МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Сумський державний університет
Центр заочної, дистанційної та вечірньої форм навчання

«До захисту допущено»
В.о. завідувача кафедри

___________ Ігор ШЕЛЕХОВ
(підпис)

23 січня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 - Комп'ютерних наук,

освітньо-наукової програми «Інформатика»

на тему: «Інформаційна технологія проектування мережевої ігрової системи на основі Unreal Engine 5»

здобувача групи ІН.мз-22с Авраменка Нікіти Олексійовича

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

_____________ Нікіта АВРАМЕНКО
(підпис)

Доцент кафедри комп'ютерних наук,
кандидат технічних наук, доцент Віктор АВРАМЕНКО _____________
(підпис)

Суми – 2024

Сумський державний університет
Центр заочної, дистанційної та вечірньої форм навчання

«Затверджую»
В.о. завідувача кафедри
____________ Ігор ШЕЛЕХОВ
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми «Інформатика»

здобувача групи ІН.мз-22с Авраменка Нікіти Олексійовича

1. Тема роботи: «Інформаційна технологія проектування мережевої ігрової системи на основі Unreal Engine 5»
затверджую наказом по СумДУ від *«20» листопада 2023 року № 1307-VI*

2. Термін здачі здобувачем кваліфікаційної роботи *до 25 січня 2024 року*

3. Вхідні дані до кваліфікаційної роботи

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналіз проблеми предметної області, постановка й формування завдань дослідження.
2) Огляд типів мережевого підключення.
3) Розробка мережевої ігрової системи.
4) Аналіз результатів.

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

6. Консультанти до проекту (роботи), із зазначенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» ________________ 20 ___ р.

Завдання прийняв до виконання ________________ Керівник ________________
(підпис) (підпис)

АНОТАЦІЯ

Записка: 39 стр., 29 Рисунок, 1 додаток, 20 використаних джерел.

Обґрунтування актуальності теми роботи – Тема кваліфікаційної роботи є актуальною, оскільки присвячена розробці ігрової системи з мережевим підключенням, що є домінуючою на ринці комп'ютерних ігор.

Об'єкт дослідження — процес одночасної гри з різних пристроїв.

Мета роботи — розробка інформаційної технології проектування ігрової системи з можливістю гри в мережі.

Методи дослідження — система клієнт-сервер, система реплікації, методи обміну даними.

Результати — створена інформаційна технологія проектування ігрової системи з можливістю гри в мережі на основі Unreal Engine 5

ІНФОРМАЦІЙНА СИСТЕМА, РЕПЛІКАЦІЯ, КЛІЄНТ-СЕРВЕР, PEER-TO-PEER, LISTEN SERVER, DEDICATED SERVER, C++, UNREAL ENGINE 5, BLUEPRINT, ОНЛАЙН-СЕСІЯ.

ЗМІСТ

ВСТУП 5
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 6
1.1 Як працює багатокористувацький режим 6
1.2 Види підключення 7
1.3 Види мультиплеєру 9
1.4 Огляд інструментів 10
1.5 Постановка задачі 11
2 ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ 12
2.1 Створення проекту 12
2.2 Створення механіки нанесення ударів 13
2.3 Відтворення дій гравця у інших клієнтів 20
2.4 Створення платформи, що рухається 21
2.5 Створення та приєднання до онлайн-сесій 22
3 АНАЛІЗ РЕЗУЛЬТАТІВ 25
ВИСНОВКИ 27
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 28
ДОДАТКИ **31**

ВСТУП

Актуальність. Створення мультиплеєрних ігор є надзвичайно актуальним і важливим напрямком в індустрії відеоігор. У нинішню епоху глобалізації та високошвидкісного інтернет-підключення геймінг став соціальною та захоплюючою активністю. Щодо економічного аспекту, мультиплеєрні ігри отримують більшу частину прибутків на ринку, в основному завдяки мікротранзакціям. Також конкуренція спонукає розробників відшукувати нові способи привернення уваги користувачів, створювати унікальний ігровий досвід. Враховуючи ці фактори, режим мультиплеєру є досить привабливим для ігрових розробників

Об'єкт дослідження. Режим мультиплеєру в ігровій системі, тобто можливість одночасної гри декількох користувачів з різних пристроїв.

Предмет дослідження. Система реплікації в ігровому рушії Unreal Engine 5

Структура. Дана робота складається зі вступу, аналізу поняття багатокористувацького режиму та його видів, огляду інструментів, опису програмного забезпечення інформаційної системи, висновків, списку використаних джерел та додатків.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Режим мультиплеєру в іграх(багатокористувацький режим) - режим гри, завдяки якому одразу декілька користувачів можуть грати в одну і ту ж гру, впливаючи на ігровий процес один одного.

1.1 Як працює багатокористувацький режим

Режим мультиплеєру діє по моделі “Клієнт-Сервер”.

Клієнт - программа запущена самим гравцем, яка отримує дані з сервера та відтворює гру на пристрої користувача у такому вигляді, в якому вона перебуває на сервері. Відповідає за відображення графіки та інтерфейсу, обробляє натиснуті клавіші та передає дані на сервер, також відповідає за відтворення візуальних ефектів, анімацій персонажів та аудіо[1].

Сервер - інша сутність моделі, на якій саме і існує гра. На сервері мають виконуватися самі основні для геймплею частини коду. Отримує дані від клієнтів, обробляє їх та визначає новий стан гри[1].

Приклад роботи моделі: гравець натискає кнопку руху, дані про це передаються на сервер. Сервер отримує координати переміщення, прораховує, чи може гравець взагалі туди переміститися. Якщо може, то сервер переміщує копію гравця, що існує в ньому, і повертає дані про нове положення клієнту-відправнику. Одночасно з цим, дані про нове положення гравця клієнту, що перемістився, передаються і на всі інші клієнти, на яких цей гравець також переміститься. Подібний обмін даними між сервером та клієнтами відбувається постійно, кожного кадру.

Клієнт лише надсилає дані про натиснуті клавіші, усі події прораховуються саме на сервері, котрий повертає дані про те, де має знаходитись той чи інший об’єкт, який показник здоров’я у персонажа, скільки часу до кінця матчу тощо. Клієнт тільки відтворює стан гри, що знаходиться на сервері, він не має доступу до змінних, що знаходяться на сервері. Зроблено це, щоб уникнути шахрайства від гравців. Так, навіть якщо гравець взломає гру у себе на пристрої і, наприклад, дасть можливість своєму персонажу літати, персонаж, що знаходиться на сервері не полетить, так же як і персонажі на інших клієнтах.

Дані між клієнтами і сервером передаються по протоколу TCP/IP. При передачі задля більшої безпеки, дані розбиваються на фрагменти і потім інкапсулюються у пакети, бо втратити при передачі цілий файл гірше, ніж один чи кілька пакетів.

1.2 Види підключення

1. Peer-to-peer

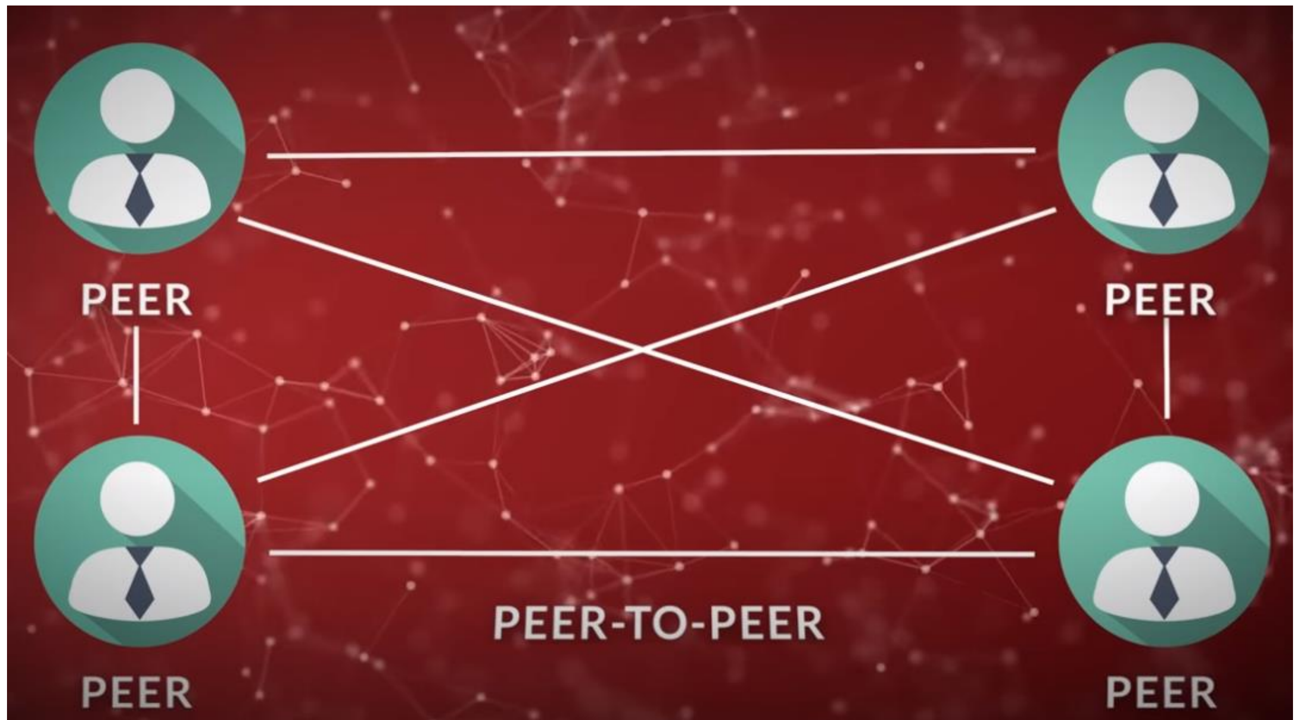

Рисунок 1.2.1

Система, в якій гравці з'єднуються напряму, без посередництва у вигляді центрального серверу. У такій мережі кожен гравець одночасно є і клієнтом, і сервером. При такому з'єднанні затримка при обміні даними зменшується, однак ці мережі можуть бути вразливими до атак і не завжди ефективно вирішують проблеми безпеки, такі як шахрайство та хакінг. Також не підходить для з'єднання великої кількості гравців, через недостатню обчислювальну потужність пристроїв звичайних гравців[1].

2. Listen server

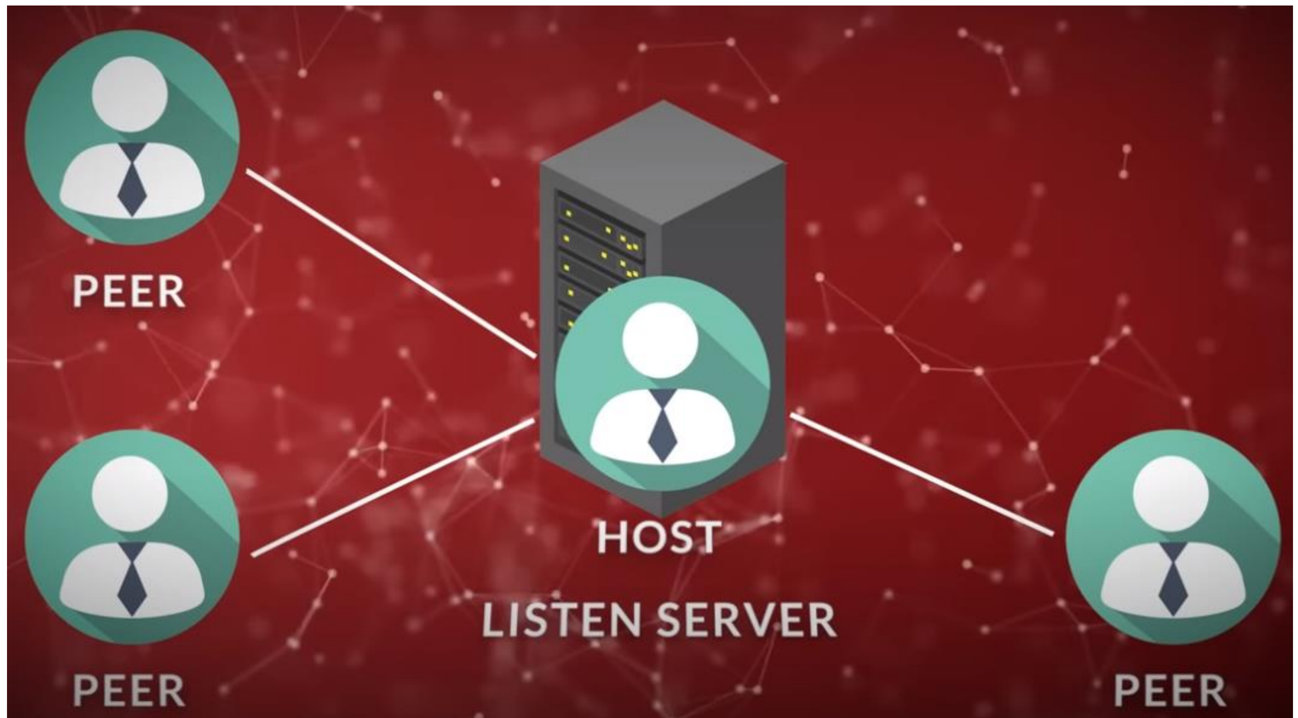

Рисунок 1.2.2

Тип підключення, при якому один із користувачів сам стає сервером, тобто хостом, який розраховує ігрову сесію для інших гравців. Використовується для ігор, у яких максимальна кількість гравців невелика, бо комп'ютер звичайного користувача не розрахований на обслуговування великої кількості запитів. При цьому затримка гри, тобто показник ping, у хоста буде дорівнювати нулю. Затримка ж у клієнтів буде залежати від того, наскільки далеко вони знаходяться від хоста. В іграх з таким підключенням, якщо гравець-господар втрачає з'єднання або виходить з гри, ігрова сесія може бути припинена для усіх гравців, або ж відбудеться host migration, тобто хостом стане інший клієнт. Для мобільних ігор таке підключення погано підійде, бо захостити сервер на телефонних пристроях навряд чи вийде[1].

3. Dedicated server

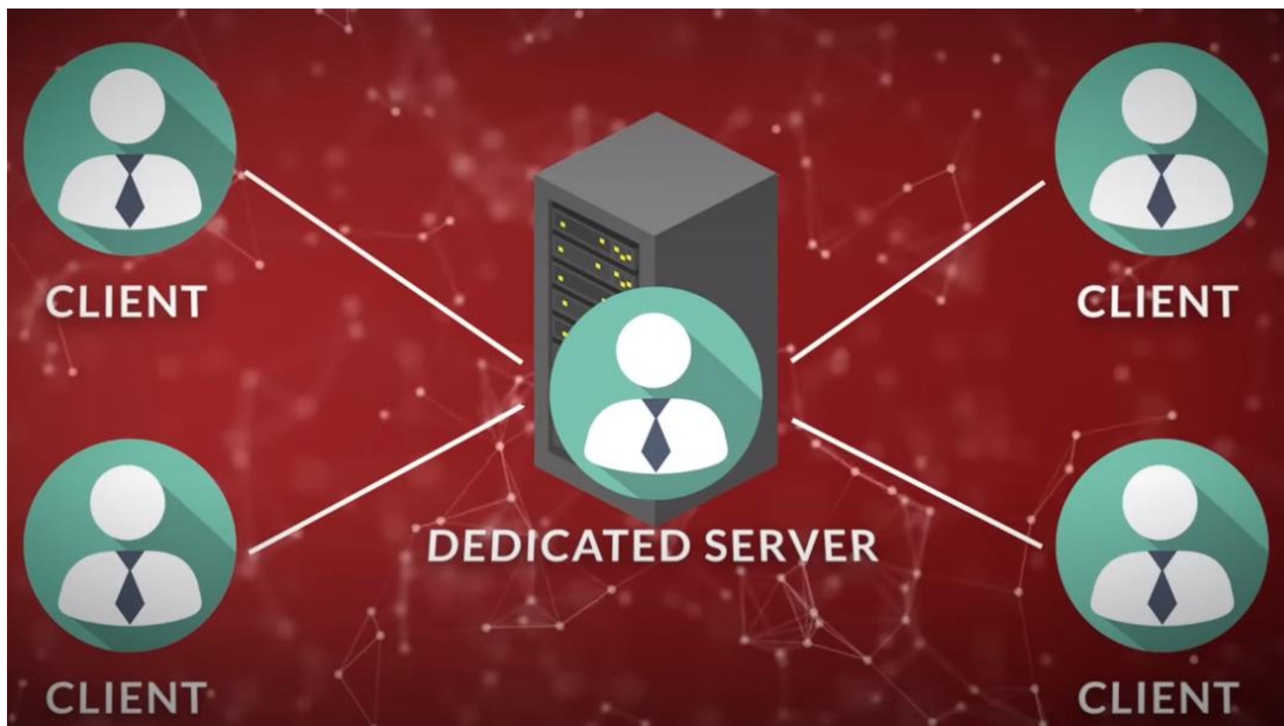

Рисунок 1.2.3

Система, в якій задіяно окремий сервер, усі ресурси якого задіяні для обслуговування ігрової сесії. Оскільки такі сервери призначені для обробки великих ігрових середовищ, то і кількість гравців значно більша, ніж при з’єднанні типу Listen server. Подібна система більш стійка і надійна, бо не залежить від хосту. Якщо будь-який гравець втратить з’єднання, ігрова сесія все одно продовжиться. Також усі ігрові дані зберігаються саме на сервері, що не дає змоги отримати доступу до них з клієнтів[1].

1.3 Види мультиплеєру

1. Async multiplayer - багатокористувацький режим, в якому гравці можуть впливати на ігровий процес інших, при цьому не беручи у ньому участі. Наприклад, у грі передбачено участь лише однієї людини, але у гравця може бути можливість залишати в локації гри повідомлення з “підказкою”, котре зможуть бачити інші гравці, а також можливість самому бачити такі повідомлення. Також, такий вид часто використовується у мобільних іграх: коли гравець змагається не з іншим користувачем, а з копією його ігрового персонажу.

2. Turn based multiplayer - режим, у якому вже безпосередньо присутні двоє або більше гравців. Проте пакети для оновлення даних на сервері відправляються

лише після закінчення ігрового ходу, а не кожного кадру. Для прикладу підійдуть звичайні онлайн-шахи або карточні ігри.

3. Real time multiplayer - найбільш поширений тип мультиплеєру. В ньому обмін даними відбувається кожного кадру.

1.4 Огляд інструментів

У ході дипломної роботи був використаний ігровий рушій Unreal Engine 5, через свою безкоштовність, доступність, гарний рівень графіки та інструментарій, зокрема для роботи з клієнт-серверною архітектурою, а саме системою реплікації. Один із підходів до реплікації в Unreal Engine 5 - це використання "Unreal Networking". Ця система дозволяє автоматично синхронізувати стани об'єктів та події між різними екземплярами гри на різних пристроях. Щоб реалізувати реплікацію в Unreal Engine 5, розробники можуть використовувати класи, такі як AActor та UActorComponent, і визначати, які дані повинні бути репліковані між клієнтом та сервером. Ніяких інших інструментів у ході розробки не було використано, анімації деяких рухів персонажів були скачані з інтернету[2].

Розробка на Unreal Engine може відбуватися у системі візуального скриптингу Blueprint, або на мові програмування C++. У обох опцій є свої переваги і недоліки.

Перевагами для системи Blueprint є простота в їх використанні, легкість читання коду, швидкість прототипування та вбудовані інструменти для графіки та анімації. Недоліками є те, що код на Blueprint повільніший за код на C++, через те, що при виконанні програми Blueprint конвертується у код C++, а на це потрібен час. Також Blueprint не має усього функціоналу C++, наприклад не можна через цю систему підключитися до бази даних[3].

Перевагами C++ у першу чергу є швидкодія, розширений функціонал. Проте і сама розробка стає складнішою: C++ складніший у вивченні, а код складніше сприймати, ніж написаний у системі Blueprint

Проте кращим варіантом буде використання одразу обох варіантів, так як файли Blueprint може наслідуватися від файлів C++[4]. Отже можна прописати складні функції у файлах C++, створити файл Blueprint, який буде наслідувати файл C++, і продовжити програмувати менш складну логіку у ньому.

1.5 Постановка задачі

Завданням даної роботи є створення інформаційної технології проектування ігрової системи з можливістю гри в мережі на основі Unreal Engine 5. Система має задовольняти наступним вимогам:

1) Гравець може створювати ігрову сесію, або підключатися до існуючої.
2) Персонаж гравця може рухатися, стрибати та бити.
3) Всі дії одного гравця відтворюються у всіх інших.
4) Ігрові об’єкти на рівні у всіх гравців знаходяться в одному й тому ж місці.

2 ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1 Створення проекту

Для початку створюється проект у Unreal Engine 5.

Так як розроблювана гра має мати камеру від третьої особи, за шаблон обирається ThirdPerson.

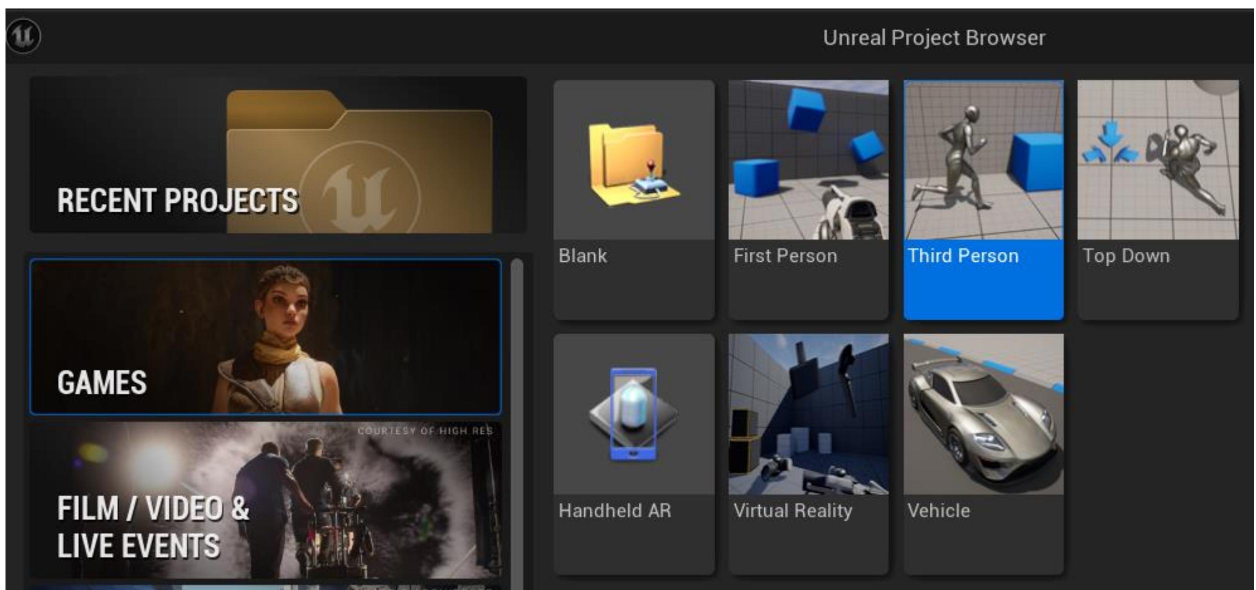

Рисунок 2.1.1

Далі встановлюються наступні налаштування і обирається директорія проекту[5].

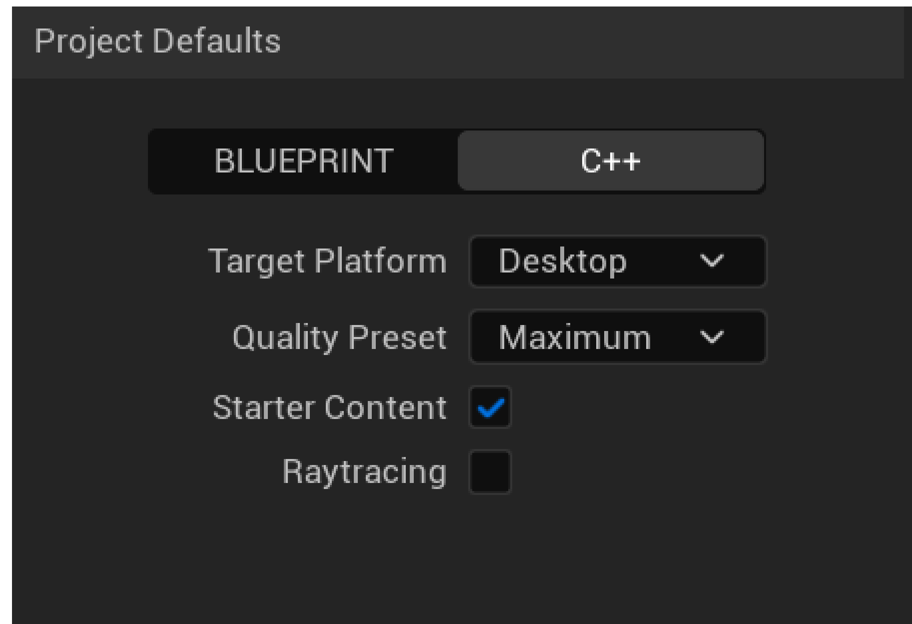

Рисунок 2.1.2

Проект створено.

2.2 Створення механіки нанесення ударів

Unreal Engine може працювати з 3D-об’єктами та анімаціями у форматах .fbx та .obj, а також .uasset, який є унікальним розширенням ассетів, котре Unreal Engine передає файлам у проекті. Великої різниці у виборі формату немає.

До проекту імпортуються анімації рухів персонажа у форматі .fbx.

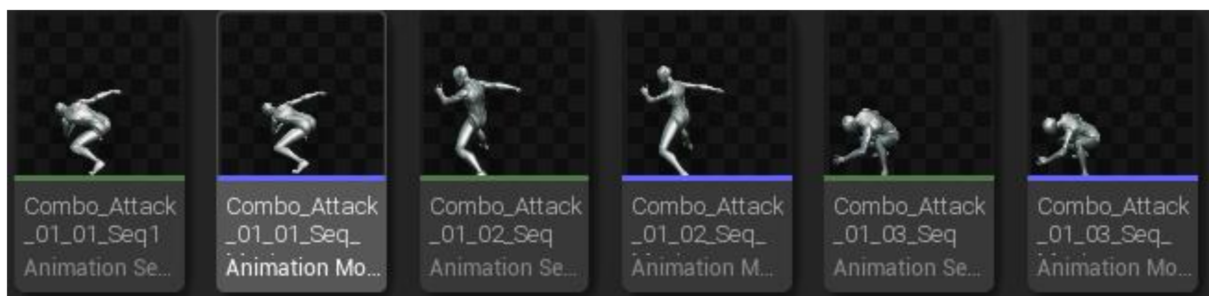

Рисунок 2.2.1

Далі потрібно перетворити ці анімації на компонент типу AnimMontage. Зроблено це задля того, щоб викликати дані анімації у Blueprint або C++, бо інакше доведеться роботи це через налаштування анімаційного Blueprint персонажу, що для даної системи буде складніше[6].

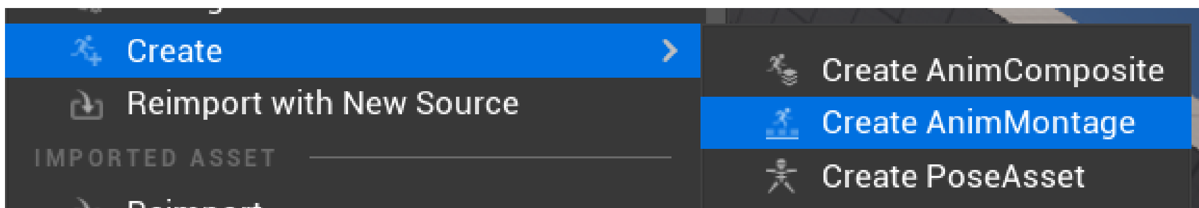

Рисунок 2.2.2

Тепер потрібно створити компонент типу AnimNotifyState

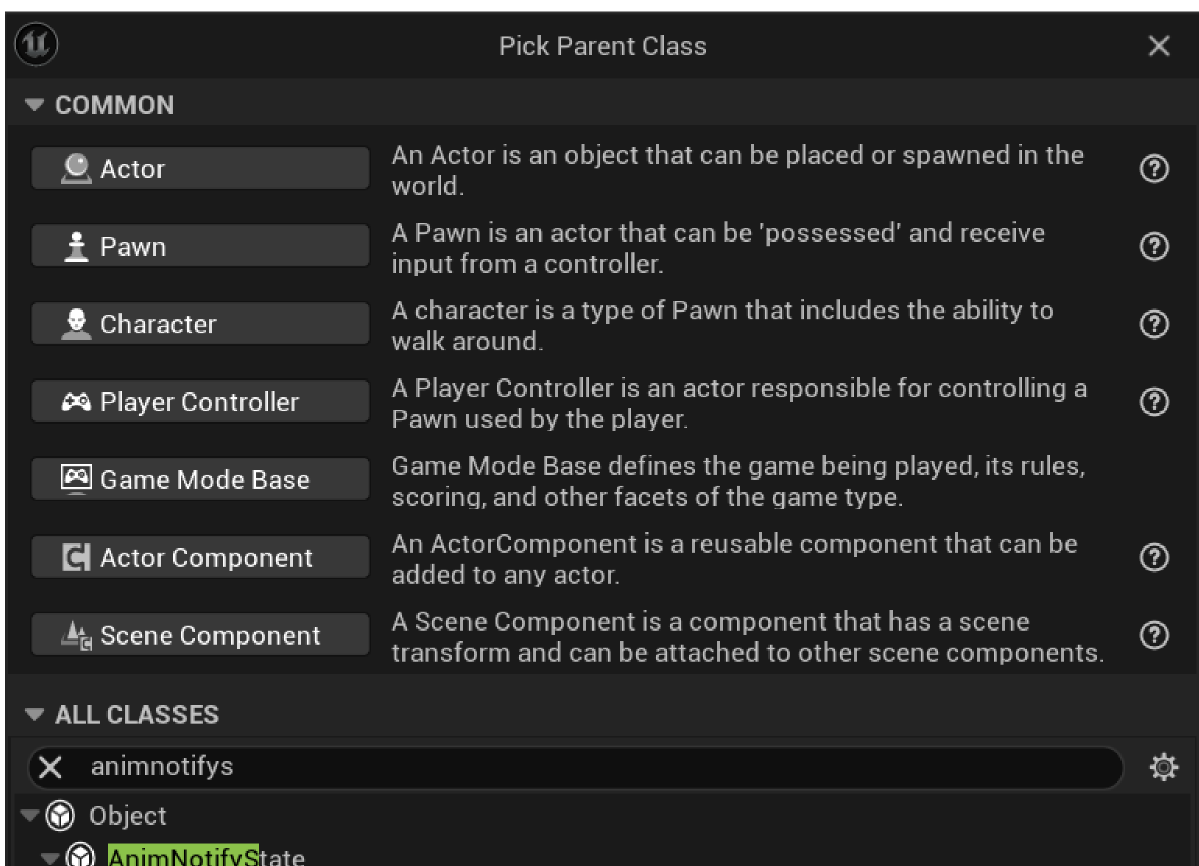

Рисунок 2.2.3

Потрібно перейти до кожного компоненту анімацій персонажа і проставити там Notify CanAttackAgain. Це буде міткою коли можна перервати поточну анімацію і перейти до наступної. Також треба проставити раніше створений AnimNotifyState[7]. Це буде міткою часу, на протязі якої має

працювати функція, що відстежує пересічення руки, що б’є, з комп’ютерними супротивниками.

Рисунок 2.2.4

У блупринті ігрового персонажу створюються наступні змінні.

isAttacking	Boolean
attackCount	Integer
Health	Integer
WasHitted	Boolean

Рисунок 2.2.5

isAttacking - булева змінна, потрібна для того, щоб поки йде одна анімація, інша завчасно не запускалася. attackCount - змінна типу int, слугує лічильником для того, щоб чергувати різні анімації атаки. Health - показник здоров’я персонажу. WasHitted - булева змінна, котра відстежує, чи був нанесений удар по персонажу.

У наступному кроці прописується логіка самого нанесення ударів.

Перед цим йде перехід в анімаційний блупринт персонажу, де прописується логіка для мітки CanAttackAgain, яка ставилася у компонентах анімаційних монтажів[7].

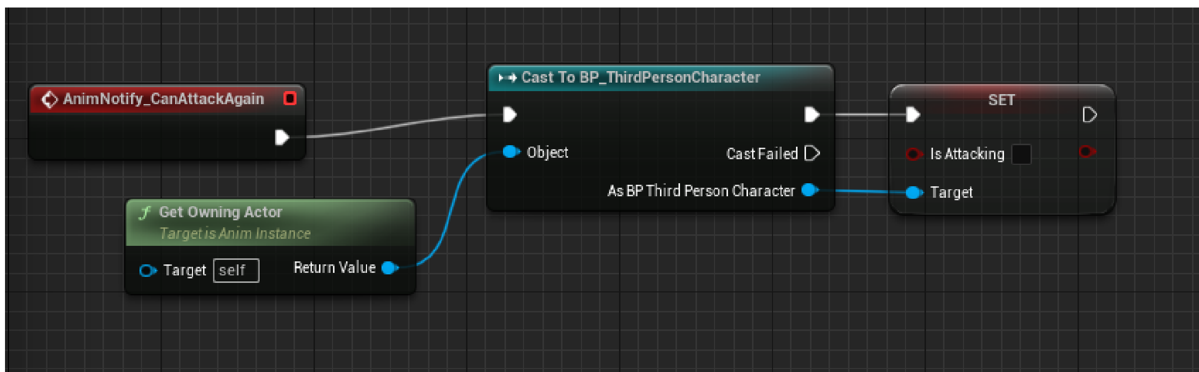

Рисунок 2.2.6

Як тільки монтаж доходить до мітки, змінна в блупринті персонажа приймає значення false. Далі, перехід до блупринту персонажа, де прописується наступне.

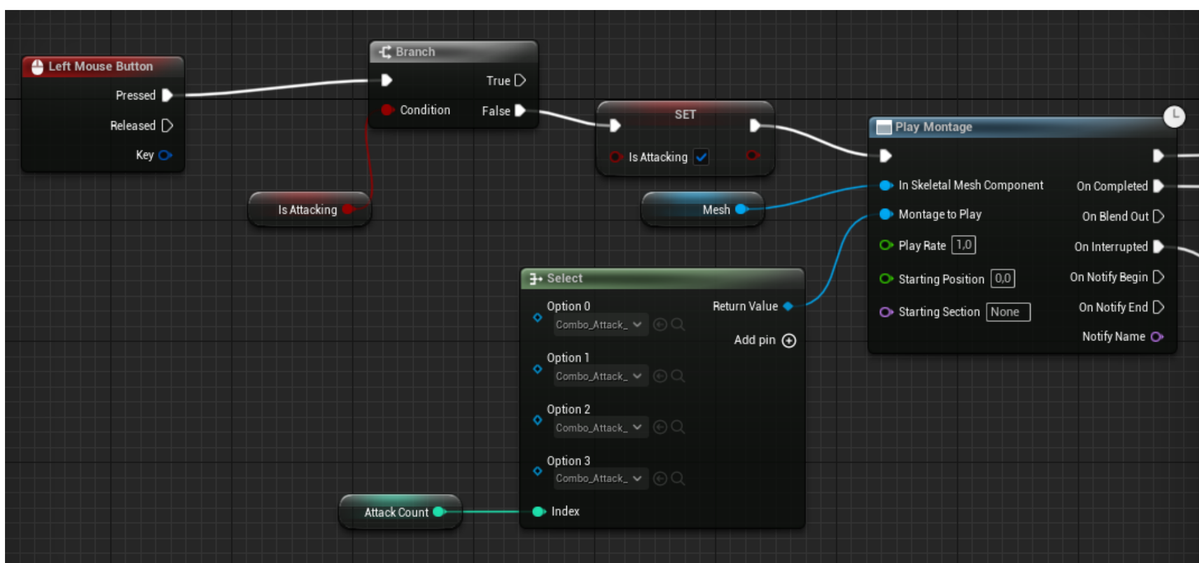

Рисунок 2.2.7

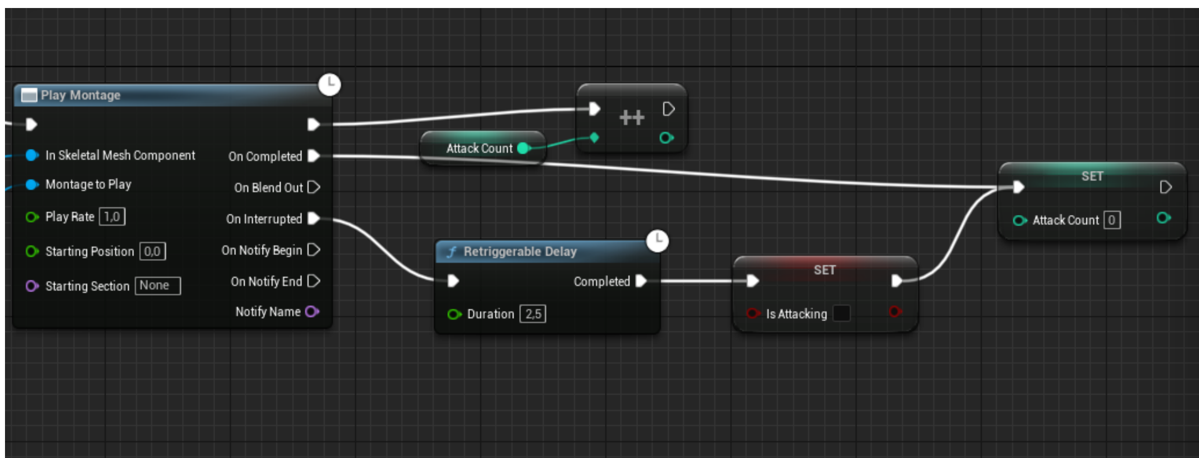

Рисунок 2.2.8

Логіка наступна: при натисканні на ліву кнопку миші йде перевірка змінної isAttacking, тобто чи проводить атаку персонаж[8]. Якщо персонаж вже проводить атаку, то нічого не відбувається. Якщо персонаж не атакує, або ж якщо попередня атака дійшла до фази, в якій можна проводити наступну(тобто дійшла до мітки CanAttackAgain), змінна isAttacking приймає значення true, далі на основі змінної attackCount обирається і відтворюється анімаційний монтаж. Після цього attackCounter збільшується на одиницю, щоб при наступному натисканні на ліву клавішу миші відтворювалася наступна анімація, і паралельно з цим починає дію функція Retriggerable Delay - таймер, що кожного разу, коли раніше ніж за 2,5 секунди відтворюватиметься нова анімація, буде перезапускатися знову з початковим часом 2,5 секунди. Після того, як пройде відведений час, змінна attackCount буде знову прирівняна до 0, щоб ланцюг ударів можна було розпочати з першої анімації. Таким чином персонаж може наносити удари[9].

Потім прописується функція HitDetection. Попередній код лише запускав видимість ударів, самі анімації, ніякої шкоди ботам це не завдасть[10].

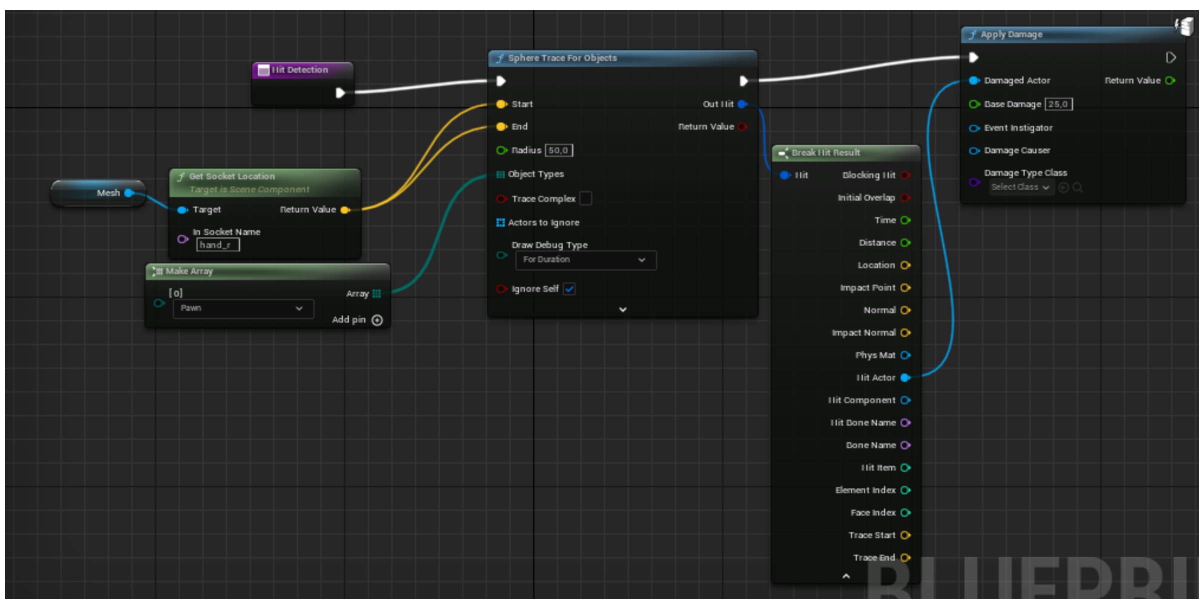

Рисунок 2.2.9

При виклику даної функції, буде створюватися сфера на сокеті під назвою hand_r, тобто навкруги правої руки. У полі Object Types підключаємо массив типу Pawn(тобто об’єкти, котрими може управляти гравець, або комп’ютер), щоб інші об’єкти дана функція ігнорувала. При перетині з об’єктом відповідного типу йому буде завдаватися шкода[11].

Далі повертаємося до об’єкту типу AnimNotifyState, який був створений і проставлений у анімаційних монтажах раніше.

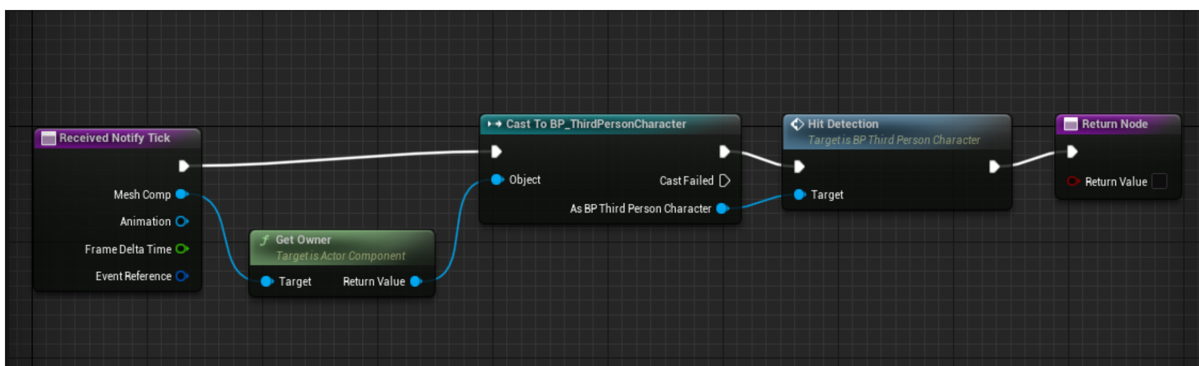

Рисунок 2.2.10

Таким чином при анімації на відрізку часу, поміченим Anim_NS_Hit, буде викликатися функція HitDetection.

Далі прописується логіка того, як персонаж зазнає шкоди.

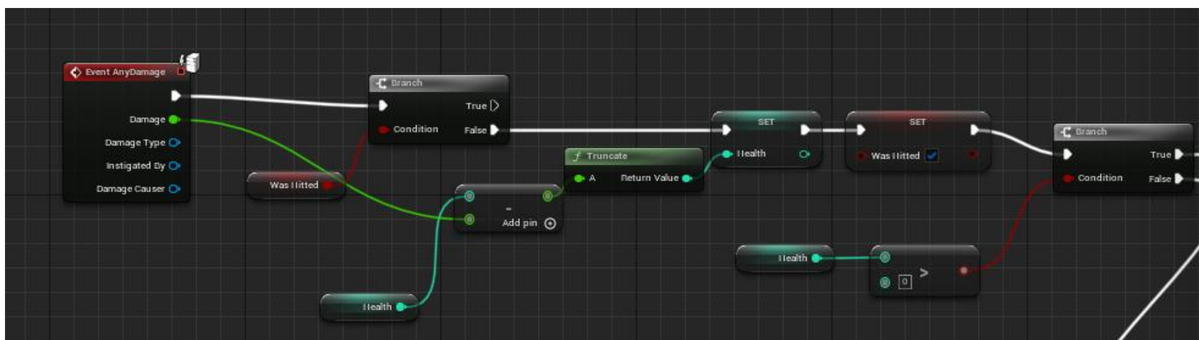

Рисунок 2.2.11

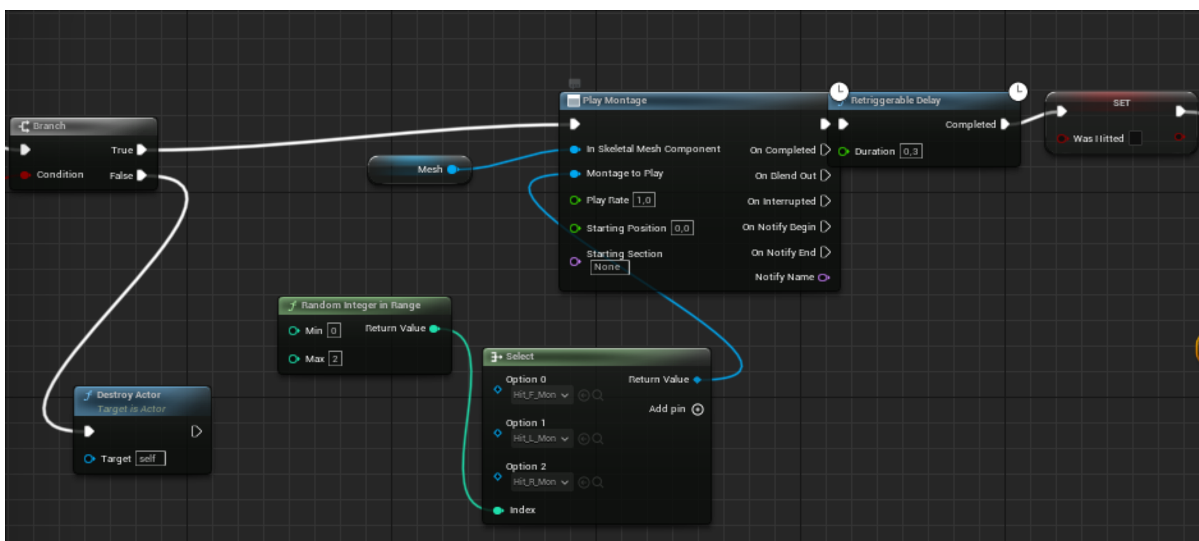

Рисунок 2.2.12

Перш за все, потрібна змінна WasHitted, так як при виконанні функції одного удару функція HitDetected буде спрацьовувати багато разів, так як у кожний момент часу створюється сфера, яка повертає дані про перетин об’єкту. Таких сфер достатньо багато, тому за одну анімацію удару може бути зараховано одразу декілька пересічень одного й того самого об’єкту. Саме щоб цього не допустити потрібна змінна WasHitted. Зробимо так, щоб після попереднього зазнавання шкоди мало пройти хоча б 0,3 секунди[12].

Логіка наступна: персонаж зазнав шкоди за останні 0,3 секунди - нічого не відбувається. Якщо не отримував, то від змінної Health віднімається показник завданої шкоди, змінна WasHitted стає true. Далі йде перевірка, якщо показник здоров’я дорівнює або менше нуля, то актор знищується[13]. Якщо після віднімання показник все ще більше 0, то запускається випадкова анімація

отримання удару і запускається Retriggerable Delay, після якого змінна WasHitted стане false і персонаж знову зможе зазнавати шкоди[14].

2.3 Відтворення дій гравця у інших клієнтів

На усіх створених змінних у полі налаштування реплікації має стояти “Replicated”.

Рисунок 2.3.1

Далі треба замінити подію натискання лівої клавіші миші на іншу.

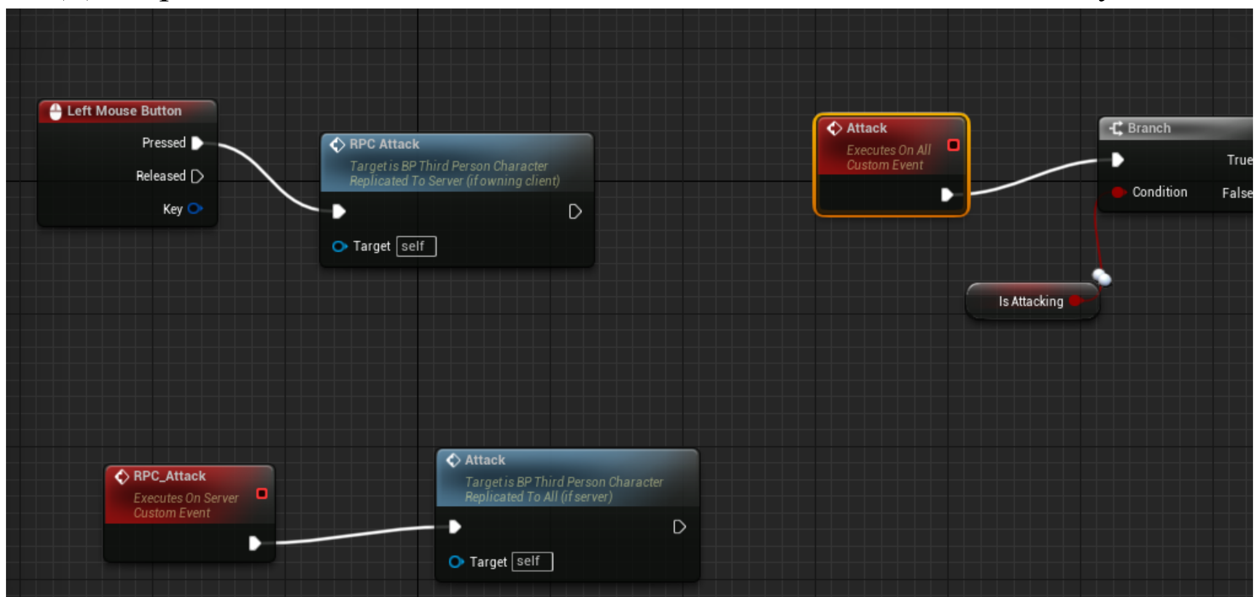

Рисунок 2.3.2

Раніше, атака викликалася одразу після натискання на кнопку. Тепер же при натисканні спрацьовує подія з типом реплікації “Run on Server”, тобто подія буде відбуватися лише на сервері[15].

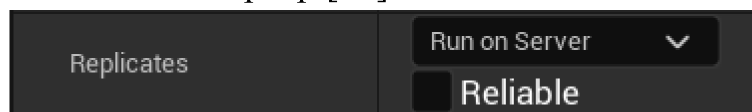

Рисунок 2.3.3

Ця подія у свою чергу викликає подію “Attack” котрій присвоїли минулу логіку удару. Тип реплікації у цієї події “Multicast”, тобто подія буде відбуватися одразу на всіх пристроях, але за умови, що цю подію викликав сервер.

Рисунок 2.3.4

Логіка полягає у наступному: клієнт натискає кнопку, тим самим запускає подію типу “Run on Server”, що виконується на сервері, ця подія у свою чергу викликає подію типу “Multicast”, і так як все це відбувається на сервері, то усі клієнти побачать зміни. Таким чином гравець зможе з клієнту змінювати картинку на усіх пристроях[16].

2.4 Створення платформи, що рухається

Для наповнення ігрового рівня було створено платформи, що рухаються в один кінець і назад. Логіка руху платформи має виконуватися лише на серверній стороні. Тому до коду було додано рядок if(HasAuthority()) [17], що буде виконувати логіку у фігурних дужках лише за умови, що код виконується на стороні серверу.

```
if(HasAuthority())
{
	FVector Location = GetActorLocation()
	float JourneyLength = (GlobalTargetLocation - GlobalStartLocation).Size();
	float JourneyTravelled = (Location - GlobalStartLocation).Size();

	if(JourneyTravelled >= JourneyLength)
	{
		FVector temp = GlobalStartLocation;
		GlobalStartLocation = GlobalTargetLocation;
		GlobalTargetLocation = temp;
	}

	FVector Direction = (GlobalTargetLocation - GlobalStartLocation).GetSafeNormal();
	Location += (Speed * DeltaTime * Direction);
	SetActorLocation(Location);
}
```

У цьому коді платформа кожного кадру змінює свої координати від точки початку до кінцевої точки на 20 одиниць. Після її досягнення відбувається повернення в початкову точку.

Слід звернути увагу на передостанній рядок “Location += (Speed * DeltaTime * Direction);”. Змінна DeltaTime - це кількість кадрів у секунду на конкретному пристрої. Додана вона через те, що зазвичай на серверах кількість кадрів у секунду менша, ніж на клієнтах[18].

2.5 Створення та приєднання до онлайн-сесій

Сесії - це спосіб створення, приєднання та управління грою через Інтернет або локальну мережу[19]. Сесії з’єднують гравців з сервером, містять у собі інформацію про максимальну та поточну кількість гравців, надають можливість налаштовувати параметри гри тощо[20].

На рисунку показано код кнопки з головного меню, що створює онлайн-сесію та відкриває ігровий рівень.

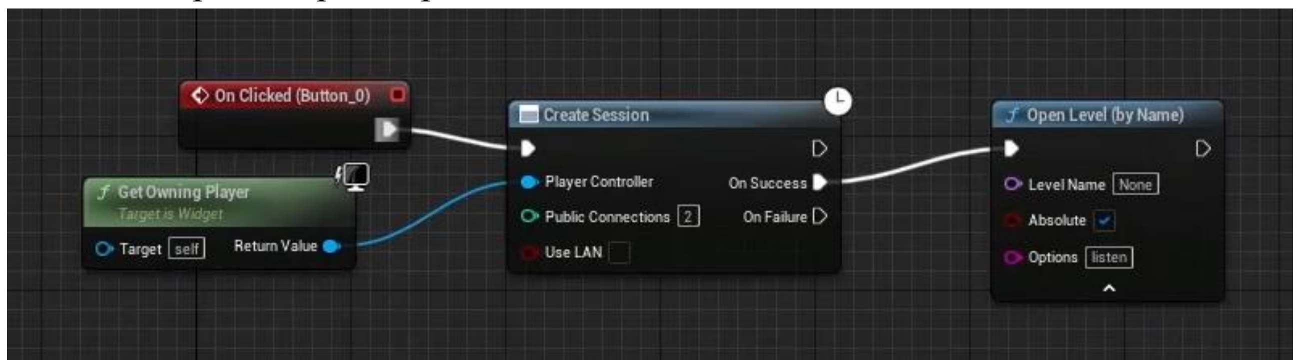

Рисунок 2.5.1

На наступному рисунку показано код для кнопки пошуку існуючих сесій. Даний код повертає масив усій знайдених сесій, але не більше числа зазначеного у полі “Max Results”.

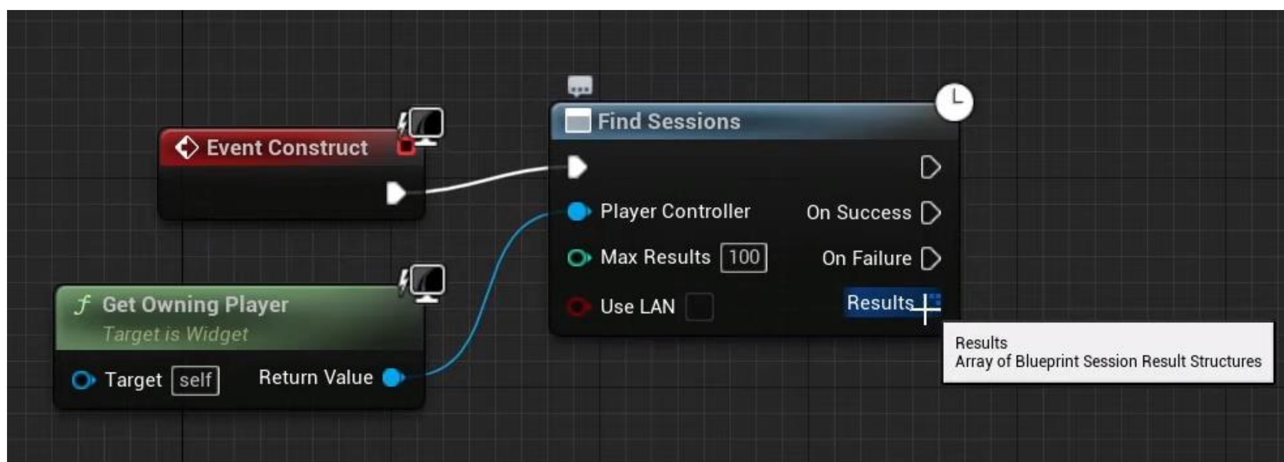

Рисунок 2.5.2

Далі отримавши массив з попередньої функції, гравець може приєднатися до сесії.

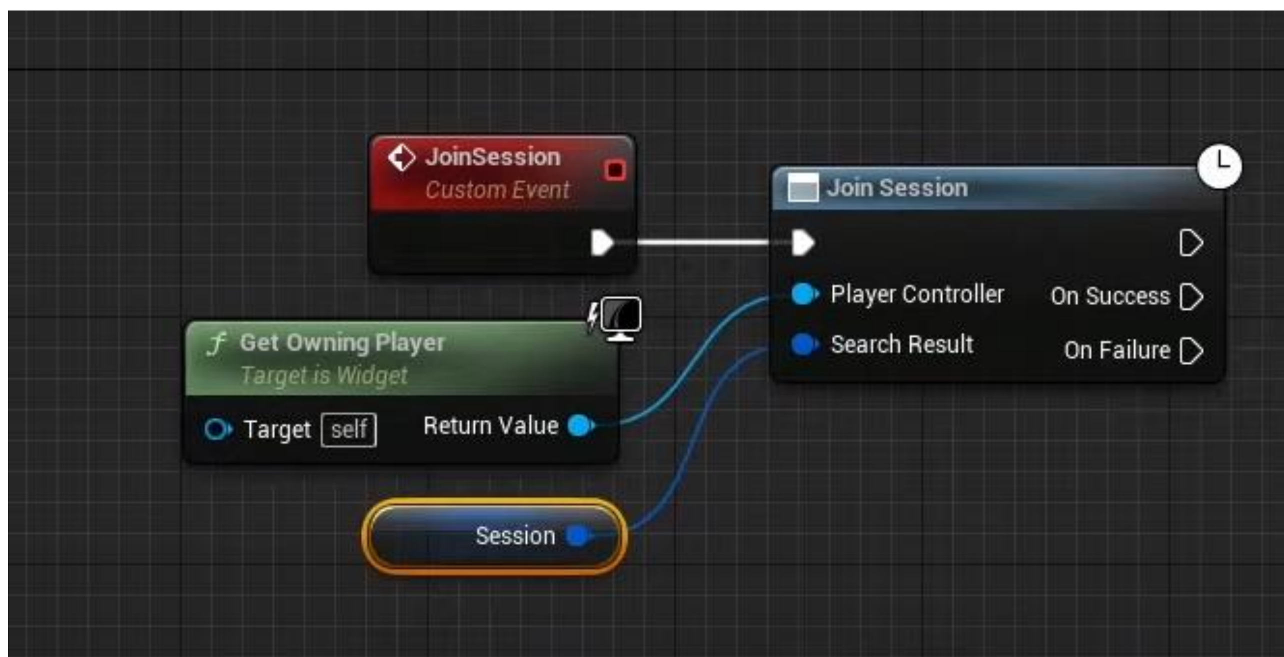

Рисунок 2.5.3

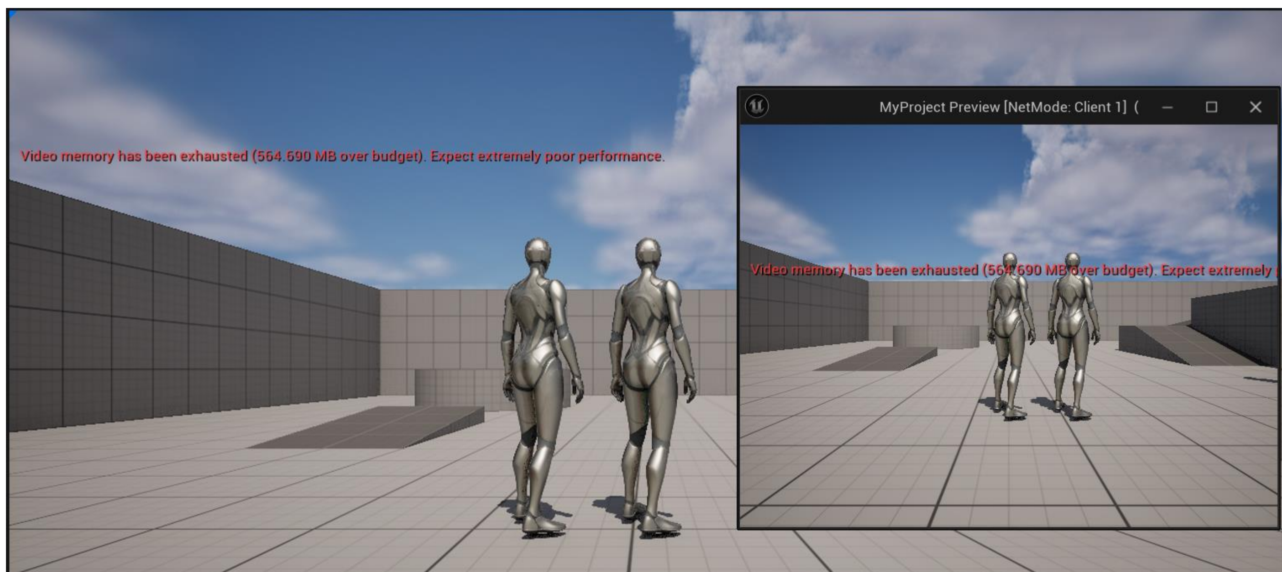

Рисунок 2.5.4

3 АНАЛІЗ РЕЗУЛЬТАТІВ

На початку роботи з інформаційною технологією користувач має можливість створити ігрову сесію, приєднатися до вже існуючої як через список серверів, так і напряму через ір-адресу, запустити гру у однокористувацькому режимі.

Рисунок 3.1

У меню “Create Server” можна обрати назву серверу, ігровий рівень, кількість гравців та тип підключення.

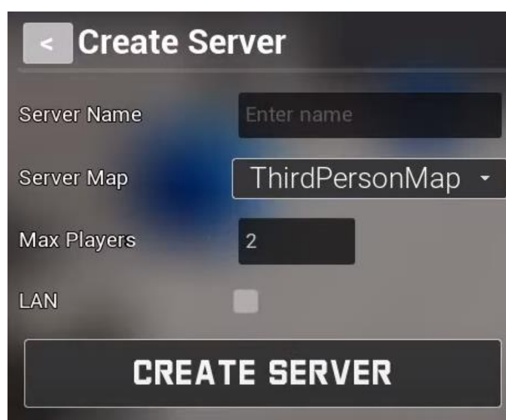

Рисунок 3.2

У меню “Server List” користувач бачить список серверів з їх назвами, кількістю поточних гравців, максимальну кількість гравців, а також затримку підключення.

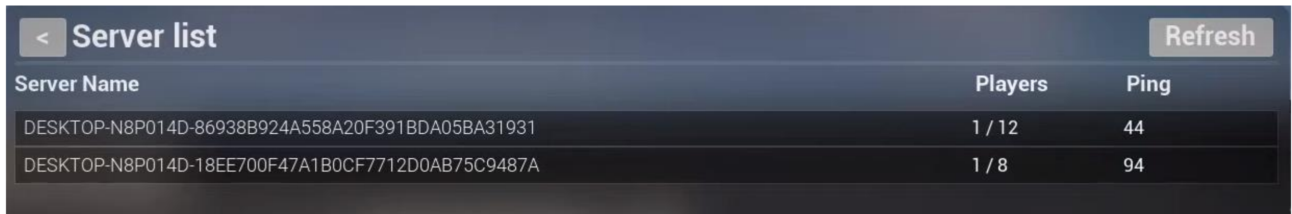

Рисунок 3.3

При успішному підключенні на ігровому рівні з’являються два чи більше гравців, які можуть контактувати з іншими гравцями або ігровими об’єктами.

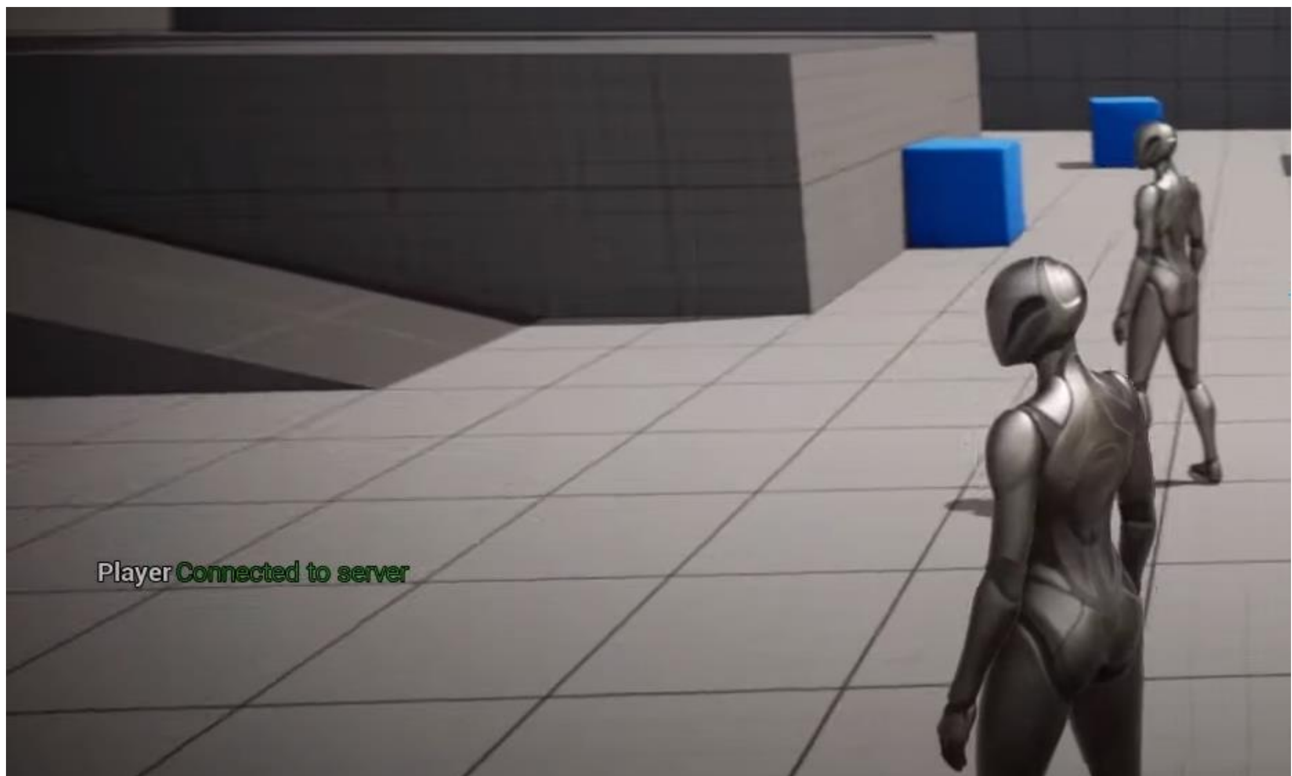

Рисунок 3.4

ВИСНОВКИ

В результаті виконання роботи була створена інформаційна технологія проектування ігрової системи з можливістю гри в мережі на основі Unreal Engine 5. У ході роботи було виконано наступні завдання:

1) розглянуто принципи роботи багатокористувацького режиму та видів підключення користувачів
2) проведено аналіз інструментів розробки
3) проведено дослідження по роботі з мережевим підключенням у Unreal Engine 5
4) створено мережеву систему, у якій дії одного гравця відтворюються у всіх інших
5) відтворено механізм створення, знаходження та підключення до онлайн-сесій

У подальшому результати дипломної роботи можуть бути розвинені у повноцінний додаток. Подальші плани по розвитку додатку:

1) Покращення графічного інтерфейсу
2) Додавання до проекту Online Subsystem
3) Додавання нових об’єктів для покращення ігрового процесу
4) Створення нових ігрових режимів

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unreal Engine 5 Мультиплеєр [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/playlist?list=PL2suyruNHd0hxXUHQHWpWeiKY5bRzDKF5
2. Unreal Engine 5.3 Documentation [Електронний ресурс] // https://docs.unrealengine.com/ – Режим доступу до ресурсу: https://docs.unrealengine.com/5.3/en-US/
3. Replication In Unreal Engine 4 [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/playlist?list=PLQN3U_-lMANOaPmzSGLDhyWzKzwgKDz9V
4. C++ Network Multiplayer - Unreal Engine [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/playlist?list=PLRrOfLSL-UhQgXSHmY2eH6DgvuaJKBvmo
5. C++ Programming for Unreal Game Development [Електронний ресурс] // https://www.coursera.org/ – Режим доступу до ресурсу: https://www.coursera.org/specializations/cplusplusunrealgamedevelopment?utm_medium=sem&utm_source=gg&utm_campaign=B2C_EMEA__coursera_FTCOF_courseraplus&campaignid=20858197888&adgroupid=156245795749&device=c&keyword=coursera%20cost&matchtype=b&network=g&devicemodel=&adposition=&creativeid=684297719990&hide_mobile_promo=&term=%7Bterm%7D&gclid=EAIaIQobChMI_tiTr8btgwMVYpODBx0c3wYREAAYASAAEgJC6_D_BwE
6. Blueprint Multiplayer | v4.11 | Unreal Engine [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/playlist?list=PLZlv_N0_O1gYqSlbGQVKsRg6fpxWndZqZ
7. Unreal Engine 5 | Basic Melee Combat Tutorial Series [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/playlist?list=PLlswSOADCx3fG8ATHm3GDmmdGGCfPhl0y

8. Intermediate Object-Oriented Programming for Unreal Games [Електронний ресурс] // https://www.coursera.org/ – Режим доступу до ресурсу: https://www.coursera.org/learn/intermediate-object-oriented-programming--unreal-games
9. C++ Class Development [Електронний ресурс] // https://www.coursera.org/ – Режим доступу до ресурсу: https://www.coursera.org/learn/cpp-class-development
10. More C++ Programming and Unreal [Електронний ресурс] // https://www.coursera.org/ – Режим доступу до ресурсу: https://www.coursera.org/learn/more-programming-unreal?
11. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14 / Scott Meyers
12. Комп'ютерні мережі. 5-й лід. / Таренбаум Е. С., Везерол Д.

13. Unreal Engine 5 Beginner To Advanced | UE5 Full Length Course [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/watch?v=Yv2y7pnIqws
14. Blueprints VS C++ [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/playlist?list=PLF2xQSe67MU1jnCjlzOAKpNQsaZ5R0fdP
15. Gamium Dev UE4 FPS Course [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/playlist?list=PLF2xQSe67MU0asA_Ar6NtUHs6EL3AhALD
16. Unreal Engine 5 Multiplayer Course [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/playlist?list=PLF2xQSe67MU1T2cQJK3mgUmIL_ckS_Lug
17. C / C++ [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/playlist?list=PLWKjhJtqVAbmUE5IqyfGYEYjrZBYzaT4m

18. Multiplayer in Unreal Engine: How to Understand Network Replication [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/watch?v=JOJP0CvpB8w
19. Blueprints vs. C++: How They Fit Together and Why You Should Use Both [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/watch?v=VMZftEVDuCE
20. Why Solo Developers Should Use Unreal [Електронний ресурс] // https://www.youtube.com/ – Режим доступу до ресурсу: https://www.youtube.com/watch?v=l9y5B0cgUHY

ДОДАТКИ

Файл MyCharacter.h

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "Camera/CameraComponent.h"
#include "GameFramework/SpringArmComponent.h"
#include "MyCharacter.generated.h"

UCLASS()
class Diplom AMyCharacter : public ACharacter
{
   GENERATED_BODY()

public:
   // Sets default values for this character's properties
   AMyCharacter();

protected:
   // Called when the game starts or when spawned
   virtual void BeginPlay() override;

public:
   // Called every frame
   virtual void Tick(float DeltaTime) override;

   // Called to bind functionality to input
```

```
    virtual void SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent) override;

private:
    // Camera boom positioning the camera behind the character
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
    class USpringArmComponent* CameraBoom;

    // Follow camera
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
    class UCameraComponent* FollowCamera;

    // Variables for movement
    FVector MovementInput;
    bool bJumpPressed;

    // Functions for input
    void MoveForward(float Value);
    void MoveRight(float Value);
    void StartJump();
    void StopJump();
};
```

Файл MyCharacter.cpp

```
#include "MyCharacter.h"

// Sets default values
AMyCharacter::AMyCharacter()
{
   // Set this character to call Tick() every frame
   PrimaryActorTick.bCanEverTick = true;

   // Create CameraBoom (spring arm component)
   CameraBoom =
CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoo
m"));
   CameraBoom->SetupAttachment(RootComponent);
   CameraBoom->TargetArmLength = 300.0f; // How far the camera
should be from the player
   CameraBoom->bUsePawnControlRotation = true;

   // Create FollowCamera
   FollowCamera =
CreateDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"
));
   FollowCamera->SetupAttachment(CameraBoom);
}
```

```
// Called when the game starts or when spawned
void AMyCharacter::BeginPlay()
{
   Super::BeginPlay();
}

// Called every frame
void AMyCharacter::Tick(float DeltaTime)
{
   Super::Tick(DeltaTime);
}

// Called to bind functionality to input
void AMyCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
   Super::SetupPlayerInputComponent(PlayerInputComponent);

   // Bind functions to input
   PlayerInputComponent->BindAxis("MoveForward", this, &AMyCharacter::MoveForward);
   PlayerInputComponent->BindAxis("MoveRight", this, &AMyCharacter::MoveRight);
   PlayerInputComponent->BindAction("Jump", IE_Pressed, this, &AMyCharacter::StartJump);
   PlayerInputComponent->BindAction("Jump", IE_Released, this, &AMyCharacter::StopJump);
}
```

```
// Handle movement input
void AMyCharacter::MoveForward(float Value)
{
   // Find out which way is forward
   const FRotator Rotation = Controller->GetControlRotation();
   const FRotator YawRotation(0, Rotation.Yaw, 0);

   // Get forward vector
   const FVector Direction =
FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);

   // Add movement in that direction
   AddMovementInput(Direction, Value);
}

// Handle strafing input
void AMyCharacter::MoveRight(float Value)
{
   // Find out which way is right
   const FRotator Rotation = Controller->GetControlRotation();
   const FRotator YawRotation(0, Rotation.Yaw, 0);

   // Get right vector
   const FVector Direction =
FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);

   // Add movement in that direction
   AddMovementInput(Direction, Value);
}
```

```
// Handle jump input
void AMyCharacter::StartJump()
{
   bJumpPressed = true;
   Jump();
}

void AMyCharacter::StopJump()
{
   bJumpPressed = false;
   StopJumping();
}
```

Файл MyPlatform.h

```
#pragma once

#include "CoreMinimal.h"
#include "Engine/StaticMeshActor.h"
#include "MyPlatform.generated.h"

UCLASS()
class Diplom_API AMyPlatform:public AStaticMeshComponent
{
GENERATED_BODY()

public:
```

```
        AMyPlatform();

        virtual void BeginPlay() override;

        virtual void Tick(float DeltaTime) override;

        UPROPERTY(EditAnywhere, BlueprintReadWrite)
          float Speed = 20;

        UPROPERTY(EditAnywhere, BlueprintReadWrite, Meta = (MakeWidget =
true))
          FVector Target location;

private:

Fvector GlobalStartLocation;

Fvector GlobalTargetLocation;

}
```

Файл MyPlatform.cpp

```
#include "MyPlatform.h"

AMyPlatform::AMyPlatform()
{
        PrimaryActorTick.bCanEverTick = true;
```

```
        SetMobility(EComponentMobility::Movable);
}

void AMyPlatform::BeginPlay()
{
        Super::BeginPlay();
        if(HasAuthority())
        {
           SetReplicates(true);
           SetReplicateMovement(true);
        }

        GlobalStartLocation = GetActorLocation();
        GlobalTargetLocation =
GetTransform().TransformPosition(TargetLocation);

}

void AMyPlatform::Tick(float DeltaTime)
{
        Super::Tick(DeltaTime);

        if(HasAuthority())
        {
           FVector Location = GetActorLocation()
           float JourneyLength = (GlobalTargetLocation -
GlobalStartLocation).Size();
           float JourneyTravelled = (Location - GlobalStartLocation).Size();
```

```
            if(JourneyTravelled >= JourneyLength)
            {
                FVector temp = GlobalStartLocation;
                GlobalStartLocation = GlobalTargetLocation;
                GlobalTargetLocation = temp;
            }

            FVector Direction = (GlobalTargetLocation - 
GlobalStartLocation).GetSafeNormal();
            Location += (Speed * DeltaTime * Direction);
            SetActorLocation(Location);
        }


}
```