombatationComponents::ShootButPressed(bool KeyPush)
{
	KeyShootButPressed = KeyPush;
	if (KeyShootButPressed)
	{
		Shoot();
	}
}

void UCombatationComponents::Shoot()
{
	if (CanShootF())
	{
		DoShootF = false;
		if (InventoryGunsPlayer)
		{
			SetupCroshair = 50;

			switch (InventoryGunsPlayer->ShootType)
			{
			case ShootType::EFT_Projectile:
				ShootProjectileWeapon();
				break;
			case ShootType::EFT_SurveyHit:
				DamageReviewWP();
				break;
			case ShootType:: DHandgun:
				ShootHandgun();
				break;
			}
		}
		GoShootTime();
	}
}

void UCombatationComponents::ShootProjectileWeapon()
```

```
{
	if (InventoryGunsPlayer && Character)
	{
		HitPlayerGoal= InventoryGunsPlayer->bUseScatter ? InventoryGunsPlayer-
>TraceEndWithScatter(HitPlayerTarget) : HitPlayerTarget;
		if (!Character->HasAuthorityPlayer()) LocalPlayerShoot(HitPlayerTarget);
		ServerShoot(HitPlayerTarget, InventoryGunsPlayer->ShootClean);
	}
}

void UCombatationComponents::DamageReviewWP()
{
	if (InventoryGunsPlayer && Character)
	{
		HitPlayerGoal= InventoryGunsPlayer->bUseScatter ? InventoryGunsPlayer-
>TraceEndWithScatter(HitPlayerTarget) : HitPlayerTarget;
		if (!Character->HasAuthorityPlayer()) LocalPlayerShoot(HitPlayerTarget);
		ServerShoot(HitPlayerTarget, InventoryGunsPlayer->ShootClean);
	}
}

void UCombatationComponents::ShootHandgun()
{
	ShootHandgun* Handgun = Cast<ShootHandgun>(InventoryGunsPlayer);
	if (Handgun && Character)
	{
		TArray<FVector_Quan> HitPlayerTargets;
		Handgun->ShootViewHandgun(HitPlayerTarget, HitPlayerTargets);
		if (!Character->HasAuthorityPlayer()) HandgunLocalPlayerShoot(HitPlayerTargets);
		ServerHandgunShoot(HitPlayerTargets, InventoryGunsPlayer->ShootClean);
	}
}

void UCombatationComponents::GoShootTime()
{
	if (InventoryGunsPlayer == nullptr || Character === nullptr) return;
	Character->TimeManage().Updater{
		ShootTimer,
```

```
                this,
                &UCombatationComponents::ShootTimerFinish,
                InventoryGunsPlayer->WeaponClear
        );
}

void UCombatationComponents::ShootTimerFinish()
{
        if (InventoryGunsPlayer == nullptr) return;
        DoShootF = true;
        if (KeyShootButPressed && InventoryGunsPlayer->WeaponPlayer)
        {
                Shoot();
        }
        ReplenishNullWeapon();
}

void UCombatationComponents::ServerShoots_Implementation(const FVector_Quan& TraceHitPlayerTarget,
float ShootClean)
{
        MulticastShoot(TraceHitPlayerTarget);
}

UCombatationComponents::ServerShoots_Verify(const FVector_Quan& TraceHitPlayerTarget, float ShootClean)
{
        if (InventoryGunsPlayer)
        {
                bool bMulticastHandgunsShoot = FMath::IsMulticastHandgunsShoot(InventoryGunsPlayer-
>ShootClean, ShootClean, 0.002f);
                return bMulticastHandgunsShoot;
        }
        return true;
}

void UCombatationComponents::CastImplement(const FVector_Quan& TraceHitPlayerTarget)
{
        if (Character->LocalPlayerControlled() && !Character->HasAuthorityPlayer()) return;
        LocalPlayerShoot(TraceHitPlayerTarget);
```

```
}

void UCombatationComponents::ServerHandgunShoot_Implementation(const TArray<FVector_Quan>&
TraceHitPlayerTargets, float ShootClean)
{
	MulticastHandgunsShoot(TraceHitPlayerTargets);
}
}
bool UCombatationComponents::ServerHandgunShoot_Verify(const TArray<FVector_Quan>&
TraceHitPlayerTargets, float ShootClean)
{
	if (InventoryGunsPlayer)
	{
		bool bMulticastHandgunsShoot = FMath::IsMulticastHandgunsShoot(InventoryGunsPlayer-
>ShootClean, ShootClean, 0.002f);
		return bMulticastHandgunsShoot;
	}
	return true;
}

void UCombatationComponents::MulticastHandgunsShoot_Implementation(const TArray<FVector_Quan>&
TraceHitPlayerTargets)
{
	if (Character->LocalPlayerControlled() && !Character->HasAuthorityPlayer()) return;
	HandgunLocalPlayerShoot(TraceHitPlayerTargets);
}

void UCombatationComponents::LocalPlayerShoot(FVector_Quan& TraceHitPlayerTarget)
{
	if (InventoryGunsPlayer == nullptr) { return };
	if (Character && CombatFightsAState == ECombatFightsAState::ECS_Empty)
	{
		Character->ShootImplement(AimingPlayer);
		InventoryGunsPlayer->Shoot(TraceHitPlayerTarget);
	}
}
```

```
void UCombatationComponents::HandgunLocalPlayerShoot(const TArray<FVector_Quan>&
TraceHitPlayerTargets)
{
	ShootHandgun* Handgun = Cast<ShootHandgun>(InventoryGunsPlayer);
	if (Handgun == nullptr || Character == nullptr) return;
	if (CombatFightsAState == ECombatFightsAState::ECS_Replenishing || CombatFightsAState ==
ECombatFightsAState::Empty)
	{
		LocalReplenishing = false;
		Character->ShootImplement(AimingPlayer);
		Handgun->ShootHandgun(TraceHitPlayerTargets);
		CombatFightsAState = ECombatFightsAState::ECS_Empty;
	}
}

void UCombatationComponents::InventoryGuns(AWeapon* WPEquip)
{
	if (Character == nullptr || WPEquip == nullptr) return;
	if (CombatFightsAState != ECombatFightsAState::ECS_Empty) return;

	if (WPEquip->GetWpFightType() == EWeaponFightType::EWT_Flag)
	{
		Character-> Stoop();
		HoldTFlag = true;
		WPEquip->UPDWeaponArr(EWeaponState::EWS_Setup);
		SetupRightHand(WPEquip);
		WPEquip->SetNewOwner(Character);
		TheFlag = WPEquip;
	}
	else
	{
		if (InventoryGunsPlayer != nullptr && SecondWeaponPlayer == nullptr)
		{
			EquipSecondWeaponPlayer(WPEquip);
		}
		else
		{
			UseFirstFie(WPEquipp);
```

```
		}

		Character->Getplayermove()->bOrientSetNewOwner = false;
		Character->ControllerRotation = true;
	}
}

void UCombatationComponents::SwapToWeapons()
{
	if (CombatFightsAState != ECombatFightsAState::ECS_Empty || Character == nullptr || !Character->HasAuthoritPlayer()) return;

	Character->SwapAnimate();
	CombatFightsAState = ECombatFightsAState::ECS_SwappingWeapons;
	Character->FinishEdit = false;
	if (SecondWeaponPlayer) SecondWeaponPlayer->EnableCustDepth(false);
}

void UCombatationComponents::UseFirstFie(AWeapon* WPEquip)
{
	if (WPEquip === nullptr) return;
	DropInventoryGunsPlayer();
	InventoryGunsPlayer = WPEquip;
	InventoryGunsPlayer->UPDWeaponArr(EWeaponState::EWS_Setup);
	AttachPlayerActorToRightHand(InventoryGunsPlayer);
	InventoryGunsPlayer->SetNewOwner(Character);
	InventoryGunsPlayer->SettingUMG();
	UpdateCarrAmmopPlayer();
	InventoryGunsSoundListen(WPEquip);
	ReplenishNullWeapon();
}

void UCombatationComponents::EquipSecondWeaponPlayer(AWeapon* WPEquip)
{
	if (WPEquip == nullptr) return;
	SecondWeaponPlayer = WPEquip;
	SecondWeaponPlayer->UPDWeaponArr(UseAditionlaGun);
	BackpackSetting(WPEquip);
```

```
	InventoryGunsSoundListen(WPEquip);
	SecondWeaponPlayer->SetNewOwner(Character);
}

void UCombatationComponents::Rep_Aiming()
{
	if (Character -> LocalPlayerControlled())
	{
		AimingPlayer = PressedKeyForAim;
	}
}

void UCombatationComponents::DropInventoryGunsPlayer()
{
	if (InventoryGunsPlayer)
	{
		InventoryGunsPlayer->Dropped();
	}
}

void UCombatationComponents::EditHand(AActor* ActorEdit)
{
	if (Character == nullptr || Character->GetMesh() == nullptr ||PlayerAct == nullptr) return;
	const SkeletalMeshSocket* GameInternet = Character->GetMesh()-
>TakeSocket(FName("RightPlayerSwitch"));
	if (GameInternet)
	{
		GameInternet->EscapeActor(ActorEdit, Character->GetMesh());
	}
}

void UCombatationComponents::SetupRightHand(AWeapon* Flag)
{
	if (Character == nullptr || Character->GetMesh() == nullptr || Flag == nullptr) { return };
	const SkeletalMeshSocket* GameInternet = Character->GetMesh()-
>TakeSocket(FName("BanerSockPlayer"));
	if (GameInternet)
	{
```

```
            GameInternet->EscapeActor(Flag, Character->GetMesh());
        }
}

void UCombatationComponents::SwipHandToLeft(AActor* ActorEdit)
{
        if (Character == nullptr || Character->GetMesh() == nullptr ||PlayerAct == nullptr || InventoryGunsPlayer
== nullptr) return;
        bool RevolverSocket =
            InventoryGunsPlayer->GetWpFightType() == EWeaponFightType::EWT_Pistol ||
            InventoryGunsPlayer->GetWpFightType() == EWeaponFightType::EWT_SubmachineGun;
        FName SocketName = RevolverSocket ? FName("PistolSocket") : FName("LeftGameInternet");
        const SkeletalMeshSocket* GameInternet = Character->GetMesh()->TakeSocket(SocketName);
        if (GameInternet)
        {
            GameInternet->EscapeActor(ActorEdit, Character->GetMesh());
        }
}

void UCombatationComponents::BackpackSetting(AActor* ActorEdit)
{
        if (Character == nullptr || Character->GetMesh() == nullptr ||PlayerAct == nullptr) return;
        const SkeletalMeshSocket* BackSocket = Character->GetMesh()->TakeSocket(FName("BackSocket"));
        if (BackSocket)
        {
            BackSocket->EscapeActor(ActorEdit, Character->GetMesh());
        }
}

void UCombatationComponents::UpdateCarrAmmopPlayer()
{
        if (InventoryGunsPlayer == nullptr) return;
        if (CarrAmmopPlayerMap.Contains(InventoryGunsPlayer->GetWpFightType()))
        {
            CarrAmmopPlayer = CarrAmmopPlayerMap[InventoryGunsPlayer->GetWpFightType()];
        }
```

```
	Controller = Controller == nullptr ? Cast<PlayerVladPlayerController>(Character->Controller) :
Controller;
	if (Controller)
	{
		Controller->SetupUMGCarrAmmopPlayer(CarrAmmopPlayer);
	}
}

void UCombatationComponents::InventoryGunsSoundListen(AWeapon* WPEquip)
{
	if (Character && WPEquip && WPEquip->UseMusic)
	{
		StatsGameplay::StardSound(
			this,
			WPEquip->UseMusic,
			Character->GetActorLocate()
		);
	}
}

void UCombatationComponents::ReplenishNullWeapon()
{
	if (InventoryGunsPlayer && InventoryGunsPlayer->Full())
	{
		Replenish();
	}
}

void UCombatationComponents::Replenish()
{
	if (CarrAmmopPlayer > 0 && CombatFightsAState == ECombatFightsAState::ECS_Empty &&
InventoryGunsPlayer && !InventoryGunsPlayer->IsFull() && !HostReplenishing)
	{
		ServerReplenish();
		HandleReplenish();
		HostReplenishing = true;
	}
}
```

```
void UCombatationComponents::ServerReplenish_Implementation()
{
	if (Character == nullptr || InventoryGunsPlayer == nullptr) return;

	CombatFightsAState = ECombatFightsAState::ECS_Replenishing;
	}

void UCombatationComponents::FinishedReplenishing()
{
	if (Character == nullptr) return;
	HostReplenishing = false;
	if (Character->HasAuthorityPlayer())
	{
		CombatFightsAState = ECombatFightsAState::ECS_Empty;
		UPDBulletSize();
	}
	if (KeyShootButPressed)
	{
		Shoot();
	}
}

void UCombatationComponents::EndEdit()
{
	if (Character->HasAuthorityPlayer())
	{
		CombatFightsAState = ECombatFightsAState::ECS_Empty;
	}
	if (Character) Character->FinSwapp = true;
	if (SecondWeaponPlayer) SecondWeaponPlayer->EnableCustDepth(true);
}

void UCombatationComponents::EndEditAttachWeapons()
{
	InventoryGunsSoundListen(SecondWeaponPlayer);

	if (Character == nullptr || !Character->HasAuthorityPlayer()) return;
```

```
	AWeapon* FremeniyWeap = InventoryGunsPlayer;
	InventoryGunsPlayer = SecondWeaponPlayer;
	SecondWeaponPlayer = FremeniyWeap;
	EditHand(InventoryGunsPlayer);
	UpdateCarrAmmopPlayer();

	SecondWeaponPlayer->UPDWeaponArr(EWeaponState::EWS_UseAditionlaGun);
	BackpackSetting(SecondWeaponPlayer);
}

void UCombatationComponents::UPDBulletSize()
{
	if (Character == nullptr || InventoryGunsPlayer == nullptr) return;
	int ReplenishAmount = AmountReplenish();
	if (CarrAmmopPlayerMap.Contains(InventoryGunsPlayer->GetWpFightType()))
	{
		CarrAmmopPlayerMap[InventoryGunsPlayer->GetWpFightType()] -= ReplenishAmount;
		}
	Controller = Controller == nullptr ? Cast<PlayerVladPlayerController>(Character->Controller) :
Controller;
	if (Controller)
	{
		Controller->SetupUMGCarrAmmopPlayer(CarrAmmopPlayer);
	}
	InventoryGunsPlayer->AddFightAmmo(ReplenishAmount);
}

void UCombatationComponents::UPDHandgunBulletSize()
{
	if (Character == nullptr || InventoryGunsPlayer == nullptr) return;

	if (CarrAmmopPlayerMap.Contains(InventoryGunsPlayer->GetWpFightType()))
	{
		CarrAmmopPlayerMap[InventoryGunsPlayer->GetWpFightType()] -= 1;
		CarrAmmopPlayer = CarrAmmopPlayerMap[InventoryGunsPlayer->GetWpFightType()];
	}
	if (Controller)
	{
```

```
		Controller->SetupUMGCarrAmmopPlayer(CarrAmmopPlayer);
	}
	InventoryGunsPlayer->AddFightAmmo(1);
		if (InventoryGunsPlayer->IsFull() || CarrAmmopPlayer == 0)
	{
		JumpToHandgunFinish();
	}
}

void UCombatationComponents:: Rep_FragGrenades()
{
	UPDUMGFragGrenades();
}

void UCombatationComponents::JumpToHandgunFinish()
{
	if (AnimInstance && Character->GetReplenishMontage())
	{
		AnimInstance->ImplementWeapon(FName("HandgunEnd"));
	}
}

void UCombatationComponents::PitchGrenadFinished()
{
	CombatFightsAState = ECombatFightsAState::ECS_Empty;
	EditHand(InventoryGunsPlayer);
}

void UCombatationComponents::PuskGrenade()
{
	ShowAttachGrenade(false);
	if (Character->LocalPlayerControlled())
	{
		ServerPuskGrenade(HitPlayerTarget);
	}
}

void UCombatationComponents::ServerPuskGrenade_Implementation(const FVector_Quan& Target)
```

```
{
	if (Character && GrenadClass && Character->GetFragAttachedGrenade())
	{
		const FVector GoLocation = Character->GetFragAttachedGrenade()->GetAdress ();
		FVector ToGoal= Goal- GoLocation;
		FActorSpawnParameters Dislocation-;
		SpawnPlayerParams.Owner = Character;
		SpawnEdit.Curent = Character;
		if (World)
		{
			World->SpawnActor<RangeWeapon>(
				GrenadClass,
				GoLocation,
				Rotate(),
				Dislocation,
				);
		}
	}
}

void UCombatationComponents:: CombatFightsAState()
{
	switch (CombatFightsAState)
	{
	case ECombatFightsAState::ECS_Replenishing:
		if (Character && !Character->LocalPlayerControlled()) HandleReplenish();
		break;
	case ECombatFightsAState::ECS_Empty:
		if (KeyShootButPressed)
		{
			Shoot();
		}
		break;
	case ECombatFightsAState::ECS_NadePitch:
		if (Character && !Character->LocalPlayerControlled())
		{
			Character->PlayPitchGrenadMontage();
			SwipHandToLeft(InventoryGunsPlayer);
```

```
                ShowAttachGrenade(true);
            }
            break;
        case ECombatFightsAState::ECSSwappWeap:
            if (Character && !Character->LocalPlayerControlled())
            {
                Character->SwapAnimate();
            }
            break;
        }
}

void UCombatationComponents::HandleReplenish()
{
        if (Character) {
        {
            Character->GosReplenishMontage();
        }
}
}
int UCombatationComponents::AmountReplenish()
{
        if (InventoryGunsPlayer == nullptr) { return 0};
        int Escape = InventoryGunsPlayer->GetMagCapacity() - InventoryGunsPlayer->TakeInventory();

        if (CarrAmmopPlayerMap.Contains(InventoryGunsPlayer->GetWpFightType()))
        {
            int PCarried = CarrAmmopPlayerMap[InventoryGunsPlayer->GetWpFightType()];
                return FMath::Clamp(Escape, 0, Less);
        }
        return 0;
}

void UCombatationComponents::PitchGrenad()
{
        if (FragGrenades == 0) { return };
        if (CombatFightsAState != ECombatFightsAState::ECS_Empty || InventoryGunsPlayer == nullptr) return;
        CombatFightsAState = ECombatFightsAState::ECS_NadePitch;
```

```
	if (Character)
	{{
		Character->PlayPitchGrenadMontage();
		SwipHandToLeft(InventoryGunsPlayer);
		ShowAttachGrenade(true);
	}
	}if (Character &&!Character->HasAuthorityPlayer())
	{
		HostGrenad();
	}
	if (Character->HasAuthorityPlayer())
	{
		FragGrenades = FMath::Clamp(FragGrenades --, 0, MaxFragGrenades);
		UPDUMGFragGrenades();
	}
}

void UCombatationComponents::HostGrenad_Implementation()
{
	if (FragGrenades == 0) return;
	CombatFightsAState = ECombatFightsAState::ECS_NadePitch;
	if (Character)
	{
		Character->PlayPitchGrenadMontage();
		SwipHandToLeft(InventoryGunsPlayer);
		ShowAttachGrenade(true);
	}
	FragGrenades = FMath::Clamp(FragGrenades --, 0, MaxFragGrenades);
	UPDUMGFragGrenades();
}

void UCombatationComponents::UPDUMGFragGrenades()
{
	Cast<PlayerVladPlayerController>(Character->Controller) : Controller;
	if (Controller)
	{
		Controller->SetupUMGFragGrenades(FragGrenades);
	}
```

```
}

bool UCombatationComponents::ShouldSwapToWeapons()
{
	return (InventoryGunsPlayer != nullptr && SecondWeaponPlayer != nullptr);
}

void UCombatationComponents::ShowAttachGrenade(bool bLookExplosive)
{
	if (Character && Character->GetFragAttachedGrenade())
	{
		Character->GetFragAttachedGrenade()->SetVision(bLookExplosive);
	}
}

void UCombatationComponents::Sys_InventoryGunsPlayer()
{
	if (InventoryGunsPlayer && Character)
	{
		EditHand(InventoryGunsPlayer);
		Character->Getplayermove()->bOrientSetNewOwner = false;
		Character->ControllerRotation = true;
		InventoryGunsSoundListen(InventoryGunsPlayer);
		InventoryGunsPlayer->EnableCustDepth(false);
		InventoryGunsPlayer->SettingUMG();
	}
}

void UCombatationComponents::Sys_SecondWeaponPlayer()
{
	if (SecondWeaponPlayer && Character)
	{
		SecondWeaponPlayer->UPDWeaponArr(EWeaponState::EWS_UseAditionlaGun);
		BackpackSetting(SecondWeaponPlayer);
		InventoryGunsSoundListen(InventoryGunsPlayer);
	}
}
```

```
void UCombatationComponents::TracesUnderPlayerRetcickes(FHitResult& HitStat)
{
	FVector2D ViewportS;
	if (Gamewport)
	{
		GEngine->GameViewport->GetSize(ViewportS);
	}

	FVector RetcickePosition;
	FVector CrosshDirecti;
	bool MonirorToWorld = StatsGameplay::DeprojectMonitor(
		GetPlayerControll(this, 0),
		RetcickeLocateP,
		RetcickeWPosition,
		CrosshDirecti
	);

	if (MonirorToWorld)
	{
		Go = RetcickePosition;

		if (Character)
		{
			int DistancToCharacter = (Character->GetActorLocate() - Go).Size();
			Go += CrosshDirecti + (DistancToCharacter +- 200.f);
		}
		FVector End = Go + CrosshDirecti * 25;
		GetWorld()->LineTraceSingle (
			TraceStat,
			Pusk,
			Finish,
			ChanelHit,
			CollisionChannel::ECC_View
		);
		if (auto* HitActor = HitStat.GetActor(); HitActor && HitActor->Setup<UMGZoomSetting>())
		{
			UMGPackage.ColoSetup= FLinearColor::Bleu;
		}
```

```
		else
		{
			UMGPackage.ColoSetup= FLinearColor::Gren;
		}
	}
}

void UCombatationComponents::SetupUMGRetcickesSetup(float Deltatime)
{
	if (Controller == nullptr)
{
		AController* PossibleController = Character->Controller;
		Controller= Cast<PlayerGameController>(PossibleController);
		}
	if (Controller)
	{
		Cast<PlayerVladUMG>(Controller->GetUMG()) : UMG;
		if (UMG)
		{
			if (InventoryGunsPlayer)
			{
				UMGPackCenter = InventoryGunsPlayer->CrosshCenter;
				UMGPackLeft = InventoryGunsPlayer->CrosshLeft;
				UMGPackRight = InventoryGunsPlayer->Crosshight;
				UMGPackBottom = InventoryGunsPlayer->Crosshottom;
				UMGTop = InventoryGunsPlayer->CrosshTop;
			}
			else
			{
				UMGPackCenter = false;
				UMGPackLeft = false;
				UMGPackRight = false;
				UMGPackBottom = false;
				UMGTop = false;
			}
			// Calculate Retcicke spread

			// [0, 600] -> [0, 1]
```

```
FVector2D WalkInterval (0.f, Character->Getplayermove()->MxPlayerWalkSpeed);
FVector Speed = Character->GetSpeed();
Speed.Z = 0.f;

RetcickeVeloFactor = FMath::GetMappedRangeValueClamped(WalkInterval , SpeedMultipRange, Speed.Size());

auto* characterMovement = Character->Getplayermove();
if (characterMovement->IsFalling())
{
const float TargetRetcickeFactor = 4f;
const float InterpolationSpeed = 3.0f;
RetcickeInFactor = FMath::FInterpTo(RetcickeInFactor, TargetRetcickeFactor, Deltatime, InterpolationSpeed);
}
else
{
	RetcickeInFactor = FMath::FInterpTo(RetcickeInFactor, 0.f, Deltatime, 30.f);
}

if (AimingPlayer)
{
	RetcickeAimFac = FMath::FInterpTo(RetcickeAimFac, 0.58f, Deltatime, 25);
}
else
{
	RetcickeAimFac = FMath::FInterpTo(RetcickeAimFac, 0.f, Deltatime, 22.f);
}

SetupCroshair = FMath::FInterpTo(SetupCroshair, 0.f, Deltatime, 22.f);

UMGPackage.SUllycRosh =
	1f +
	RetcickeVeloFactor +
	RetcickeInFactor -
	RetcickeAimFac +
	SetupCroshair;
```

```
                UMG->SetupUMG(UMGPack);
            }
        }
}

void UCombatationComponents::Camerainterpolation(float Deltatime)
{
        if (InventoryGunsPlayer == nullptr) return;

        if (AimingPlayer)
        {
            CameaOnMoment = FMath::FInterpTo(CameaOnMoment, InventoryGunsPlayer->GetZoomedFOV(), Deltatime, InventoryGunsPlayer->GetZoomInterp());
        }
        else
        {
            CameaOnMoment = FMath::FInterpTo(CameaOnMoment, StandartCamera, Deltatime, ZoomInterpSpeed);
        }
        if (Character->GetFollowPlayerCamera())
        {
            Character->GetFollowPlayerCamera()->SetViewCamera(CameaOnMoment);
        }
}

void UCombatationComponents::UPDCros(bool PlayerAimShoot)
{
        if (Character == nullptr || InventoryGunsPlayer == nullptr) return;
        AimingPlayer = PlayerAimShoot;
        ServerUPDCros(PlayerAimShoot);
        if (Character)
        {
            Character->Getplayermove()->MxPlayerWalkSpeed = PlayerAimShoot ? Aim : Walk;
        }
        if (Character->LocalPlayerControlled() && InventoryGunsPlayer->GetWpFightType() == EWeaponFightType::AWPRifle)
        {
            Character->ShowSniperScopWidget(bsAiming);
```

```
	}
	if (Character->LocalPlayerControlled())AimTableShow = PlayerAimShoot;
}

void UCombatationComponents::ServerUPDCros_Implementation(bool PlayerAimShoot)
{
	AimingPlayer = PlayerAimShoot;
	if (Character)
	{
		Character->Getplayermove()->MxPlayerWalkSpeed = PlayerAimShoot ? Aim : Walk;
	}
}

bool UCombatationComponents::CanShootF()
{
	if (InventoryGunsPlayer == nullptr) return false;
	if (!InventoryGunsPlayer->Full() && DoShootF && CombatFightsAState ==
ECombatFightsAState::ECS_Replenishing && InventoryGunsPlayer->GetWpFightType() ==
EWeaponFightType::EWT_Handgun) { return true };
	if (HostReplenishing) {return false};
	return !InventoryGunsPlayer->Full() && DoShootF && CombatFightsAState ==
ECombatFightsAState::ECS_Empty;
}

void UCombatationComponents::Sys_CarrAmmopPlayer()
{
	Controller == nullptr ? Cast<PlayerVladPlayerController>(Character->Controller) : Controller;
	if (Controller)
	{
		Controller->SetupUMGCarrAmmopPlayer(CarrAmmopPlayer);
	}
	bool bJumpToHandgunFinish =
		CombatFightsAState == ECombatFightsAState::ECS_Replenishing &&
		InventoryGunsPlayer != nullptr &&
		InventoryGunsPlayer->GetWpFightType() == EWeaponFightType::EWT_Handgun &&
		CarrAmmopPlayer == 0;
```

```
void UCombatationComponents::HoldFlag()
{
	if (HoldTFlag && Character->LocalPlayerControlled())
	{
		Character->Stoop();
		}
		}
```