МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Центр заочної, дистанційної та вечірньої форм навчання
Кафедра комп’ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ РОЗРОБКИ ІГРОВОЇ СИСТЕМИ НА БАЗІ UNREAL ENGINE

Виконала здобувачка гр. ІН.мз-21с _____ Світлана Гайворонська
(підпис)

Керівник
доцентка кафедри комп’ютерних
наук, к.т.н., доцент _____ Наталія БАРЧЕНКО
(підпис)

Суми – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Центр заочної, дистанційної та вечірньої форм навчання

Кафедра комп'ютерних наук

«До захисту допущено»

В.о. завідувача кафедри

_____ Ігор ШЕЛЕХОВ

(підпис)

11 грудня 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

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

на тему: «Інформаційна технологія розробки ігрової системи на базі Unreal Engine» здобувачки групи Ін.мз-21с Гайворонської Світлани Андріївни

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

 Світлана Гайворонська

(підпис)

Керівник

доцентка кафедри комп'ютерних наук, к.т.н., доцент

Наталія БАРЧЕНКО 

(підпис)

Суми – 2023

Сумський державний університет

Центр заочної, дистанційної та вечірньої форм навчання

Кафедра комп'ютерних наук

«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»
здобувачки групи ІН.мз-21с

1. Тема роботи: «Інформаційна технологія розробки ігрової системи на базі Unreal Engine» затверджую наказом по СумДУ від *«20» листопада 2023 р. № 1308-VI*

2. Термін здачі здобувачем кваліфікаційної роботи *до 13 грудня 2023 року*

3. Вхідні дані до кваліфікаційної роботи _____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)

1) Опис предметної області, обґрунтування оптимального варіанту реалізації мети цієї роботи. 2) Вибір програмних засобів реалізації. 3) Програмна реалізація ігрового застосунку "Arden Dungeon" у жанрі RPG.

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

6. Консультанти до проєкту (роботи), із значенням розділів проєкту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «__1__» __жовтня___ 20_23_ р.

Завдання прийняв до виконання _____ (підпис)

Керівник _____ (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1.	Розробка та затвердження завдання на виконання	15.10.2023	Виконано
2.	Пошук літератури за темою роботи	06.10.2023	Виконано
3.	Складання календарного плану КРБ	17.10.2023	Виконано
4.	Аналіз предметної області	18.10.2023	Виконано
5.	Розробка проектної документації	22.10.2023	Виконано
6.	Моделювання та конструювання Акту №1 ігрового застосунку	04.11.2023	Виконано
7.	Моделювання та конструювання Акту №2 ігрового застосунку	11.11.2023	Виконано

8.	Моделювання та конструювання Акту №3 ігрового застосунку	18.11.2023	Виконано
9.	Кодування та Bluprinting	27.11.2023	Виконано
10.	Тестування та аналіз результатів тестування	28.11.2023	Виконано
11.	Оформлення пояснювальної записки до магістерської роботи	30.11.2023	Виконано

Здобувач вищої освіти _____ Керівник _____
(підпис) (підпис)

АНОТАЦІЯ

до кваліфікаційної роботи магістра

«Інформаційна технологія розробки ігрової

системи на базі Unreal Engine »

Студенка групи ІН.мз-21с.:

Гайворонська Світлана Андріївна

Керівник: доцентка кафедри комп’ютерних наук, к.т.н., доцент

Барченко Н.Л.

Кваліфікаційна робота мігастра присвячена дослідженню аналогів та розробці ігрового застосунку в жанрі RPG.

Об’єктом роботи є процес розробки ігрового застосунку в жанрі RPG.

Предметом роботи є підходи та інформаційні технології розробки ігрових застосунків.

Метою роботи є активізація пізнавальної діяльності та створення унікального світу. Мета може бути в створенні власного унікального світу, який би відзначався особливим стилем, історією та атмосферою за рахунок розробки та впровадження комп’ютерних ігрових технологій в жанрі RPG.

Практичне значення програмного застосунку полягає у можливості його використання в часи війни в Україні як варіант проводження вільного часу.

Кваліфікаційна робота магістра викладена на 94 сторінки, вона містить 4 розділи, 84 ілюстрацій, 21 джерел в переліку посилань.

Ключові слова: розробка ігор, RPG, C++, Bluprint .

ЗМІСТ

ВСТУП

У сучасному світі програмісти мають важливу роль у розвитку цивілізації через створення програмного забезпечення та веб-сайтів. Ріст комп'ютерної

індустрії та штучного інтелекту підсилює попит на їхні навички. Інформаційні технології проникають у всі сфери життя, включаючи охорону здоров'я, освіту, роботу, спілкування та розваги.

Розваги в сьогоденні не можуть не включати в себе ігри на ПК. Ігри дають додаткову можливість знайти собі хобі і в дитинстві, і в зрілому віці. Ігри до вподоби людям також через те, що вони дають шанси пізнати більше, розширити кругозір, думати нетипово й індивідуально, знаходити рішення на складні питання й вирішувати непрості довгі квести. Вони не втратять популярності ні серед дітей, ні серед дорослої молоді ще багато часу, адже розвиваються пліч о пліч з усіма технологіями. Ігри - це добрий спосіб розвити логіку, пам'ять та реакцію.

Якщо розглянути ігри з іншого погляду, то для їх розробки потрібно мати значні знання та навички. Комп'ютерні ігри поділяються на різні жанри: екшени, шутери, аркади, спортивні, квести, рольові (RPG/MMORPG), гонки, симулятори, головоломки, пригоди, стратегії та інші. Кожен жанр вимагає свого унікального набору знань для створення.

Наприклад, у випадку стратегій особливу увагу слід звертати на штучний інтелект, оскільки дії комп'ютерних опонентів сильно впливають на геймплей.

Для шутерів, крім штучного інтелекту, важливу роль відіграють анімації та рендеринг, оскільки в цьому жанрі необхідно відповідати високим технічним стандартам, які встановили провідні компанії.

Майже кожен жанр може включати мультиплеєрну модель, що означає наявність відділу, який відповідає за мережеву взаємодію, при розробці гри.

Усе це підкреслює складність та високий рівень спеціалізованих знань, необхідних для розробки ігор. В цьому процесі беруть участь десятки, іноді сотні фахівців з програмування, дизайну та анімації.

Тема роботи є актуальною, оскільки в наш час важливе постійне покращення логічного та критичного типів мислення. Метою кваліфікаційної роботи є активізація пізнавальної діяльності та удосконалення рівня розумових та

логічних (аналіз, синтез) операцій за рахунок розробки та впровадження комп’ютерних ігрових технологій в жанрі RPG.

Для досягнення поставленої мети необхідно розв’язати наступні завдання:

аналіз предметної області та існуючих аналогів;

розробка вимог до ігрового застосунку;

моделювання та проектування застосунку;

розробка дизайну ігрових сцен;

програмна реалізація, тестування та відлагодження прототипів гри.

Об’єктом роботи є процес розробки ігрового застосунку в жанрі RPG,

а предметом - підходи та інформаційні технології розробки ігрових застосунків.

1 АНАЛІЗ ПРЕДМЕТНОЇ СФЕРИ РОЗРОБКИ ІГРОВИХ ЗАСТОСУНКІВ

1.1 Опис предметної сфери розробки комп’ютерних ігор

Предметна сфера розробки комп'ютерних ігор - це область, яка охоплює всі аспекти створення та вдосконалення відеоігор на комп'ютерах і інших ігрових платформах. Ця сфера включає в себе різні етапи і аспекти розробки від ідеї до

впровадження гри на ринок. Нижче наведено опис основних аспектів предметної сфери розробки комп'ютерних ігор:

Дизайн гри: Дизайн гри включає в себе створення концепції гри, геймплейних механік, сюжету, персонажів та всіх ігрових елементів. Дизайнери визначають, як гра буде виглядати та як гравці будуть взаємодіяти з нею.

Програмування: Розробники програмного забезпечення відповідають за написання коду гри, розробку ігрового движка і реалізацію ігрових механік. Це включає в себе створення штучного інтелекту, фізичної моделі гри та інших технічних аспектів.

Мистецтво та дизайн: Ця галузь охоплює створення графіки, звуку, анімації, дизайну інтерфейсу та ігрових об'єктів. Мистецтво та дизайн роблять гру візуально та аудіально привабливою для гравців.

Сценарій та сюжет: Письменники та сценаристи розробляють сюжет гри, створюючи діалоги, квести та інші ігрові події, які розповідають історію та взаємодіють з персонажами.

Тестування та якість: Тестери гри відповідають за перевірку наявних помилок, багів та вдосконалення гри відповідно до потреб гравців. Також забезпечують відповідність гри встановленим стандартам якості.

Маркетинг та реклама: Команди маркетингу відповідають за рекламу гри, побудову аудиторії та виведення гри на ринок. Це включає в себе створення рекламних матеріалів, роботу з соціальними медіа та інші методи просування гри.

Дистрибуція і продаж: Для того, щоб гра потрапила до рук гравців, необхідно вирішити питання дистрибуції. Це може включати в себе продаж гри через онлайн магазини, роздавання її на фізичних носіях або навіть моделі підписки.

Підтримка та оновлення: Після випуску гри, розробники зазвичай продовжують підтримувати гру, видаючи патчі, оновлення та додатковий контент.

Спільнота гравців: Важливо створити і підтримувати активну спільноту гравців, яка буде обговорювати гру, надавати зворотний зв'язок та сприяти розвитку громадськості навколо гри.

Захист інтелектуальної власності: Розробники повинні також дбати про захист своєї інтелектуальної власності, включаючи авторські права, патенти і торгові марки.

Предметна сфера розробки комп'ютерних ігор дуже різноманітна і вимагає співпраці різних фахівців для створення успішних ігор.

1.2 Жанри комп’ютерних ігор

Комп'ютерні ігри існують у багатьох різних жанрах, кожен із яких має свої унікальні особливості та характерні риси. Нижче наведено деякі з найпопулярніших жанрів комп'ютерних ігор:

Екшен (Action): Цей жанр акцентується на швидкому та динамічному геймплеї, де гравцю доводиться керувати персонажем і взаємодіяти з навколишнім середовищем для досягнення мети, яка може включати в себе бій, стрілянину та інші види фізичної активності.

Шутер (Shooter): В цьому жанрі основним елементом є стріляння. Шутери поділяються на від першої особи (FPS - First-Person Shooter) та від третьої особи (TPS - Third-Person Shooter).

Рольові ігри (RPG - Role-Playing Game): RPG надають гравцям можливість керувати персонажем, розвивати його, приймати рішення, впливати на сюжет і взаємодіяти з іншими персонажами. Цей жанр може бути як фентезі, так і науково-фантастичним.

Стратегія (Strategy): Гравці в стратегічних іграх приймають рішення, які впливають на розвиток гри та військові аспекти, такі як розташування військ, економіка і дипломатія.

Пригодницький (Adventure): Пригодницькі ігри фокусуються на розв'язанні головоломок, розслідуванні та історії. Вони можуть бути реалістичними або фантастичними.

Жахи (Horror): Цей жанр призначений для того, щоб залякувати гравця. Він включає в себе елементи страху, напруги та моторошних сюжетів.

Гонки (Racing): Гравці змагаються в гонках на автомобілях, мотоциклах, велосипедах, тощо. Гонки можуть бути реалістичними або аркадними.

Симулятор (Simulation): Симулятори моделюють реальні або уявні ситуації. Це може включати симулятори авіаспорту, сім'ї, життя фермера та інші.

Головоломка (Puzzle): Головоломкові ігри акцентуються на розв'язанні складних завдань та головоломок. Головоломки можуть бути логічними, фізичними або механічними.

Спорт (Sports): Цей жанр імітує різні види спорту, такі як футбол, баскетбол, гольф та інші, дозволяючи гравцям змагатися у спортивних подіях.

Музикальні ігри (Music): Гравці в цих іграх виконують музичні завдання та ритмічні ігри, часто використовуючи спеціальні контролери.

Мультиплеєрні ігри (Multiplayer): Це не жанр, а більше спосіб гри, де гравці можуть взаємодіяти один з одним у режимі реального часу через Інтернет або на одному екрані.

Це лише кілька прикладів жанрів комп'ютерних ігор, і існують численні варіації та поєднання цих жанрів, що робить ігрову індустрію різноманітною та захоплюючою для гравців.

Популярні комп'ютерні ігри можуть змінюватися з часом, але на даний момент можеми відокремити такий список популярних ігор:

The Witcher 3: Wild Hunt: Ця рольова гра розроблена CD Projekt Red є однією з найуспішніших ігор останнього десятиліття. Ви керуєте героєм Геральтом з Рівії, який вирушає в подорож через великий фентезійний світ,

розв'язуючи головоломки, борючись з чудовиськами і вирішуючи важливі сюжетні завдання.

Cyberpunk 2077: Інша гра від CD Projekt Red, Cyberpunk 2077, стала великим сенсацією завдяки світу наукової фантастики, який відобразив мегаполіс майбутнього. Гравці грають за головного героя Ві, який відправляється на ризиковану місію в місті Найт-Сіті.

Minecraft: Minecraft від Mojang є однією з найпопулярніших ігор у світі. Гравці мають можливість будувати світи з кубів і досліджувати їх, виживаючи у небезпечних умовах та взаємодіючи з іншими гравцями.

Fortnite: Ця битва королів від Epic Games стала сенсацією завдяки безкоштовному режиму гри "Битва королів", де 100 гравців борються на великій карті. Гра має яскравий арт-стиль та надзвичайну популярність серед молодого гравців.

Among Us: Ця гра від InnerSloth стала популярною завдяки своєму мультиплеєрному режиму, де гравці спільно вирішують завдання на космічному кораблі, водночас виявляючи імпостера серед команди.

Counter-Strike: Global Offensive: CS: GO - це популярний онлайн шутер від Valve. Гравці вступають в команди терористів та контр-терористів і змагаються в різних режимах бою.

League of Legends: Ця гра в жанрі MOBA (Мультиплеєрна онлайн битва арени) від Riot Games - одна з найпопулярніших гри в світі. Гравці утворюють команди та змагаються один проти одного на арені.

Grand Theft Auto V: Ця гра від Rockstar Games пропонує відкритий світ, в якому гравці можуть вчиняти злочини, вести справи та вивчати велике місто Лос-Сантос.

Red Dead Redemption 2: Інша гра від Rockstar Games, яка розташована в дикому заході, веде гравця через захоплюючий сюжет та великий відкритий світ.

Dota 2: Ще одна гра в жанрі MOBA, розроблена Valve Corporation. Гравці вбивають монстрів та інших гравців, розвивають своїх героїв та намагаються знищити ворожий Анкс (базу).

1.3 Аналіз ігрових застосунків в жанрі RPG

Історія рольових ігор (RPG) на комп'ютерах налічує понад чотири десятиліття і доповнюється новими досягненнями і інноваціями. Ось короткий огляд історії застосунків в жанрі RPG:

Зародження жанру (1970-1980-ті роки): RPG на комп'ютерах були справжніми піонерами. Однією з перших RPG була "Dungeon & Dragons" (1975) від Gary Gygax і Dave Arneson. У 1974 році, був створений текстовий адвенчурний варіант гри на PDP-10 під назвою "dnd". Ця гра визначила багато елементів механіки і сюжету, які стали типовими для жанру.

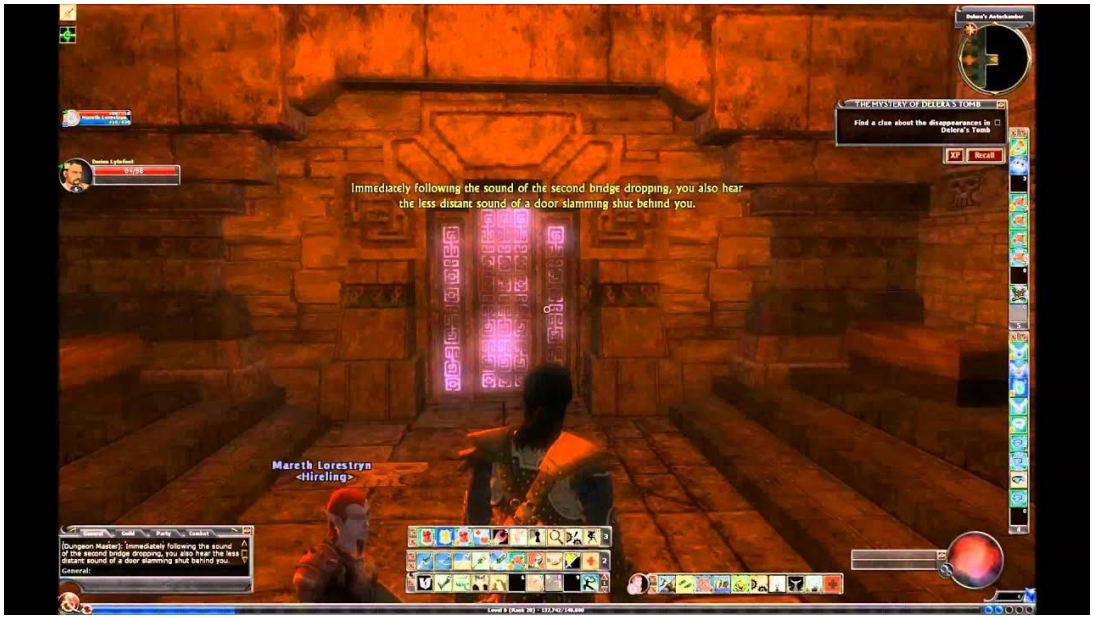


Рисунок 1.1 – "Dungeon & Dragons" (1975) від Gary Gygax і Dave Arneson

Основні характеристики існуючого аналогу:

назва: "Dungeon & Dragons"

- жанр: RPG
- розробник: Gary Gygax і Dave Arneson;

- рік виходу: 1975;

- перелік функцій, характеристик:

- Таблиці та кидки кубів: D&D використовує набір кидків кубів, таких як d20 (двадцятигранник), d12 (дванадцятигранник), d10 (десятигранник), d8 (восьмигранник), d6 (шестигранник) і d4 (четвертигранник), для вирішення різних ситуацій у грі. Велику увагу приділяється кидку d20, яка визначає успіх або невдачу дій персонажів.

- Ігровий майстер (DM): Одним із гравців виступає ігровий майстер, який створює і веде світ гри, створює ігрові ситуації, відтворює персонажів, розробляє сюжет та вирішує, як реагувати на дії гравців. Ця роль ігрового майстра дуже важлива для надання грі структури та наративності.

- Персонажі та розвиток: Гравці створюють персонажів, які мають різні атрибути, такі як сила, ловкість, інтелект, вдача і т. д. Під час гри, персонажі отримують досвід, збагачуються новими навичками, знаходять скарби та зброю. Розвиток персонажів є центральним елементом гри.

- Карта і світ: D&D може відбуватися в будь-якому фентезійному світі, включаючи середньовічну казку, науково-фантастичну фікцію і міфологічні світи. Гра зазвичай відбувається на картах, які представляють локації, включаючи печери, замки, ліси і селища.

- Сюжет та кампанії: D&D часто включає різні сюжети і кампанії, які гравці можуть відвідувати та досліджувати. Кампанії можуть бути створені розробниками або ігровим майстром, і вони можуть бути засновані на різних темах та жанрах.

- Використання уяви: D&D великою мірою ґрунтується на використанні уяви гравців і можливостях креативності. Гравці мають свободу дій і вибору, що робить гру непередбачуваною та захоплюючою.

Жанр RPG (рольових ігор) в комп'ютерних іграх має досить багатий спектр і розвинувся великою кількістю різних застосунків. Ось деякі аспекти, які можна аналізувати у цьому жанрі:

Сюжет і світ гри: Один з найважливіших аспектів RPG - це сюжет та фантастичний світ гри. Як глибокий сюжет, так і детально пророблений світ можуть вплинути на ігровий досвід. Гравці цінують різноманітність історій і світів - від фентезійних світів до постапокаліптичних локацій.

Розвиток персонажів: У багатьох RPG гравці можуть впливати на розвиток свого персонажа, вдосконалюючи навички, вибираючи спеціалізації і приймаючи рішення, які впливають на характер персонажа. Аналіз розвитку персонажів може включати в себе оцінку системи розвитку, балансу і глибини.

Бойова система: Бойова система в RPG може бути різноманітною. Деякі ігри акцентуються на реальному часі, інші на ходовій системі. Аналізувати бойову систему слід з точки зору співгравців, складності і стратегічності.

Графіка та атмосфера: Графіка та атмосфера гри можуть суттєво вплинути на гравців. Важливо дослідити якість графіки, дизайну світу та атмосферу, яка допомагає відчути іммерсію в ігровому світі.

Задачі та квести: Багато RPG мають завдання і квести для гравців. Важливо аналізувати ці завдання на предмет складності, оригінальності та впливу на сюжет.

Моральні вибори: Деякі RPG надають гравцям можливість приймати моральні рішення, які впливають на розвиток сюжету та відносини з іншими персонажами. Ці моральні аспекти можна проаналізувати на предмет глибини і наслідків.

Мультиплеєр і співгра: Деякі RPG мають мультиплеєрний режим або можливість гри в кооперативі. Аналізувати можливості співгри та взаємодії гравців є важливим аспектом.

Підтримка і додатковий контент: Підтримка гри розробниками, випуск патчів і додаткового контенту також має велике значення для гравців.

Аналізувати цю підтримку може допомогти в оцінці якості і тривалості ігрового досвіду.

Зазначені аспекти допомагають аналізувати ігрові застосунки в жанрі RPG, і розробники роблять акценти на різних аспектах, створюючи різні ігри в цьому жанрі для задоволення різних смаків гравців.

1.4 Специфіка вимог до програмного забезпечення комп’ютерної гри

Вимоги до програмного забезпечення комп'ютерної гри включають в себе ряд параметрів і характеристик, які повинні бути враховані під час розробки гри. Ось деякі основні специфікації та вимоги, які розробники враховують при створенні комп'ютерних ігор:

Платформи: Розробники повинні визначити, для яких платформ (комп'ютери, ігрові консолі, мобільні пристрої тощо) буде доступна гра. Вимоги до обладнання можуть суттєво відрізнятися в залежності від платформи.

Операційна система: Гра повинна бути сумісною з певними версіями операційних систем. Наприклад, гра може бути розроблена для Windows, macOS або Linux.

Процесор і пам'ять: Гра повинна працювати на комп'ютерах з різними типами процесорів і кількістю оперативної пам'яті. Розробники вказують мінімальні та рекомендовані системні вимоги.

Графічна підсистема: Ігри зазвичай потребують графічного обладнання, яке підтримує певні технології, такі як DirectX або OpenGL. Деякі ігри також можуть мати високі вимоги до графічних карт.

Звукова система: Гра має підтримувати різні аудіо-платформи і аудіо-кодеки. Це важливо для якісної звукової гри.

Роздільна здатність екрану: Розробники вказують мінімальну та рекомендовану роздільну здатність екрану для гри. Це важливо для оптимального відображення графіки.

Управління: Гра повинна підтримувати різні способи управління, такі як миша, клавіатура, геймпади або сенсорні екрани, в залежності від платформи.

Мови та регіональні вимоги: Ігри повинні підтримувати різні мови та регіональні налаштування, включаючи відображення тексту, голосового супроводу та культурні особливості.

Завантаження і пам'ять: Гра повинна оптимізувати використання пам'яті та завантаження ресурсів для забезпечення безперервної гри.

Мережева гра: Якщо гра має мережевий режим, вимоги до мережевого підключення та підтримки мережевого геймплею також повинні бути визначені.

Безпека і захист даних: Забезпечення безпеки гри та захисту даних гравців є важливою складовою вимог до програмного забезпечення.

Підтримка оновлень і патчів: Гри повинні мати системи оновлень та патчів для виправлення помилок та вдосконалення гри.

Ці вимоги варіюються в залежності від жанру гри, її масштабу та платформи. Розробники зазвичай публікують ці вимоги для того, щоб гравці могли переконатися, що їхні комп'ютери або консолі можуть запускати гру відповідним чином.

2 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНОЇ ГРИ

2.1 Діаграма прецедентів системи

Технічне завдання (ТЗ) визначає характеристики, методи і структуру проекту. Воно формується на підставі результатів фахового аналізу та досліджень з урахуванням вимог клієнтів, особливостей бізнес-процесів у найменших деталях. Щоб пояснити це простими словами, ТЗ - це докладний посібник, який визначає чіткі та послідовні методи створення продукту, а також обладнання та технології, які будуть використані, і передбачувані результати.

Програмне забезпечення гри «Arden Dungeon» що проєктується, має задовольнити потреби гравців та має систематизувати взаємодію системи із користувачем.

Система повинна містити такі сторінки:

- Екран з меню та анімацією початку гри;
- Головна сторінка процесу гри;
- Екран з підфищенням хар.; · Екран з уроном;
- Сторінка програшу; · Сторінка виграшу.

2.2 Діаграма станів

Сценарій використання, також відомий як короткий сценарій, демонструє, як одна або декілька осіб чи організацій взаємодіють з технологією в реальному житті. Вони описують події, послідовність кроків та/або дії, які відбуваються під час цієї взаємодії. Сценарії використання можуть бути докладними, роз'яснюючи, як саме користувач взаємодіє з інтерфейсом, або вони можуть бути загальними, визначаючи ключові бізнес-завдання, не глибоко занурюючись в технічні деталі.

У розробці гри, визначено дві ролі: користувач і система. Система виконує свої завдання на програмному рівні і контролює процес функціонування додатку. Основний сцерарій поділений на акти використання системи:

(АКТ1)Фермер помічає що гоблін краде його корову, в надії повернути її він намагається наздогнати його через ліс, натикаючись на зграю вовків котрі вбили мисливця. Підбирає лук мисливця та відбивається від вовків. Далі продовжує переслідувати гобліна, сліди якого ведуть до печери.

(АКТ2)Біля входу дл печери фермер натрапляє на гоблінів допоки не відбивається і звільняє собі вхід до печери. Блукаючи лабіринтами печери фермер знаходить вхід до великої печери.

(АКТ3)Посеред печери стоїть велетенський гоблін якому вдалося підкорити гоблінів щоб ті виконували його накази, фермеру потрібно вступити в бій щоб врятувати його корову.

Для отримання уявлення про можливості користувача, можна створити діаграму використання або діаграму прецедентів. Діаграма варіантів використання є інструментом візуалізації системи та взаємодій її користувачів. Зазвичай ця діаграма представляється як графічне зображення взаємодій між різними компонентами системи. Діаграми варіантів використання дозволяють описати події, що відбуваються в системі та їх послідовність, але не концентруються на технічних аспектах їх виконання.

Основні компоненти діаграми станів включають наступне:

Стани: Стани представляють окремі фази або стани, в яких може перебувати об'єкт або система.

Переходи: Переходи вказують на зміну стану з одного стану в інший. Вони специфікують умови або події, за яких такий перехід стає можливим.

Події: Події вказують на події або дії, які викликають переходи між станами. Вони можуть бути ініційовані зовнішніми чинниками або внутрішніми процесами.

Дія: Дії асоційовані з конкретними станами або переходами і вказують, що система робить в конкретному стані або під час переходу.

На рис. 2.1 зображена діаграма використання гри, що розробляється.

Рисунок 2.1 – Діаграма варіантів використання

2.3 Вибір програмних засобів для розробки

Розробка веб-застосунків вимагає ретельного відбору технологій та методологій. Ключовим аспектом є також зрозуміти, яким чином були обрані ці технології для конкретного проекту.

При виборі технологій слід враховувати наступні аспекти:

- Розмір та складність проекту;
- Темпи розвитку;
- Вартість і доступність фахівців;
- Тенденції в сфері розробки;
- Наявність документації;
- Витрати на підтримку;
- Вимоги до навантаження;
- Системи безпеки;
- Можливості інтеграції з іншими рішеннями.

Unreal Engine - це потужний ігровий двигун, створений компанією Epic Games. Його використовують для розробки ігор на різних платформах, від ПК до консолей і мобільних пристроїв. Unreal Engine пропонує широкі можливості для розробки відмінних візуальних ефектів, фізики, штучного інтелекту та інших ключових аспектів геймдеву.

Двигун має свою власну візуальну програмувальну систему, яка називається Blueprints, що дозволяє створювати гру без написання коду. Також Unreal має потужну систему роботи з 3D-графікою та відмінну підтримку віртуальної реальності. Багато відомих ігор, таких як Fortnite, Gears of War, а також інші проекти, були створені на базі Unreal Engine.

Створений мовою C++, рушій Unreal Engine дозволяє створювати ігри для різних операційних систем і платформ, таких як Microsoft Windows, Linux, Mac OS і Mac OS X, а також для консолей Xbox, Xbox 360, PlayStation 2, PlayStation Portable, PlayStation 3, Wii, Dreamcast і Nintendo GameCube. У минулому році Марк Рейн показав можливості рушія Unreal Engine 3 на iPod Touch і iPhone 3GS. А в березні 2010 року демонстрація роботи рушія відбулася на комунікаторі Palm Pre, що базується на мобільній платформі webOS.

Для спрощення адаптації рушія до різних середовищ використовується модульна система залежних компонентів. Ця система підтримує різні системи рендерингу (Direct3D, OpenGL, Pixomatic, раніше підтримувалися Glide API, S3 Metal, PowerVR SGL), відтворення звуку (EAX, OpenAL, DirectSound3D, раніше підтримувалися A3D), інструменти голосового відтворення тексту, розпізнавання мовлення (для Xbox360, PlayStation 3, Nintendo Wii і Microsoft Windows, планувалося також для Linux і Mac), модулі для роботи з мережею і підтримка різних пристроїв вводу.

Для гри в мережі підтримуються технології Windows Live, Xbox Live і GameSpy, з можливістю до 64 гравців одночасно. Навіть якщо офіційні засоби розробки не містять підтримки великої кількості клієнтів на одному сервері, рушій використовувався для створення MMORPG-ігор. Наприклад, Lineage II, один з відомих представників цього жанру, використовує рушій Unreal Engine.

Усі компоненти ігрового рушія представлені у вигляді об'єктів з власним набором характеристик. Кожен клас є "дочірнім" класом об'єкта. Серед основних класів і об'єктів можна виділити такі:

- Актор (actor) - базовий клас, що містить усі об'єкти, які відносяться до ігрового процесу й мають просторові координати.
- Пішак (pawn) - фізична модель гравця або об'єкта, керованого штучним інтелектом.
- Світ, рівень (world, game level) - об'єкт, що характеризує загальні властивості "простору", де розташовуються всі актори.

Щодо обробки простору, використовується бінарна розбивка простору на "заповнену" і "порожню" частини. Актори, що потрапляють в "заповнену" частину, припиняють своє існування.

Це дозволяє створювати різні типи поверхонь, які утворюють бінарне дерево простору. Групи елементів цього дерева називаються нодами, які впливають на продуктивність промальовування сцен. Також використовуються об'єми - частини простору з властивостями, відмінними від ігрового світу.

C++ це потужна мова програмування, яка широко використовується у розробці програмного забезпечення. C++ була розроблена на основі мови програмування C, тож у неї є багато спільних рис із C, але вона також має додаткові можливості.

Однією з ключових особливостей C++ є підтримка об'єктно-орієнтованого програмування (ООП). Це означає, що програми на C++ можуть бути побудовані на основі об'єктів, які мають властивості та методи. Це дозволяє створювати більш структуровані, модульні та легко зрозумілі програми.

C++ також володіє великою ефективністю і швидкодією, що робить її популярним вибором для розробки системного програмного забезпечення, ігор, додатків реального часу та інших вимогливих застосувань. Вона надає багатий набір вбудованих функцій і бібліотек для роботи з файлами, мережами, текстом та багатьма іншими аспектами програмування.

Ще однією важливою особливістю C++ є можливість використання низькорівневих операцій, таких як керування пам'яттю, що дає розробникам

більше контролю над програмою, але водночас може вимагати більшої уваги до деталей.

Мова C++ постійно розвивається, з'являються нові функції та можливості, що робить її актуальною для широкого кола завдань у сфері програмування.

Blueprints - це потужна нова функція, представлена в Unreal Engine 4. Blueprints - це спосіб створення нових UClasses без необхідності писати або компілювати код. При створенні Blueprint ви можете вибрати розширення класу C++ або іншого класу Blueprint. Після цього ви можете додавати, розташовувати та налаштовувати компоненти, реалізувати власну логіку за допомогою візуальної мови скриптування, реагувати на події та взаємодії, визначати власні змінні, обробляти ввід та створювати повноцінний власний тип об'єкту.

У кожному Blueprint є Construction Script, аналогічний конструктору в C++, який виконується при створенні об'єкта. Цей скрипт може динамічно конструювати екземпляр актора на основі будь-якої кількості факторів, наприклад, створювати паркан, який автоматично розраховує свій розмір для заповнення проміжку між будівлями. З цієї точки зору Blueprint може бути розглянутий як дуже потужна система префабрикації.

Система візуального скриптування Blueprint в Unreal Engine - це повна система скриптування геймплею, яка базується на концепції використання інтерфейсу на основі вузлів для створення геймплейних елементів прямо в Unreal Editor. Як і у багатьох розповсюджених мовах скриптування, вона використовується для визначення об'єктно-орієнтованих (ОО) класів чи об'єктів в двигуні.

Ця система надзвичайно гнучка та потужна, оскільки дозволяє дизайнерам використовувати практично всі поняття та інструменти, які, як правило, доступні лише програмістам. Крім того, специфічна розмітка, доступна в реалізації Unreal Engine на C++, дозволяє програмістам створювати базові системи, які можуть бути розширені дизайнерами.

У данній роботі використано міксування коду написаного у C++ і Blueprints в Unreal Engine.

Міксування коду і блупрінтів (Blueprints) в Unreal Engine - це потужний спосіб поєднання візуального програмування з кодуванням для створення складних систем та функціоналу у грі.

Техніка міксування може виглядати наступним чином:

- Використання Blueprints для базового функціоналу: Блупрінти - це графічний інтерфейс для створення функцій, об'єктів та ігрових систем без необхідності писати код. Ви можете створювати основні елементи гри, такі як рух гравця, взаємодія з об'єктами, анімації, реакції на події тощо за допомогою візуального програмування.
- Інтеграція коду через C++: Для більшої гнучкості і швидкості виконання складних операцій можна використовувати C++. Ви можете створювати власні функції, класи та системи у C++ та інтегрувати їх у ваш проект Unreal Engine.
- Використання BlueprintCallable та BlueprintImplementableEvent: Ці маркери дозволяють створювати код у C++, який можна викликати або реалізувати через блупрінти. Це означає, що ви можете написати певний код у C++ та викликати його у блупрінтах або навпаки - створити функцію у блупрінтах, яку реалізувати у C++.
- Налаштування параметрів через BlueprintExposed: Це дозволяє використовувати змінні та параметри, які ви створили у C++, в блупрінтах для подальшої настройки та використання.
- Створення складних систем: Часто міксування включає створення складних систем або ігрових механік у блупрінтах, але використання коду для оптимізації та додаткової функціональності цих систем.

Цей підхід дозволяє використовувати найкращі аспекти візуального програмування у блупрінтах та потужні можливості мови програмування C++ для створення ваших ігрових систем у Unreal Engine.

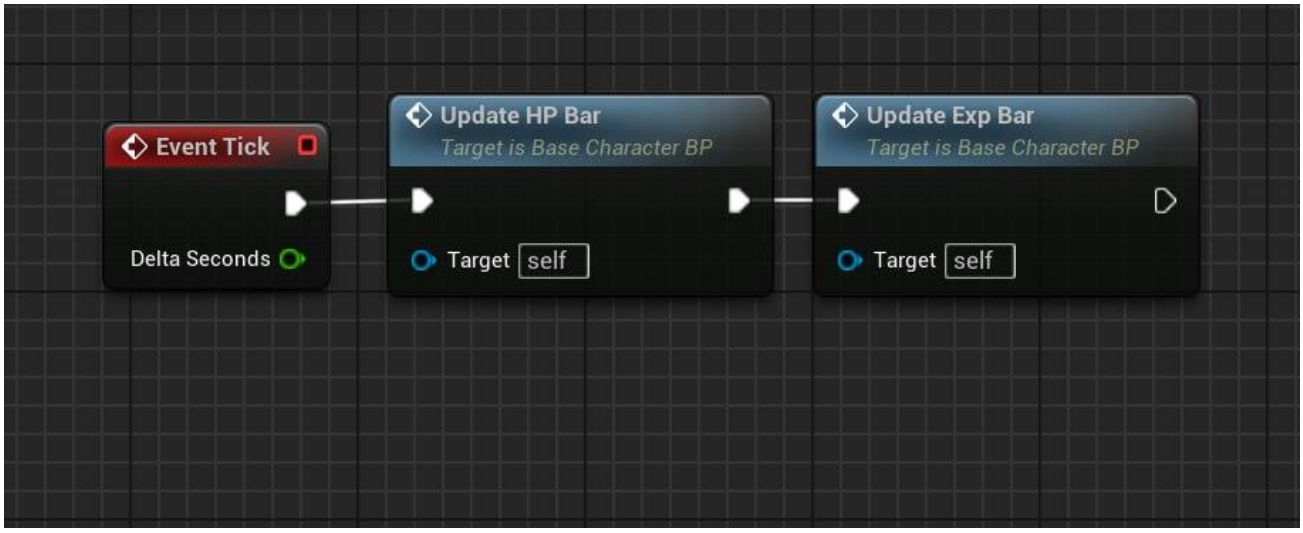


Рисунок 2.2 – Прикалад блупрінта визваний функціями які написані на C++

2.4 Опис можливостей гри та використаних технологій

Створена гра розроблена на Unreal Engine 5 використовуючи базові безкоштовні ассети, які були представлені ігровим двигуном.

Unreal Engine надає широкий набір можливостей для розробки ігор та інтерактивних додатків, і використовує різні передові технології. Ось кілька ключових можливостей та технологій, які ви можете знайти в Unreal Engine:

1. Графіка високої якості: Unreal Engine включає в себе потужну систему рендерингу, що підтримує фізично-правильний рендеринг (PBR), обчислювальне підтримання відбитку світла, глобальне освітлення, HDR-рендеринг та інші технології для створення візуально реалістичних сцен.
2. Фізика: Unreal Engine включає фізичний рух та симуляцію, що дозволяє створювати реалістичну поведінку об'єктів, обробляти зіткнення та реагувати на фізичні закони.
3. Анімація: Движок підтримує роботу зі скелетною анімацією та анімацією об'єктів. Ви можете створювати анімовані персонажі, об'єкти та об'єкти оточення.
4. Штучний інтелект: Unreal Engine має вбудовану підтримку для роботи з штучним інтелектом (ШІ). Ви можете програмувати поведінку ворожих персонажів, NPC, агентів та інших об'єктів у вашій грі.

5. Звук: Движок підтримує роботу з аудіо, включаючи 3D звук та обробку звуку в реальному часі для створення атмосферних звукових ефектів.
6. VR та AR: Unreal Engine має підтримку віртуальної реальності (VR) та аугментованої реальності (AR). Ви можете створювати ігри та додатки для різних VR- та AR-платформ.
7. Фізично-правильне освітлення: Unreal Engine використовує фізично-правильне освітлення для створення реалістичних тіней та відображення світла на поверхнях.
8. Можливості многокористувацької гри: Unreal Engine надає інструменти для розробки мультиплеєрних ігор, включаючи сервер-клієнт архітектуру та підтримку мережевого геймплею.
9. Широка кросплатформеність: Ви можете розробляти ігри та додатки на різних платформах, включаючи PC, консолі, мобільні пристрої, VR-гарнітури та інші.
10. Можливості розширення: Unreal Engine підтримує плагіни та розширення, що дозволяють розробникам додавати власні функціональність та інструменти.

Створена гра володіє такими характеристиками:

1. Переміщення (базове переміщення, стрибки, прискорення)
2. Стрільба
3. Зброя - лук
4. Роздільна здатність
5. Якість
6. Повноекранний режим
7. Чутливість прицілювання
8. Фізично-правильне освітлення
9. Штучний інтелект

3 ПРОЕКТУВАННЯ ІНТЕРФЕЙСУ ГРИ

3.1 Принципи проєктування інтерфейсів ігрових застосунків

Проєктування інтерфейсу ігрового додатка є важливою частиною розробки гри, оскільки інтерфейс взаємодії гравця з грою може значно вплинути на користувачський досвід.

Дизайн інтерфейсу ігрового додатка грає важливу роль у створенні приємного та зручного геймплею для гравців. Нижче наведено кілька кроків і рекомендацій для дизайну інтерфейсу ігрового додатка:

Визначення цілей: Почніть зі зрозуміння цілей вашого ігрового інтерфейсу. Що гравцям потрібно зробити та яку інформацію вони повинні отримати? Визначення цілей допоможе зосередитися на важливих елементах.

Інтерфейсні елементи: Визначте, які інтерфейсні елементи будуть використовуватися в грі, такі як кнопки, меню, панелі, іконки тощо. Розгляньте, як ці елементи будуть виглядати та де розміщені на екрані.

Кольори та стиль: Виберіть кольорову палітру та стиль для ігрового інтерфейсу. Кольори та стиль повинні відповідати тематиці гри та створювати затишну атмосферу.

Читабельність: Забезпечте читабельність тексту та інформації на екрані. Вибирайте чіткі шрифти та забезпечуйте достатній контраст між текстом та фоном.

Розмір та масштаб: Враховуйте розмір та масштаб інтерфейсу для різних пристроїв і роздільних здатностей екрану. Інтерфейс повинен бути добре видимим та користуватися зручною на будь-якому пристрої.

Анімація: Використовуйте анімацію для виділення важливих подій, таких як натискання кнопок, отримання досягнень або зміни стану гри. Але будьте обережні, щоб не перевантажувати інтерфейс зайвими анімаціями.

Просторова організація: Розмістіть інтерфейсні елементи логічно та зручно для гравців. Наприклад, головні дії повинні бути легко доступні, а інформація про стан гри повинна бути видимою.

Тестування та зворотній зв'язок: Проведіть тестування інтерфейсу з реальними гравцями та зберіть їхні відгуки. Це допоможе виявити недоліки та поліпшити дизайн.

Адаптація: Розгляньте можливість створення адаптивного інтерфейсу, який пристосовується до різних умов гри. Наприклад, інтерфейс може змінюватися під час бою чи інших геймплейних ситуацій.

Зручність використання: Гарантуйте, що інтерфейс легкий у використанні та не заважає гравцям насолоджуватися грою. Використовуйте інтуїтивні рішення та мінімізуйте кількість кроків для досягнення результату.

Загалом, дизайн інтерфейсу ігрового додатка повинен підкреслювати геймплей і допомагати гравцям насолоджуватися грою.

Дана грана була розробленна у стилі low poly.

"Low poly" - це стиль в комп'ютерних іграх і графіці, який характеризується використанням обмеженої кількості полігонів (геометричних фігур) для створення об'єктів та оточення. На рис. 3.1 та 3.2 зображено середовище Low poly

рис. 3.1, 3.2

Цей стиль популярний у графіці, оскільки він надає грі специфічний вигляд і може бути особливо естетично привабливим. Ось деякі основні риси стилю low poly:

Мінімалізм: В low poly-графіці використовується обмежена кількість полігонів, що призводить до спрощених геометричних форм та об'єктів. Це створює мінімалістичний вигляд.

Яскраві кольори: Часто low poly-графіка використовує яскраві та насичені кольори, щоб виділити об'єкти та зробити їх більш помітними на фоні.

Грані та ребра: У low poly-графіці можна часто побачити виділені грані та ребра об'єктів. Це може надати їм художній вигляд і відзначити геометричну структуру.

Спрощена текстура: Low poly-графіка часто включає спрощені текстури, які можуть включати пласкі кольори або малюнки низької роздільної здатності.

Ретро-відчуття: Багато low poly-графіки натякає на ретро-стиль, нагадуючи графіку старих ігор або ігрових консолей.

Легка анімація: Завдяки спрощеному об'єму обчислень, low poly-графіка може бути легше анімованою, що робить її відмінним вибором для інді-розробників.

Мистецька свобода: Цей стиль дозволяє художникам більшу творчу свободу і експериментування з формами та кольорами.

Low poly-стиль може використовуватися в різних жанрах ігор, від інді-проектів до ігор з великим бюджетом. Він може створювати цікавий та запам'ятовуючий візуальний стиль, який відзначається своєю простотою та оригінальністю.

3.2 Стратегії розробки інтерфейсів ігрових застосунків

Відеоігри, переважно розглядаються як творча платформа для інтерактивного розповідання, проте з технічної точки зору, це просто програмне забезпечення, яке вимагає переважно стандартних, іноді - сучасних способів взаємодії користувача. Навіть у найпростіших іграх, вам потрібно використовувати меню навігації для запуску, завантаження або збереження гри.

Розробка інтерфейсу гри - важливий етап у створенні ігрового додатка, який визначає, як гравець буде взаємодіяти з грою та отримувати необхідну інформацію. Ось декілька стратегій розробки інтерфейсу ігрових застосунків:

Спрощення та мінімалізація: Спробуйте уникнути перенавантаження інтерфейсу з надмірними елементами. Зосередьтеся на основних функціях та інформації, яка дійсно необхідна гравцеві для гри.

Аналіз користувача: Розгляньте потреби та очікування вашої цільової аудиторії. Які інтерфейсні елементи та функціонал найбільш важливі для них?

Прототипування: Створіть прототип інтерфейсу, щоб визначити основну структуру та функціонал. Це допоможе вам візуалізувати, як гравець буде навігувати та взаємодіяти з грою.

Контекстна інформація: Забезпечте гравця корисною інформацією відповідно до ситуації у грі. Наприклад, підказки під час завантаження, інструкції під час нових завдань тощо.

Інтуїтивність: Зробіть інтерфейс інтуїтивно зрозумілим для гравця. Використовуйте стандартні символи та вказівники, щоб полегшити навігацію.

Анімація: Використовуйте анімацію, щоб підкреслити важливі події або взаємодії гравця з елементами інтерфейсу.

Адаптація до платформ: Враховуйте особливості різних платформ, на яких гра буде доступна (комп'ютер, мобільні пристрої, консолі) та адаптуйте інтерфейс відповідно.

Тестування з гравцями: Проводьте тестування прототипів та готового інтерфейсу з реальними гравцями, щоб отримати їхні відгуки та виявити можливі покращення.

Оптимізація продуктивності: Забезпечте, щоб інтерфейс працював ефективно та мінімізував навантаження на систему гравця.

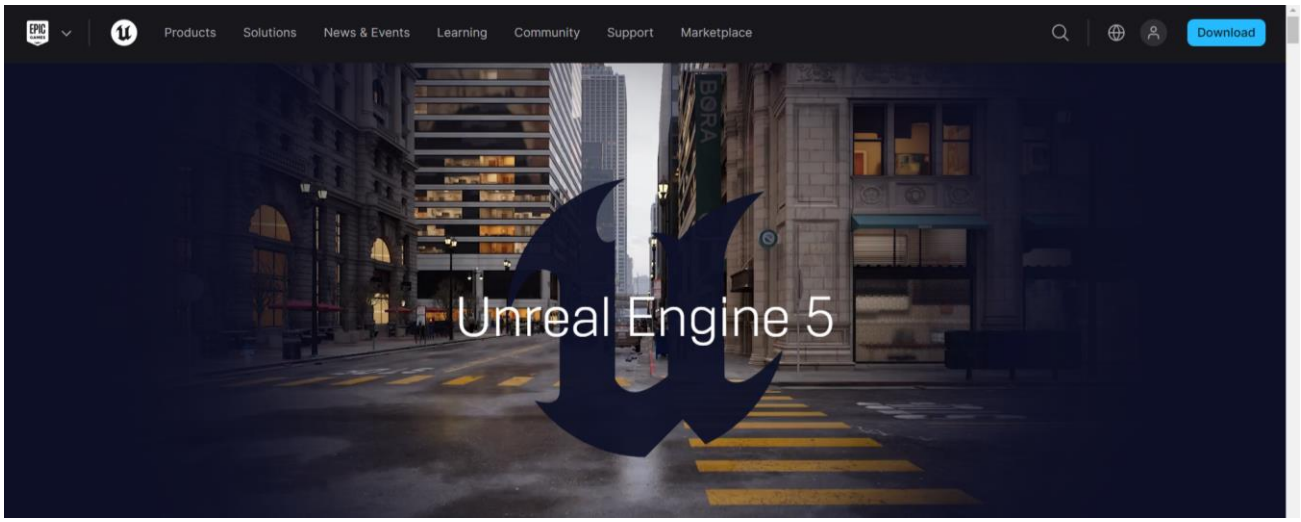
Навчання гравця: Забезпечте можливість навчання гравця базовим елементам інтерфейсу, особливо якщо гра має складний або унікальний геймплей.

Загалом, розробка інтерфейсу гри повинна бути спрямована на створення зручного та ефективного інструменту для взаємодії гравця з грою, допомагаючи йому насолоджуватися геймплеєм.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ ГРИ

4.1 Початок та опис програмного функціоналу

Для встановлення Unreal Engine 5 використовується програма Epic Games Launcher. Для цього треба перейти на веб-сайт Unreal Engine та натиснути кнопку "Download" у верхньому правому куті.



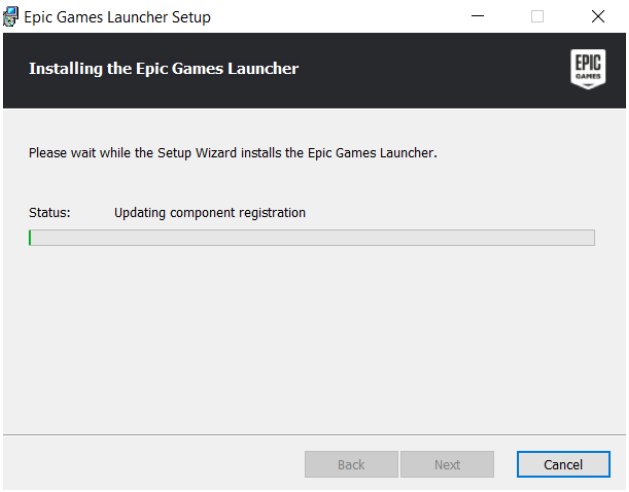


Рис. 4.1.1 та 4.1.2- Download and Installing

Далі отрібно буде створити обліковий запис, перш ніж почати запуск.

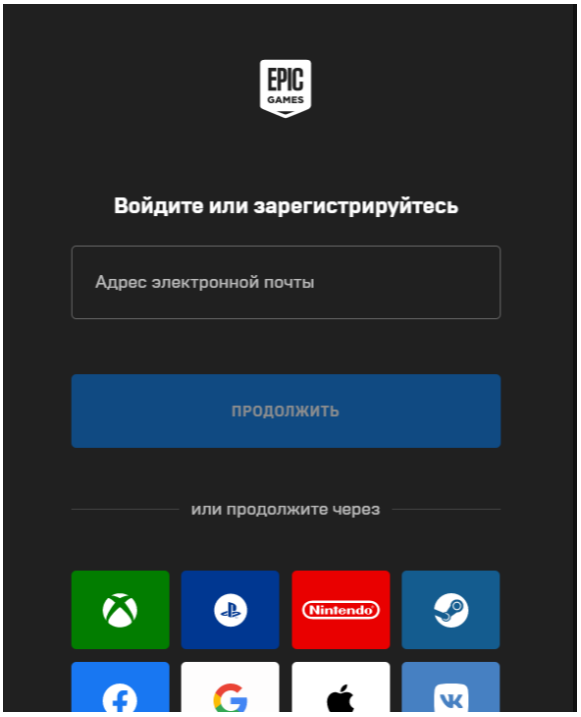


Рис. 4.1.3 – Створення облікового запису

Ввести електронну адресу та пароль, та натиснути “Увійти”. Після входу з'явиться таке вікно:

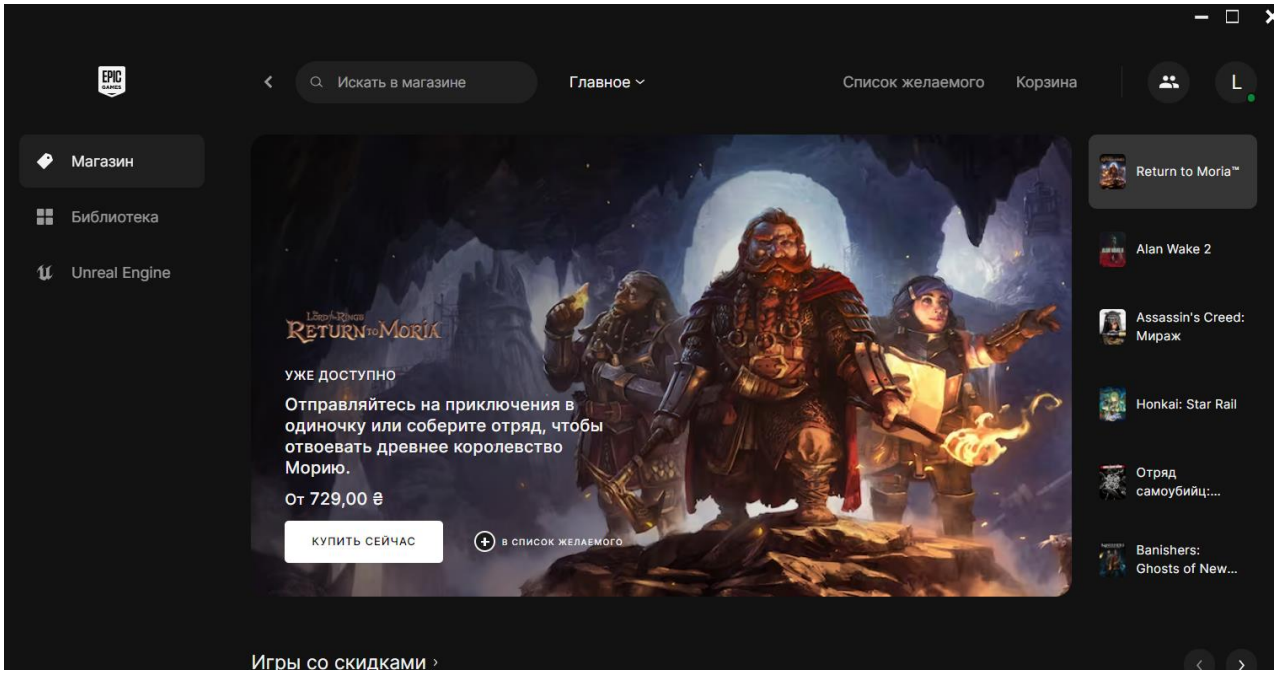


Рис. 4.1.4 - Головне вікно Epic Games

Перейти на табу Unreal Engine

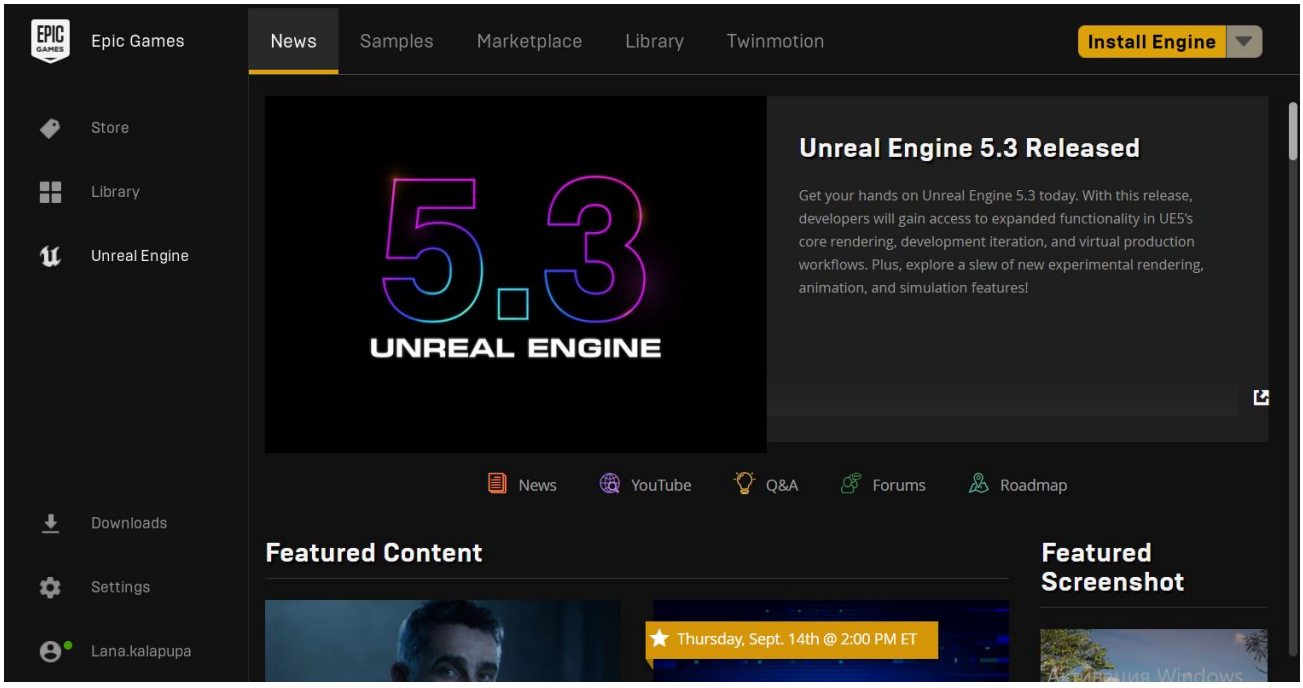


Рис. 4.1.5 - вікно Unreal Engine

У верхньому лівому куті, натиснути кнопку “Встановити двигун”.

Натиснути кнопку “Встановити”.

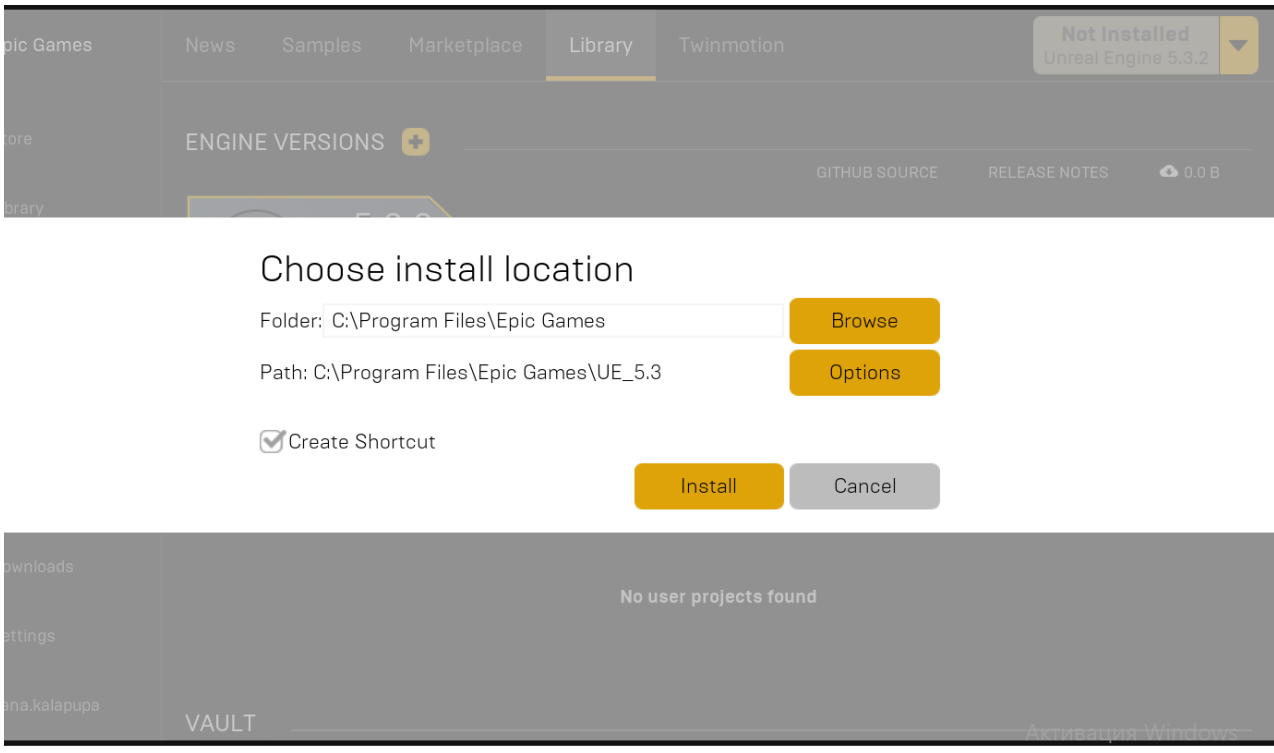


Рис. 4.1.6 – вікно Запуску

Після завершення встановлення двигун з'явиться у бібліотеці.

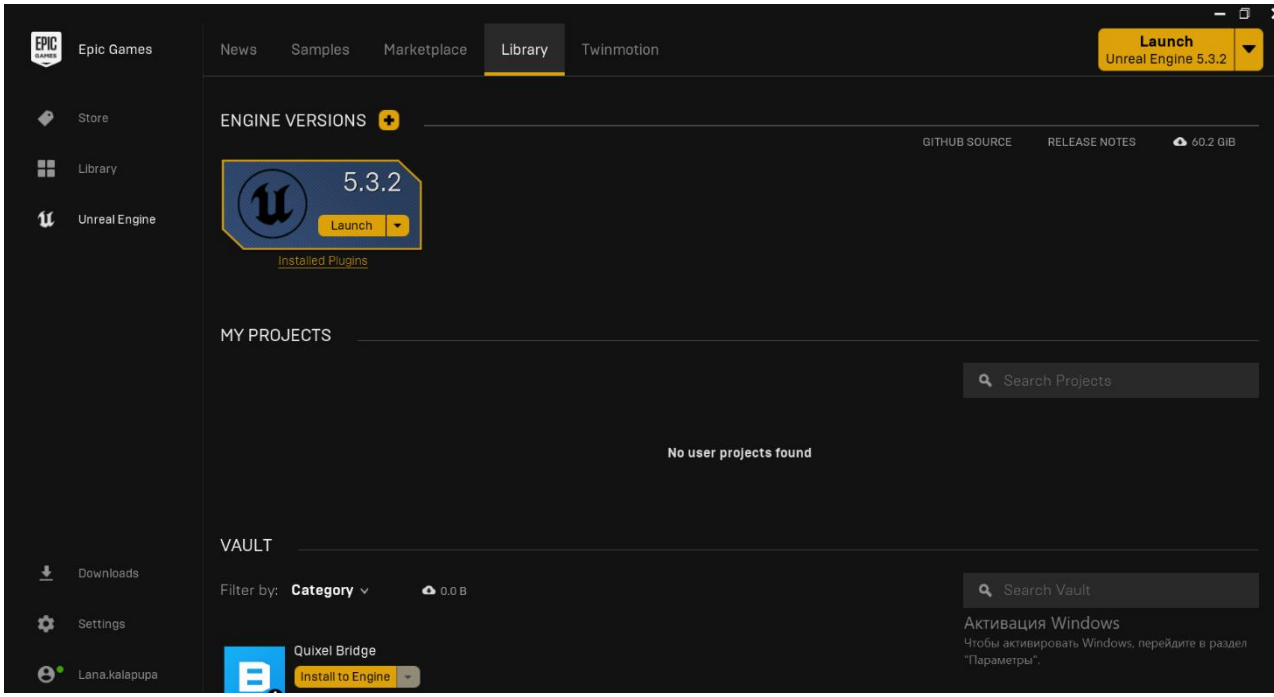


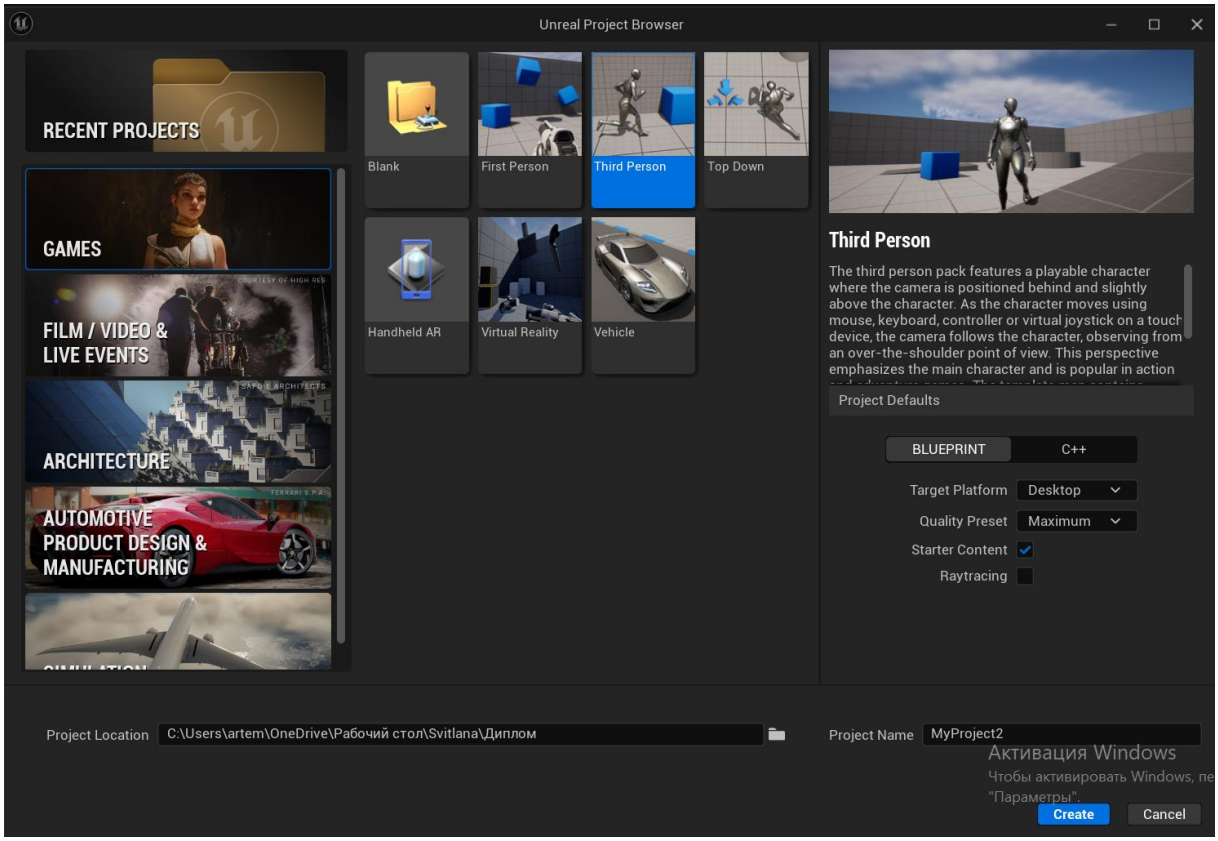
Рис. 4.1.7 - Бібліотека Epic Games

Натиснути одну з кнопок запуску, щоб відкрити браузер проекту. Коли він відкриється, перейти на вкладку “Новий проект”.

Обрати шаблон: Створюємо шаблон проекту Third Person, в ньому нам знадобляться базові налаштування гри від третьої особи. Ми будемо використовувати і доповнювати базовий функціонал блупрінт персонажу, анімаційний блупрінт і контролер управління персонажем(BP Third Person, ABP Third Person, Third Person Controller)

Називаємо проект: Вводимо назву для нашого проекту.

Вибераємо розташування: Обераємо папку для збереження нашого проекту на нашому комп'ютері.



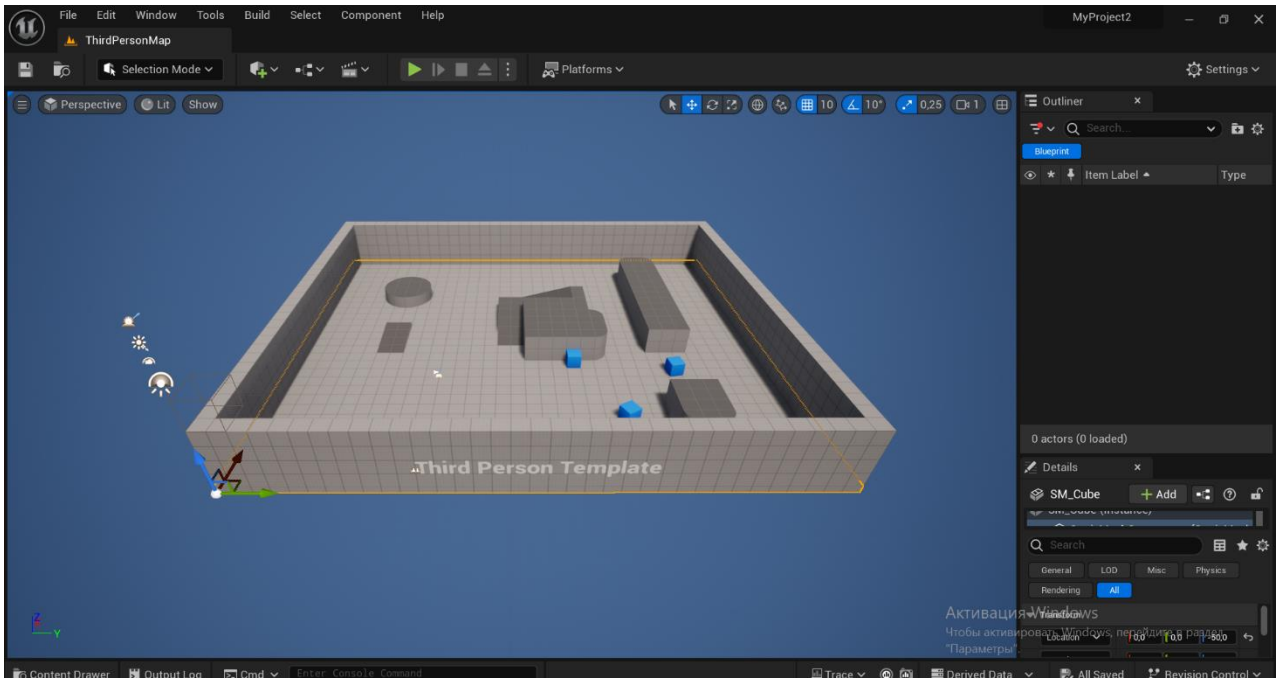


Рис. 4.1.8 та 4.1.9 - Створення проекту

Обераємо налаштування: Після створення проекту треба створити C++ класс, який буде батьківським класом для нашого персонажу.

Перейдемо в вкладку Tools - New C++ Class

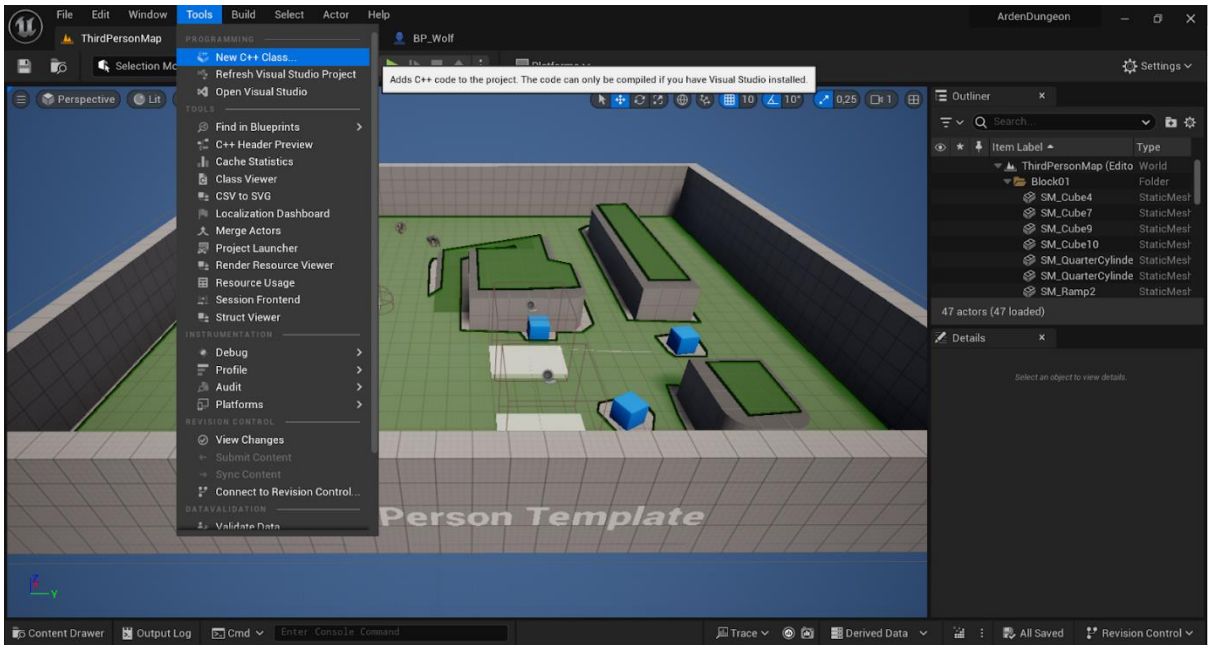
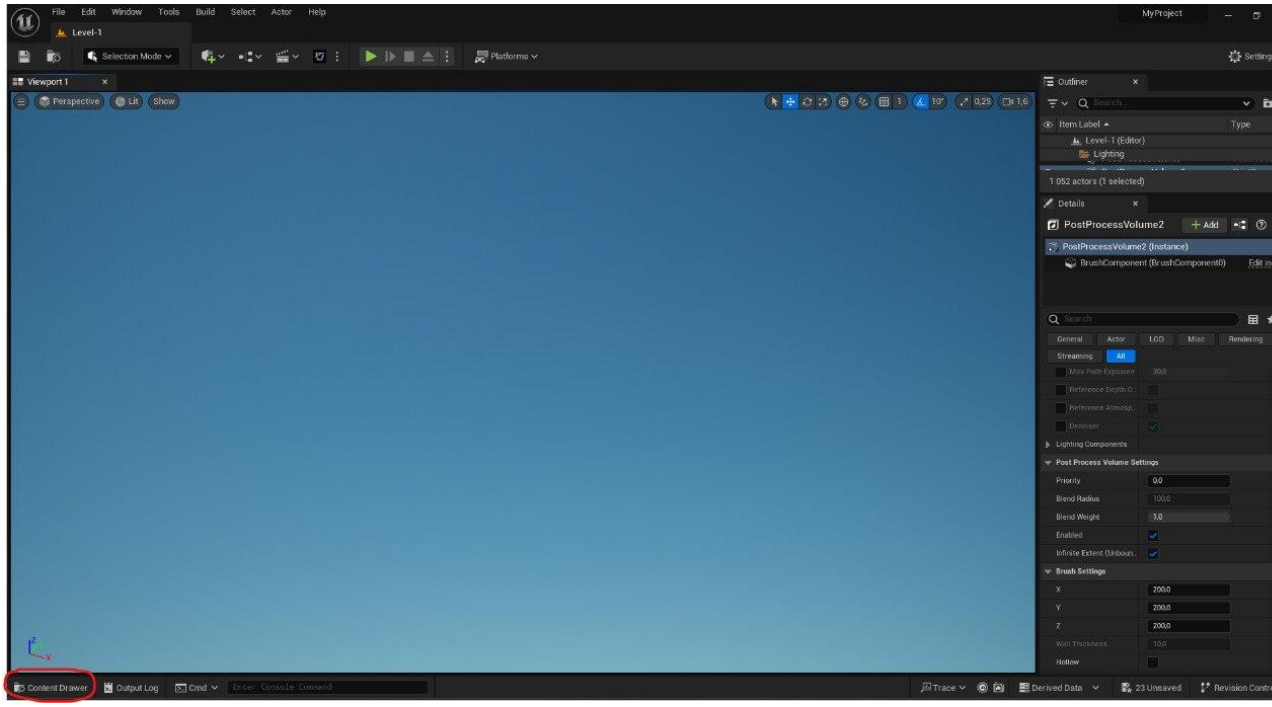


Рис. 4.1.10 – Створення C++ классу

Після створення проекту опиняємося у середовищі Unreal Engine. Далі ознайомлюємося з основними вікнами і панелями, такими як вікно перегляду світу (Viewport), панель ресурсів (Content Browser), панель деталей (Details Panel) тощо

4.2 Інтерфейс, імпорт та додавання мешів на рівень

Перш ніж Unreal зможе використовувати будь-які файли, їх потрібно імпортувати. Перейти до браузера натиснувши кнопку “Content Drawer” змісту далі створити папку та натиснути кнопку “Імпортувати”.



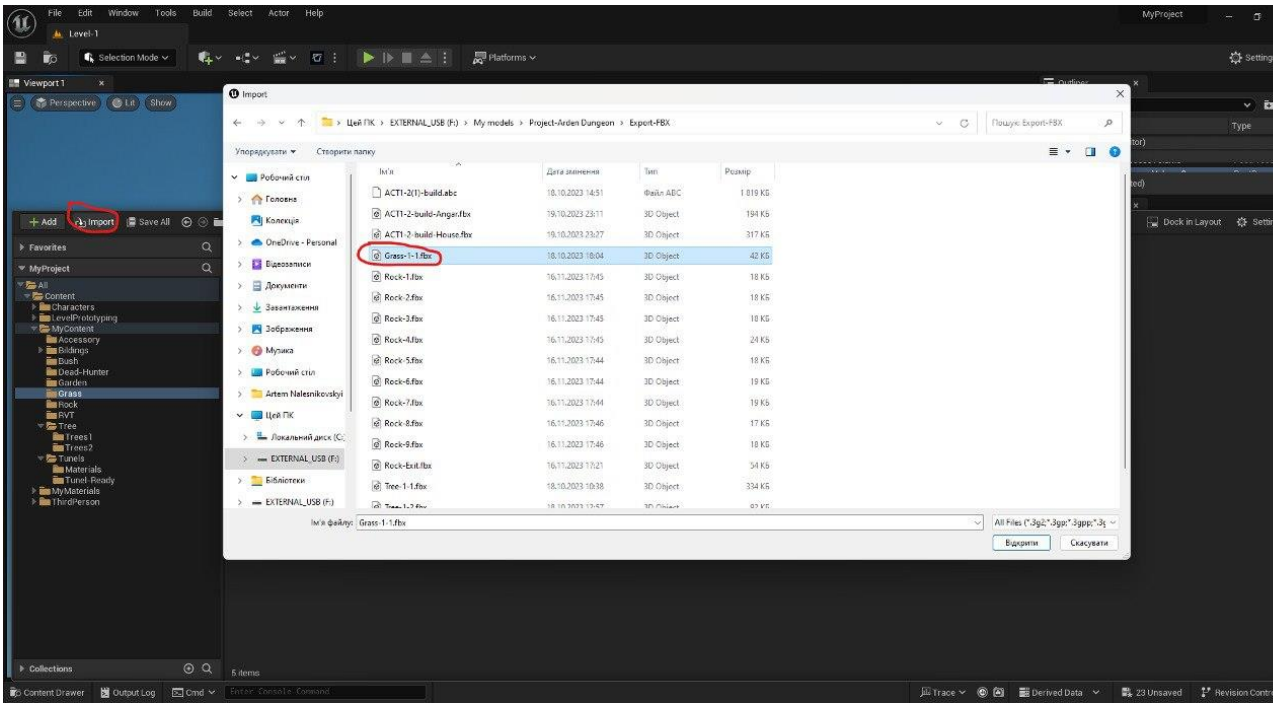


Рис. 4.2.1 та 4.2.2 - Імпортування

Unreal дає деякі параметри імпорту для файлу .fbx. Треба переконатися, що у випадаючому списку буде обрано “Create new materials див. рис 4.2.3

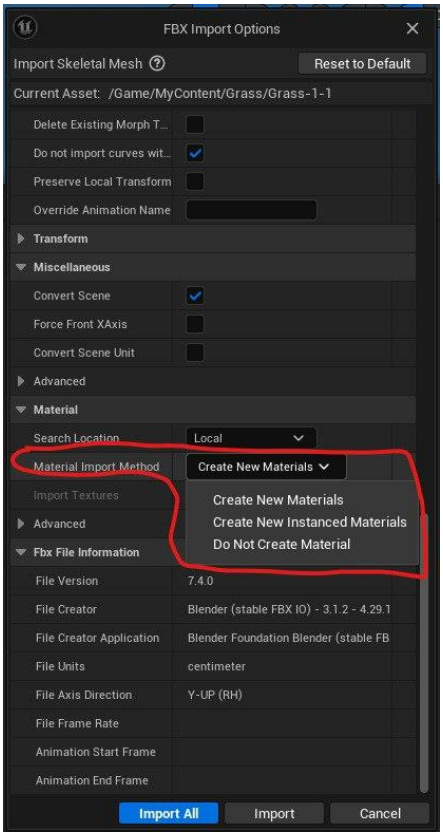


Рис. 4.2.3

Всі моделі були створені в програмі Blender та підготовлені заздалегіть:

1.Characters:

-Poly_Man (рис.4.2.4)

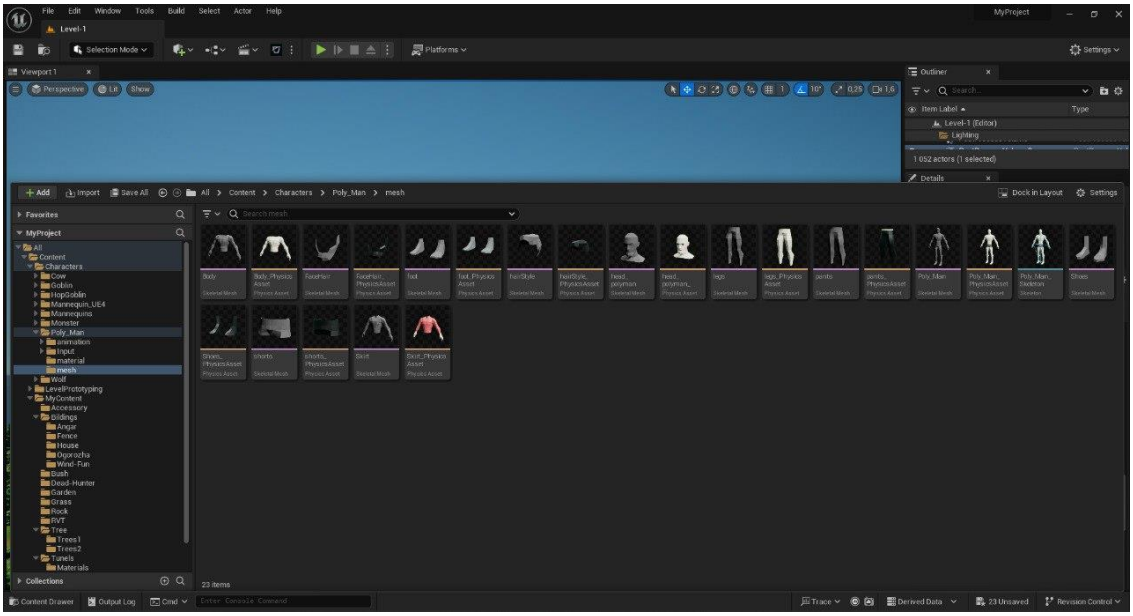


рис.4.2.4

-Dead Hunter (рис.4.2.5)

рис.4.2.5

-Wolf (рис.4.2.6)

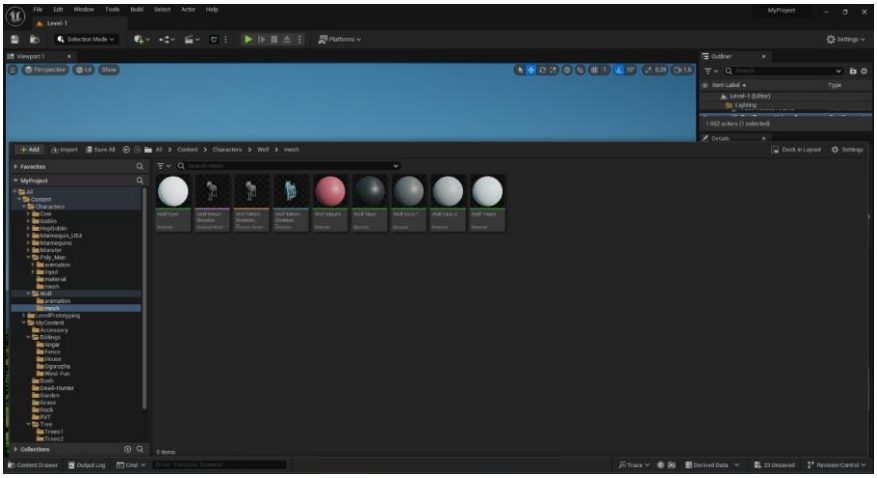


рис.4.2.6

-Goblin (рис.4.2.7)

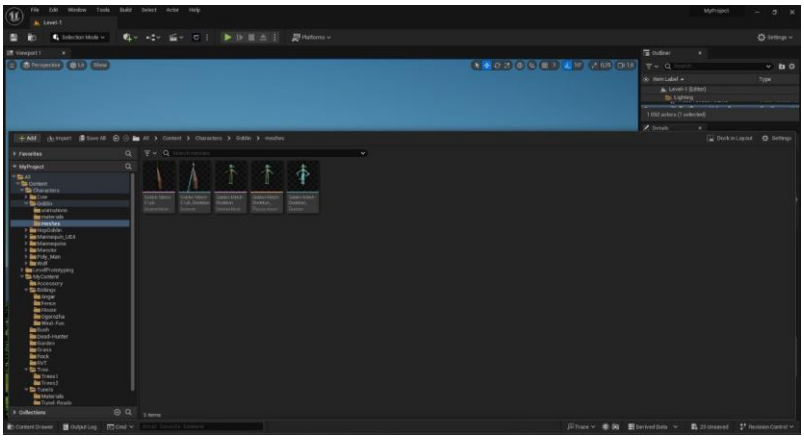


рис.4.2.7

-Hop Goblin (рис.4.2.8)

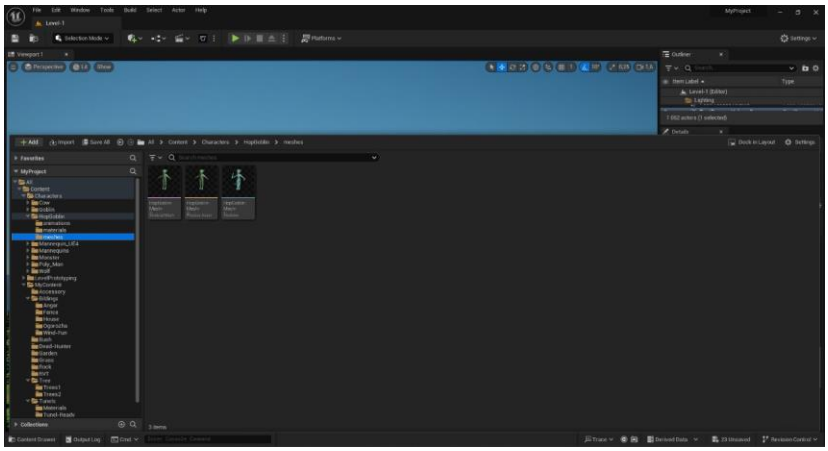


рис.4.2.8

-Cow (рис.4.2.9)

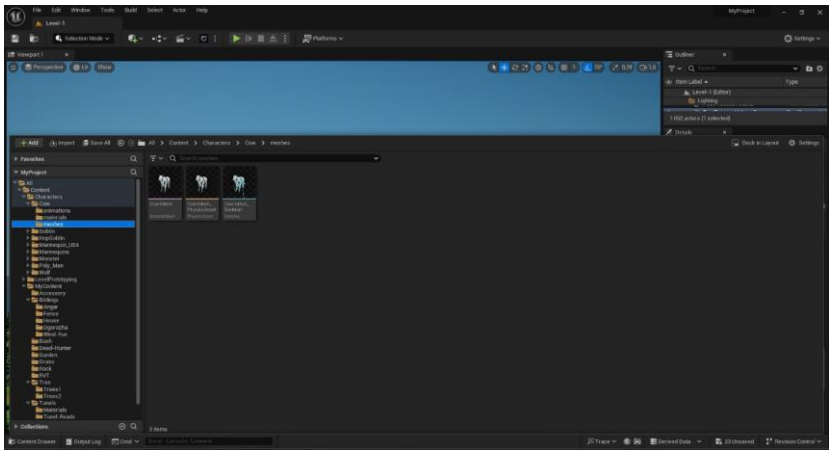


рис.4.2.9

2.Farm:

-Angar (рис.4.2.10)

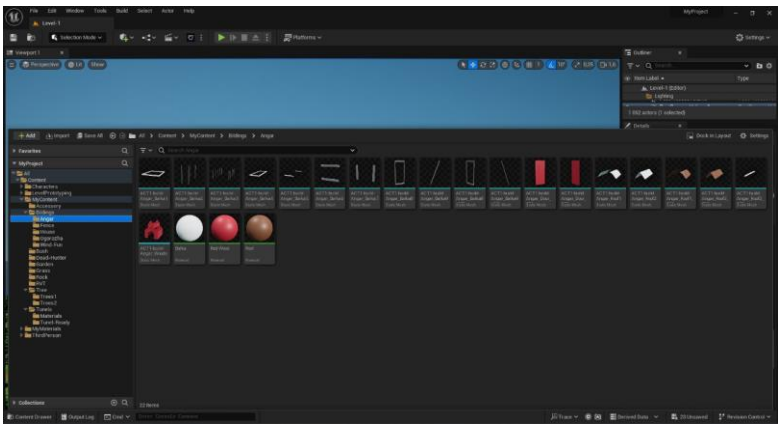


рис.4.2.10

-House (рис.4.2.11)

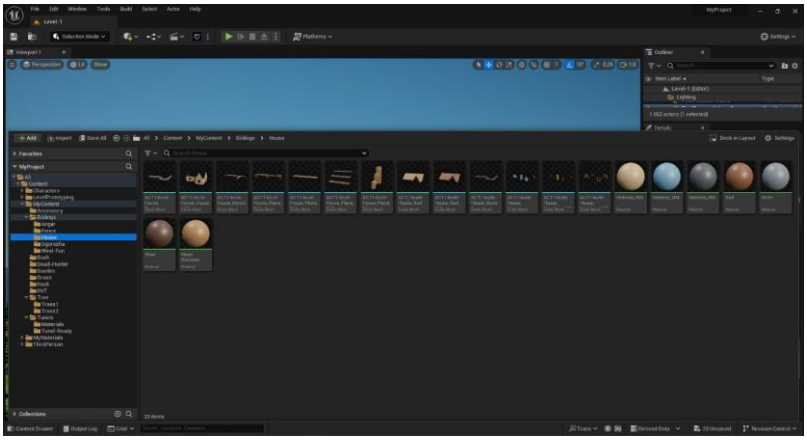


рис. 4.2.11

-Fence (рис.4.2.12)

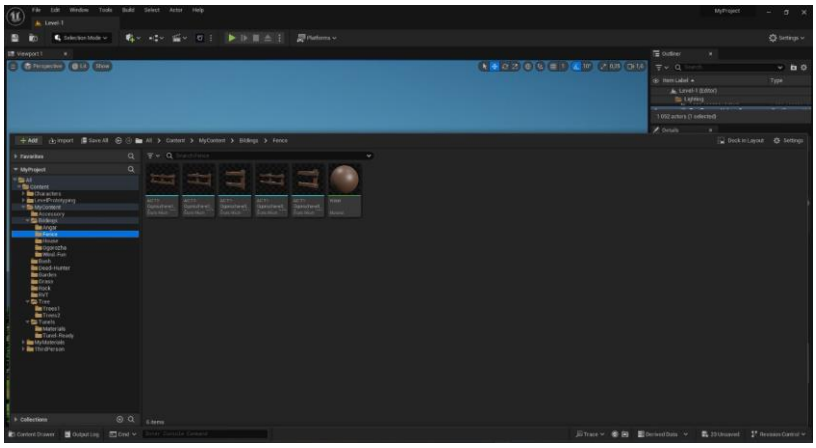


рис. 4.2.12

-Wind-Fun (рис.4.2.13)

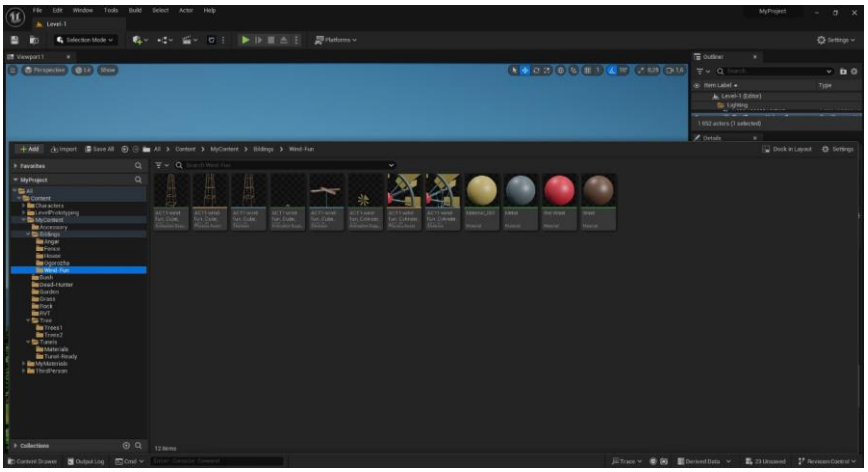


рис.4.2.13

-Garden (рис.4.2.14)

рис.4.2.14

-Accessory (рис.4.2.15)

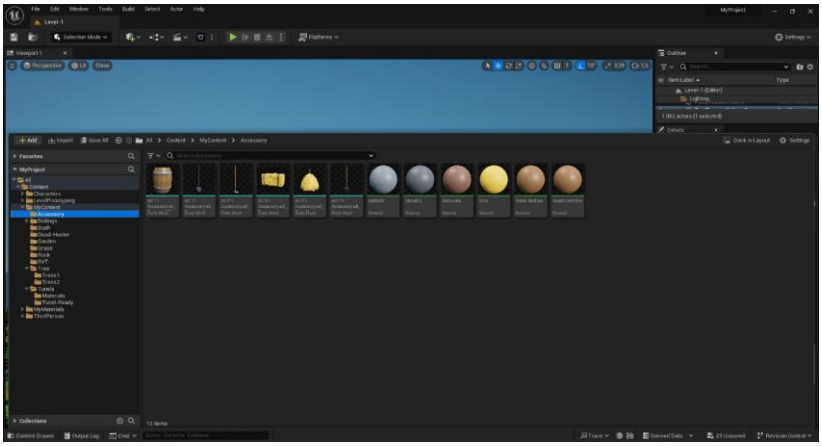


рис.4.2.15

3.Plants:

-Grass (рис.4.2.16)

рис.4.2.16

-Trees (рис.4.2.17 та рис.4.2.18)

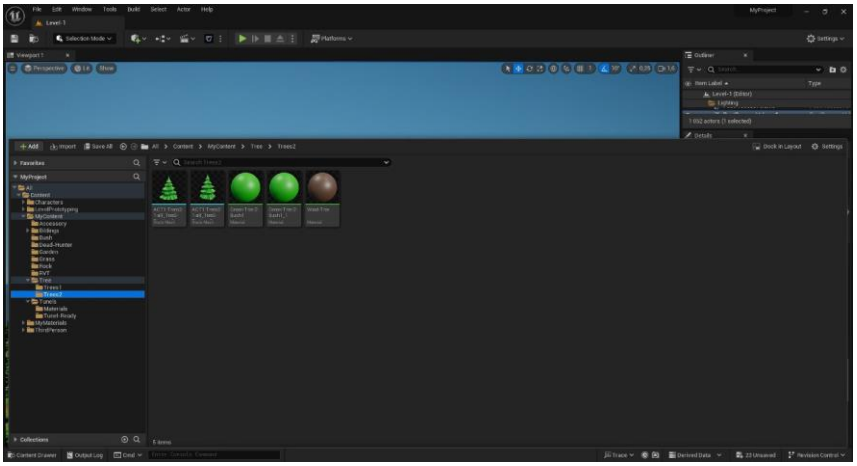


рис.4.2.17

рис.4.2.18

-Bushes (рис.4.2.19)

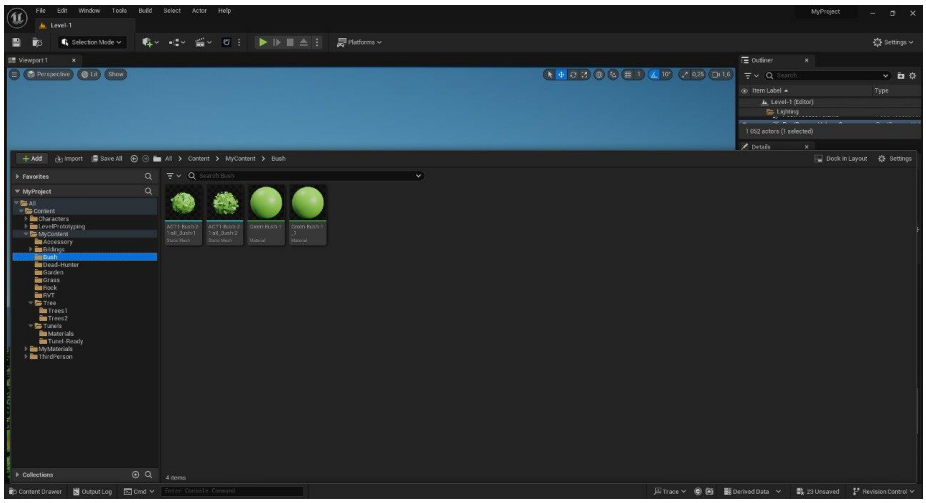


рис.4.2.19

4.Rocks (рис.4.2.20)

рис.4.2.20

5.Tunels (рис.4.2.21)

рис.4.2.21

6. Bow (рис.4.2.22)

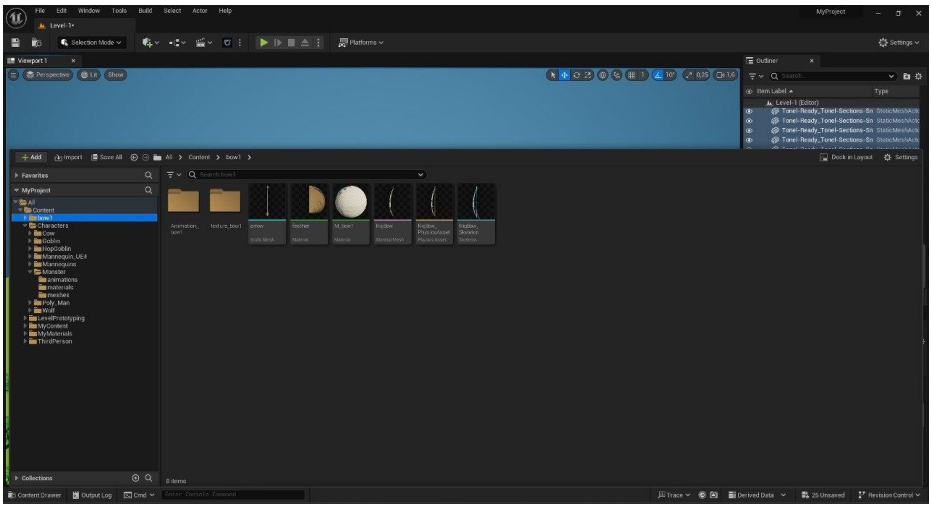


рис.4.2.22

Об'єкти на рівні можна переміщувати, обертати та масштабувати. Комбінації клавіш для них – W(переміщення), E(поворот) та R(маштабування). переміщувнна, обертання та маштабування відбувається по глобальним координатам x, y, z.

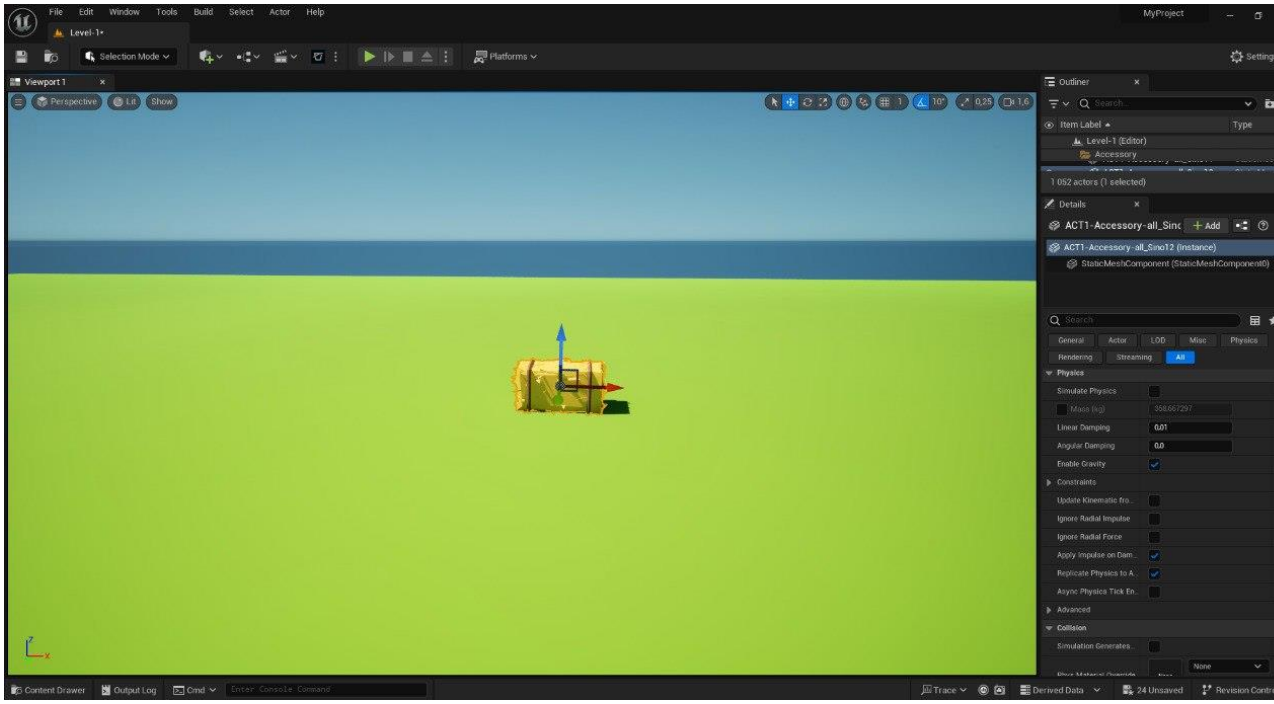


рис.4.2.23 – Переміщення по глобальним координатам x, y, z.

Таким чином було створено ігрове середовище

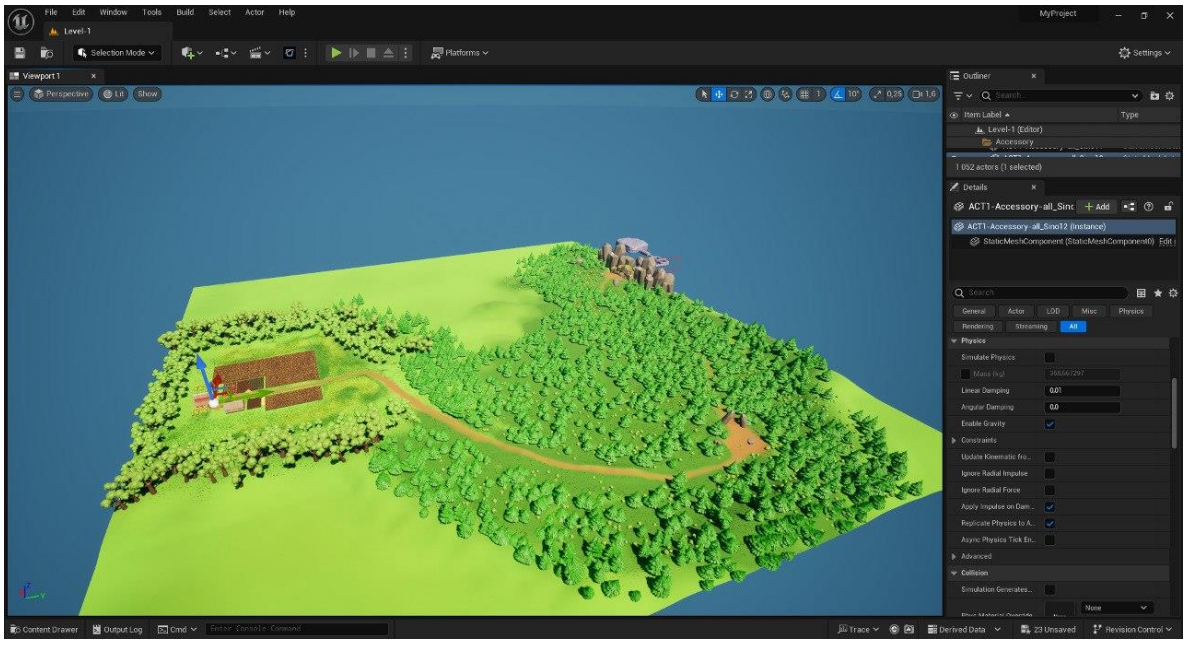


рис.4.2.24 - Ігрове середовище

Там де було потрібно розташувати багато однакових моделей до прикладу: дерева, кущі, трава, був використатний інструмент “Folliage”. Див рис. 4.2.25 та рис. 4.2.26

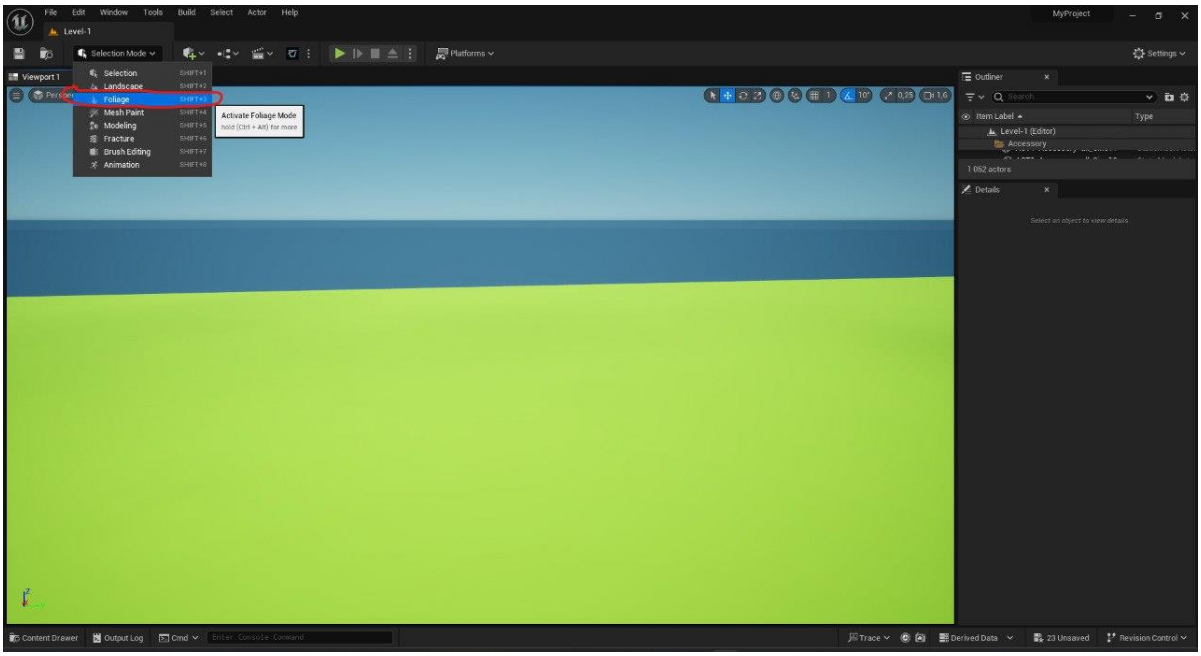


рис. 4.2.25

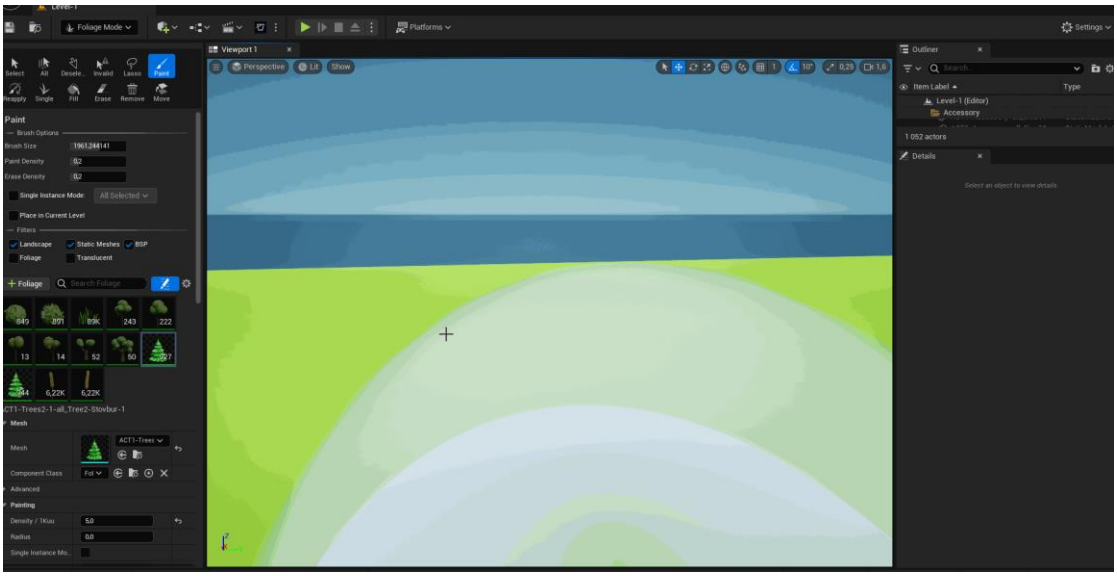


рис. 4.2.26

4.3 Скелет та анімація для 3D моделей

Для імпорту анімації виконуються такі самі степи як і при імпорті 3D моделей а саме: перейти до браузера натиснувши кнопку “Content Drawer” змісту далі створити папку та натиснути кнопку “Імпортувати”. Обрати файл анімації у форматі .fbx

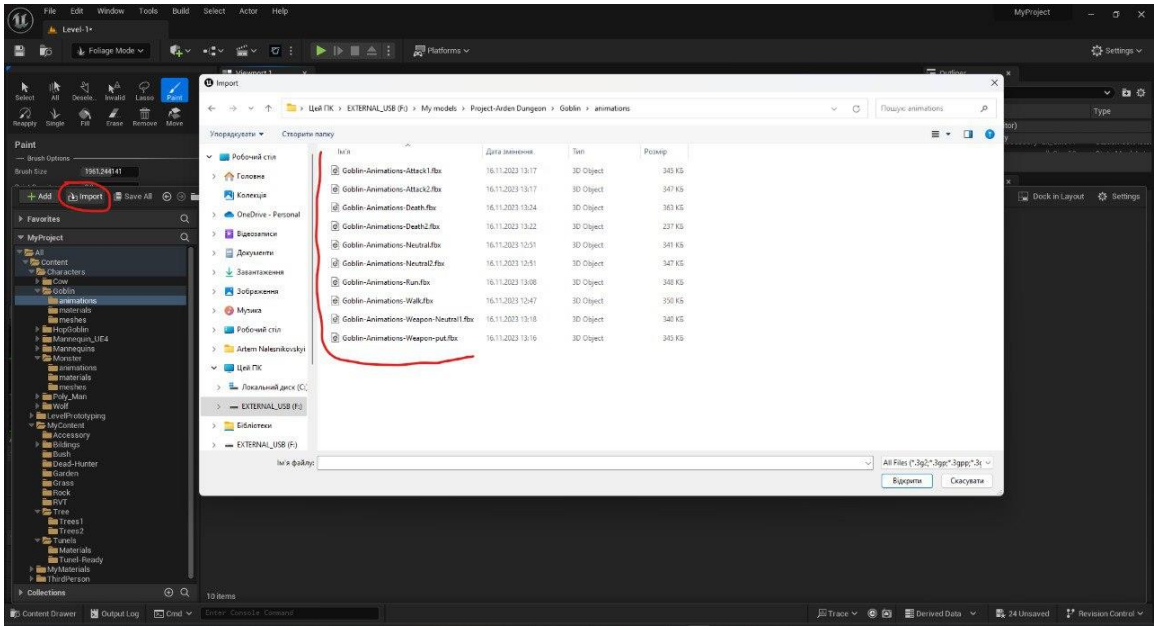


Рис. 4.3.1 – Імпорт анімації

Для імпорту анімації треба вимкнути чекбокс “Import Mesh” та обрати потрібний скелет який був імпортований разом з мешем раніше. Див. рис. 4.3.2

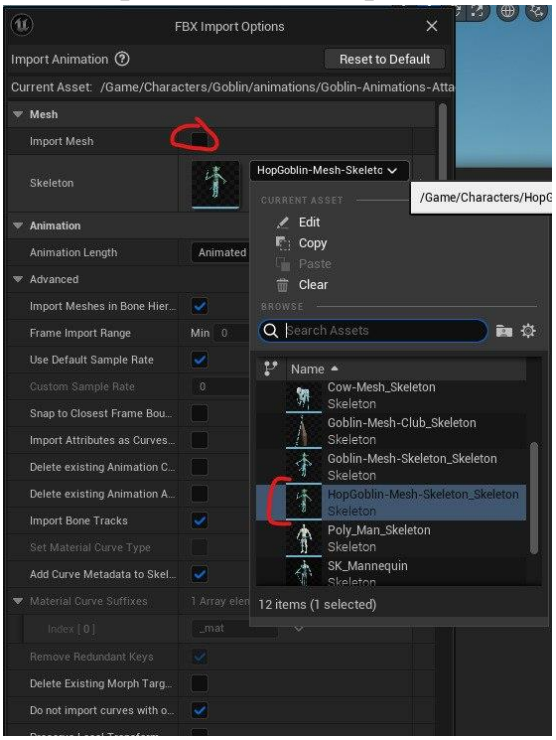


Рис. 4.3.2

Далі треба для поля “Animation Lenght” обрати Animated time. Це потрібно для того щоб захопити всі кадри анімації які були створенні для неї в програмі 3D моделюванні. Потім натиснути кнопку “Імпорт”. Див. рис. 4.3.3

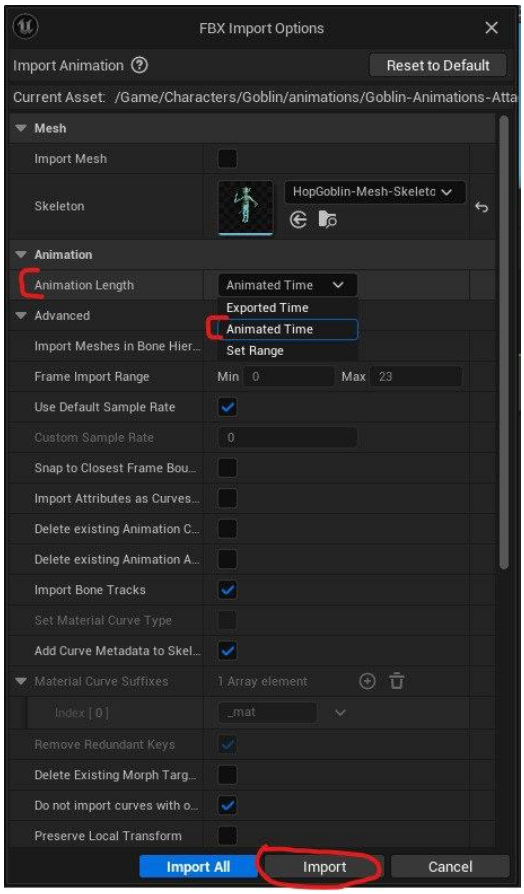


Рис. 4.3.3

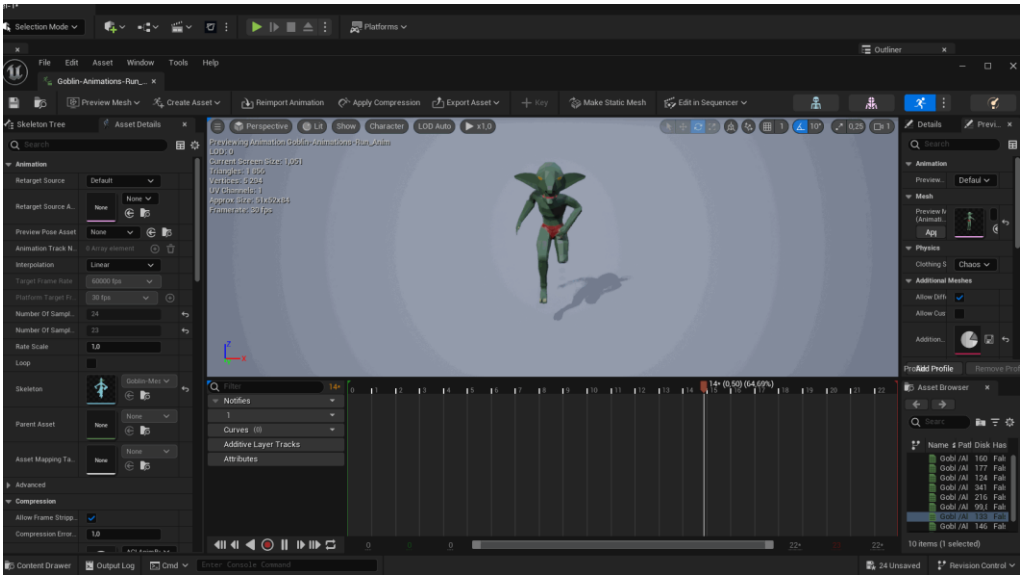


Рис. 4.3.4 – Імпортована анімація

Всі анімації крім Poly_Man були створені в програмі Blender та підготовлені заздалегіть:

1.Wolf:

- Атака
- Смерть
- Споживання їжі1 та Споживання їжі2
- Біг

2. Goblin

- Атака та Атака2
- Смерть1 та Смерть2
- Нейтральна позиція1 та Нейтральна позиція2
- Біг
- Ходьба
- Нейтральні позиція зі зброєю
- Діставання зброї

3. Hop Goblin

- Атака та Атака2
- Смерть1 та Смерть2
- Нейтральна позиція1 та Нейтральна позиція2
- Біг
- Ходьба
- Нейтральні позиція зі зброєю
- Діставання зброї
- Біг з коровою

4. Cow

- Ходьба

- Біг

Для головного персонажу Poly_Man був використаний скелет з Unreal Engine та інтегрований:

- На сайті Mixamo було вставлено стандартний манікен з Unreal Engine разом з його скелетом і на його основі були завантажені анімації
- За допомогою программи Mixamo Converter завантажені анімації були підготовлені до імпорту в Unreal Engine
- В Unreal Engine імпортовані анімації працюють на стандартному манекені а так як наш персонаж був зроблений на основі того самого скелету, за допомогою IK Retargeter були перенесені анімаціїї на персонажа та створено анімаційний блюпринт. Див. рис. 4.3.5

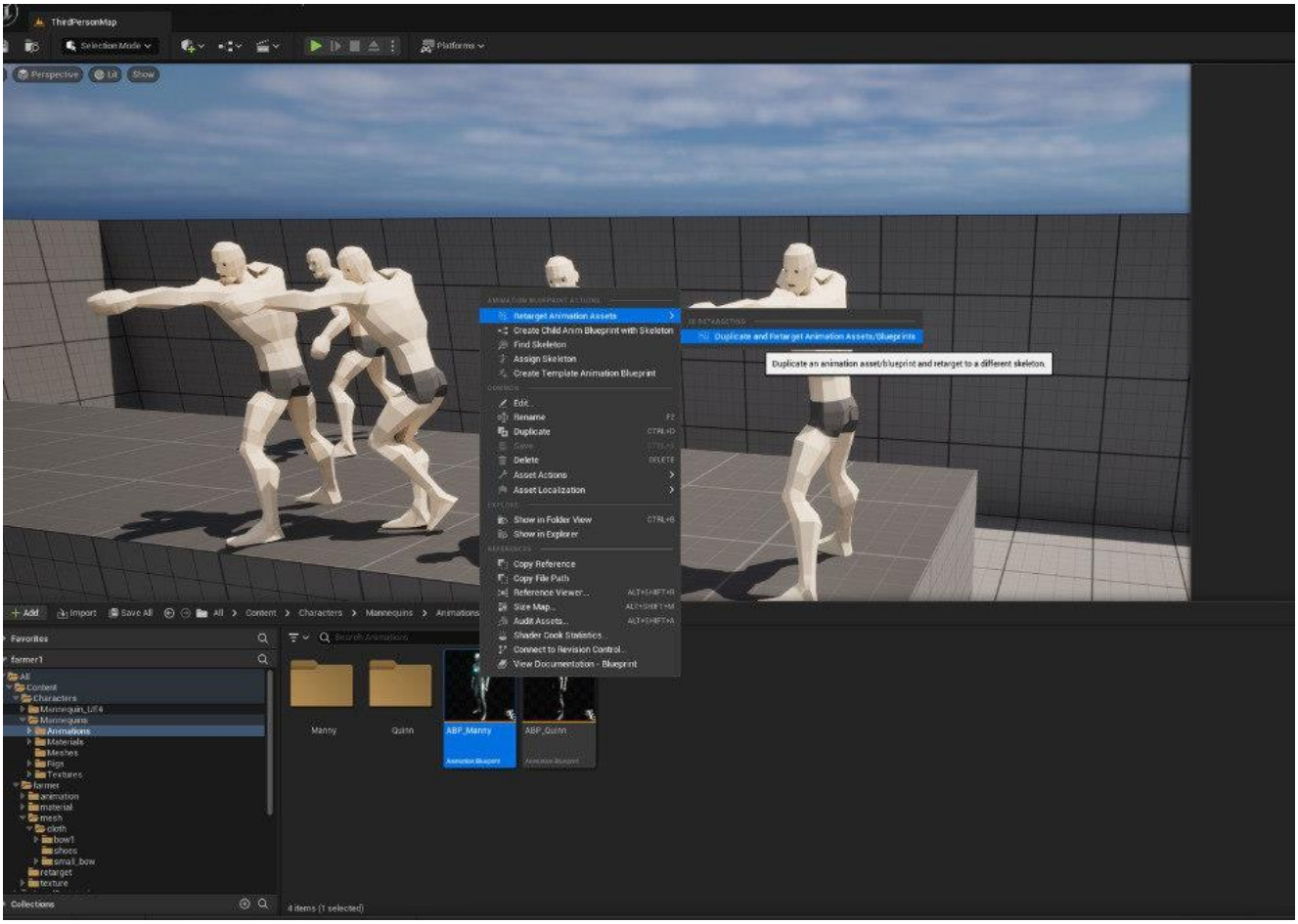


Рис. 4.3.5

- Блюпринт класс було взято з стандартного контенту гри від Third Persone, скопійовано і в ньому замінено анімаційний блюпринт на свій а також модель

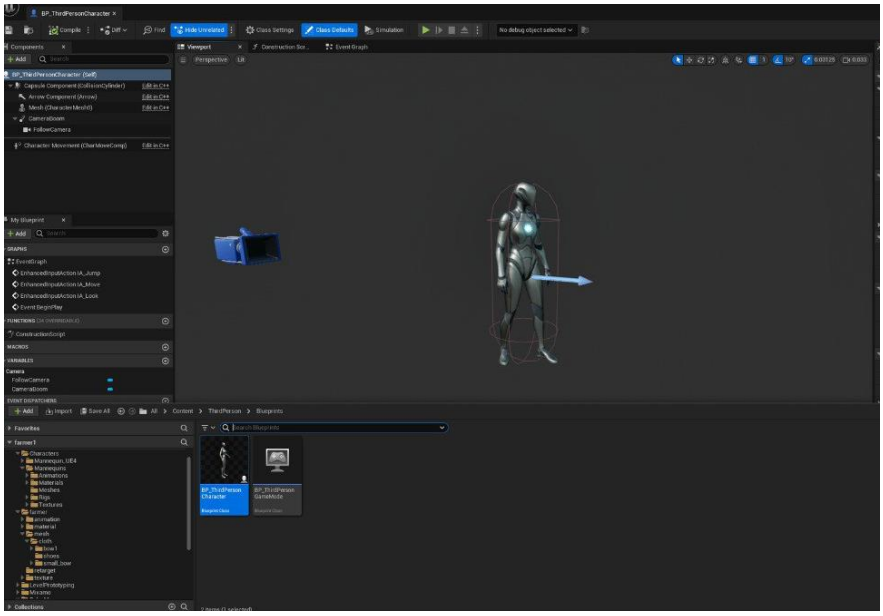


Рис. 4.3.6 - Стандартний контент гри від Third Persone

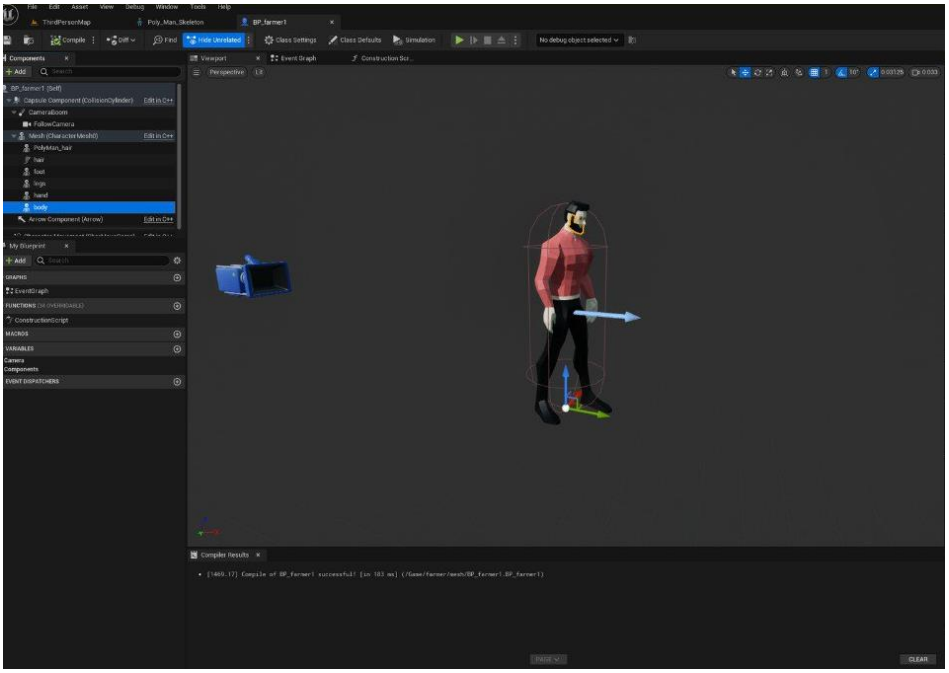


Рис. 4.3.7 - Головний персонаж Poly_Man з інтегрованим скелетом з Unreal Engine

Список анімацій для Poly_Man:

- Рух вперед
- Рух вправо
- Рух Вліво

- Стрільба
- Стрибок

4.4 Міксування коду і Blueprints в Unreal Engine

4.4.1 Налаштування проекту

Наш персонаж успадковується від класу Character, тому наш C++ клас теж повинен успадковувати цей клас. Назвемо його BaseCharacter, так як він буде основним класом для всіх персонажів гравця. Див. рис. 4.4.1

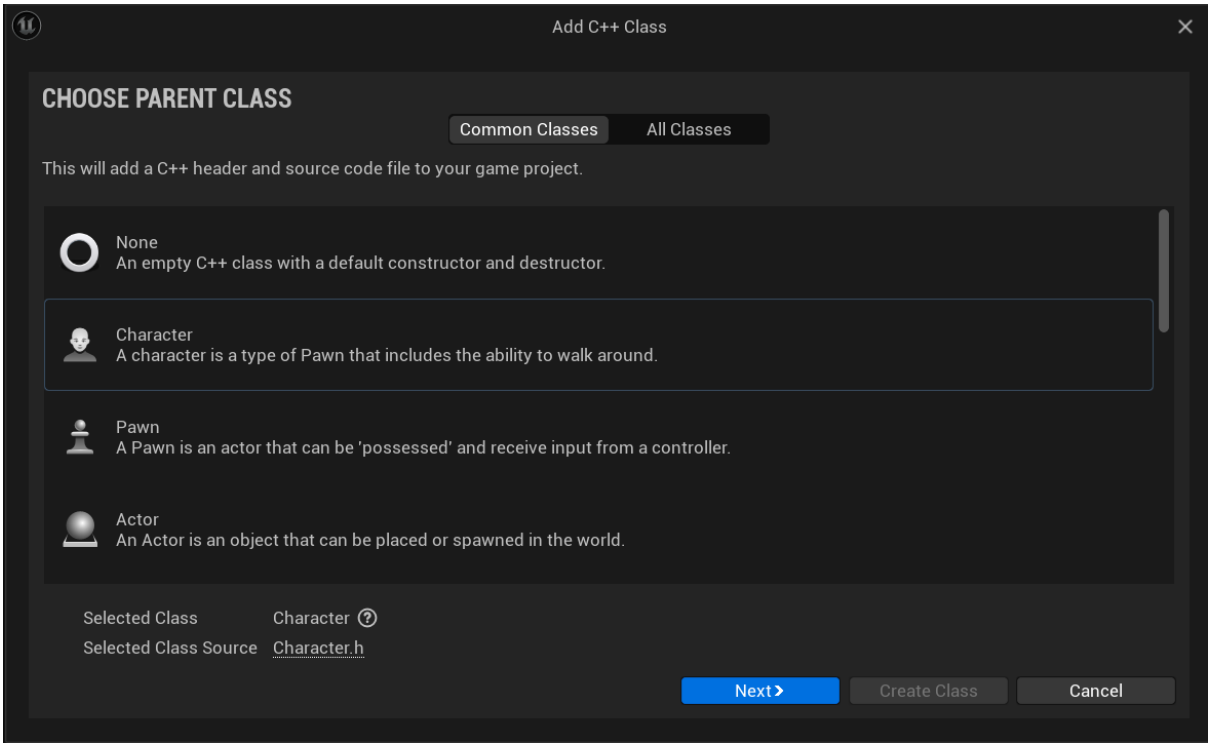


рис. 4.4.1

Далі треба відкрити скелет нашого персонажу, знайти в списку кістку правої руки і створити сокет за назваю “arrow_socket”. З цього місця буде створюватися стріла коли ми будемо стріляти з лука. Див. рис. 4.4.2

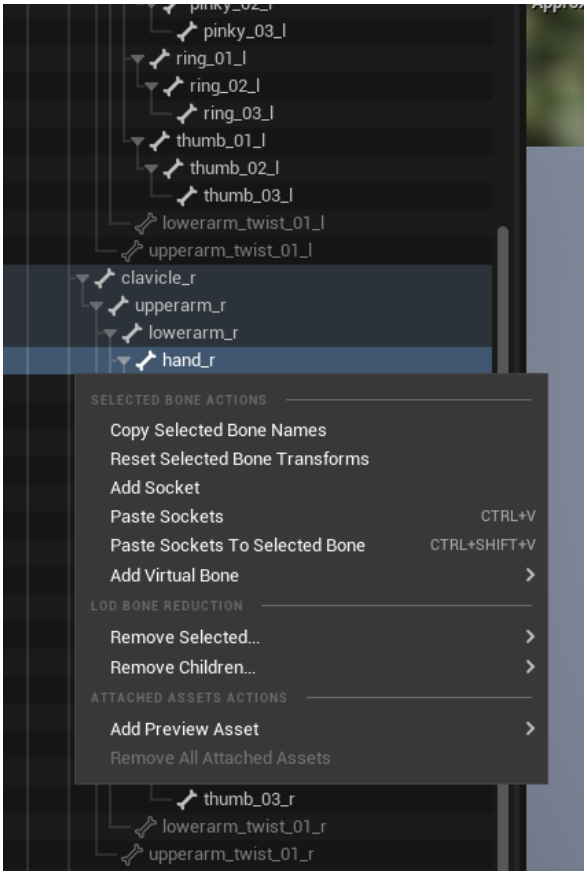


рис. 4.4.2

Створимо пустий блупрінт з назвою BP_Arrow і додамо туди статичний меш стріли, і компонент ProjectileMovement.(Рис. 4.4.3)

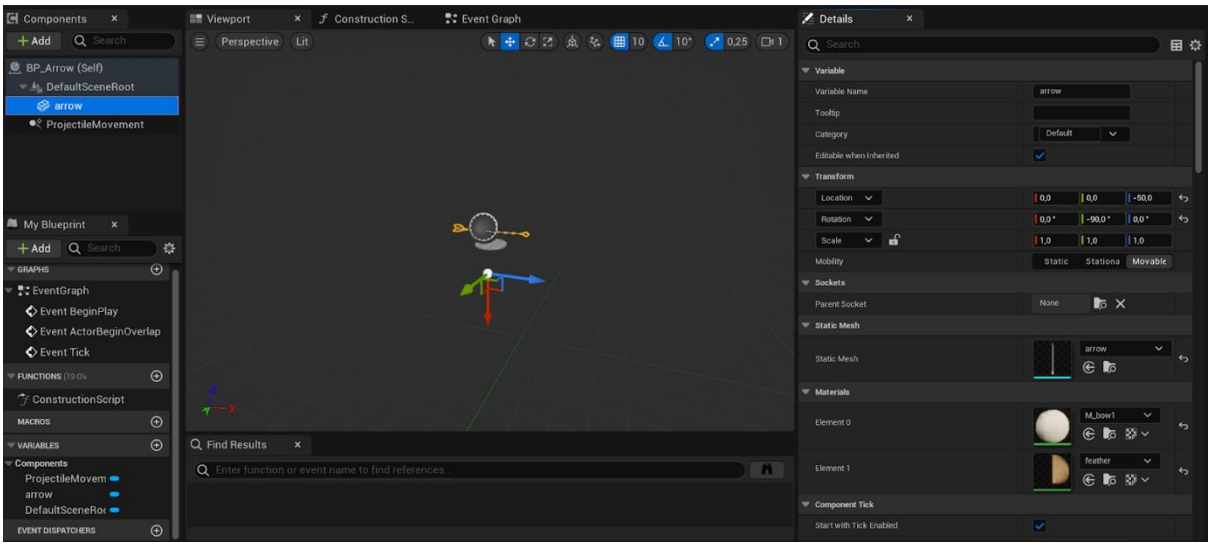


Рис. 4.4.3

4.4.2 Створення основних змінних і компонентів

Спочатку в файлі BaseCharacter.h оголосимо основні змінні які нам потрібні для реалізації конструктора.

```
USpringArmComponent* CameraBoom;
UCameraComponent* FollowCamera;
UInputMappingContext* DefaultMappingContext;
```

CameraBoom - типу SpringArmComponent, це буде посилання на базовий компонент в Unreal Engine 5, який намагається підтримувати своїх дочірніх компонентів на фіксованій відстані від батьківського елемента

FollowCamera - типу CameraComponent, посилання на камеру.

DefaultMappingContext - типу InputMappingContext, відображає введені користувачами дії вхідних даних

Далі створюємо змінні для дій користувача (InputAction): дії відповідальні за стрибок, переміщення, перегляд, біг, натискання правої і лівої клавіши миші.

```
UInputAction* JumpAction;
UInputAction* MoveAction;
UInputAction* LookAction;
UInputAction* RunAction;
UInputAction* RMBAction;
UInputAction* LMBAction;
```

Треба створити InputMappingContext і всі описані вище InputAction

В content browser, натиснути +Add - Input (Рис. 4.4.4)

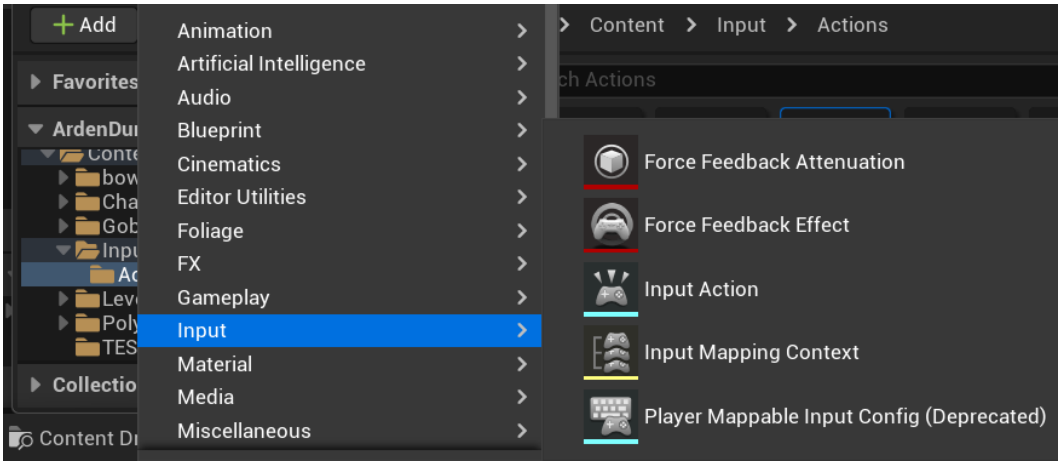


Рис. 4.4.4

InputMappingContext назвемо IMC_Default, і налаштуємо клавіши натискання які будуть визивати функціїї дій (Рис. 4.4.5)

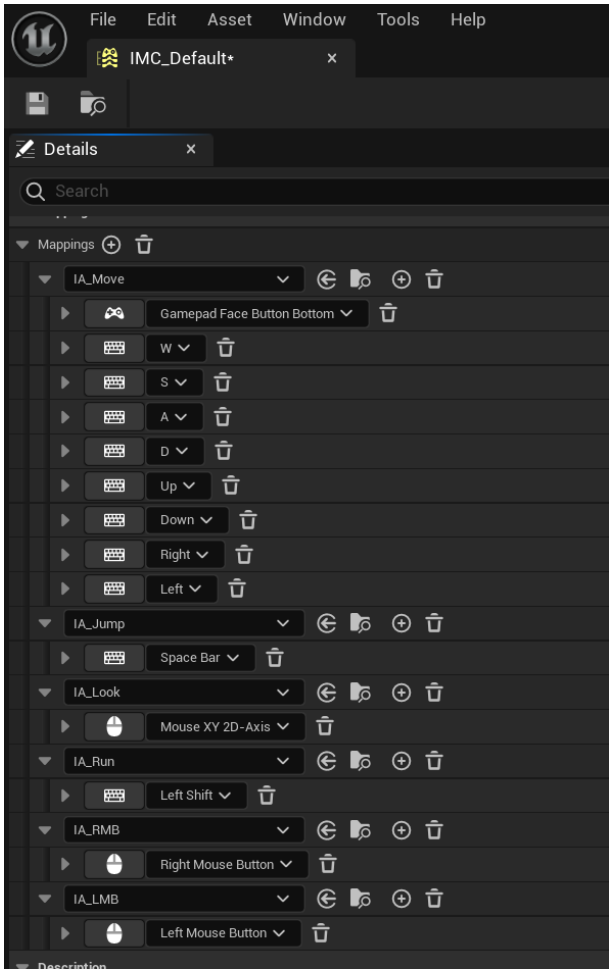


Рис. 4.4.5

Для того щоб можна було налаштовувати InputAction і MappingContext в рушії, треба додати перед кожною змінною макрос UPROPERTY()

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input)
```

EditAnywhere - саме дозволяє нам змінювати змінні у рушії в кладці Class Default
BlueprintReadOnly - дозволяє визивати значення у блупринті але не змінювати його, це нам буде необхідно для виклику подій при роботі з інтерфейсом користувача.
Category - об'єднує змінні в одну категорію, як показано на Рис.4.4.6.

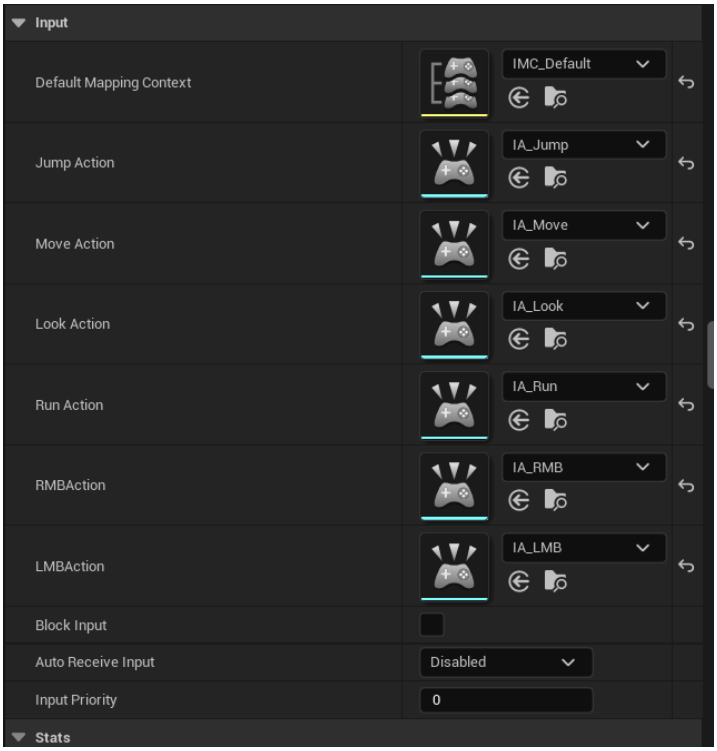


Рис.4.4.6

4.4.3 Реалізація конструктора класу

В конструкторі класу нам треба реалізувати створення компоненту FollowCamera і CameraBoom, і приєднання FollowCamera до CameraBoom як батьківського об’єкту

```
CameraBoom =
CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));
CameraBoom->SetupAttachment(RootComponent);
CameraBoom->TargetArmLength = 400.0f;
CameraBoom->bUsePawnControlRotation = true;


FollowCamera =
CreateDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"));
FollowCamera->SetupAttachment(CameraBoom,
USpringArmComponent::SocketName);
FollowCamera->bUsePawnControlRotation = true;
```

4.4.4 Підключення і реалізація системи керування (input system)

В функції BeginPlay нам треба додати наш DefaultMappingContext, в підсистему керування вхідними діями, щоб рушій замінив MappingContext за замовчуванням, на наш

```
Subsystem->AddMappingContext(DefaultMappingContext, 0);
```

Додамо функції для вхідних дій в файлі BaseCharacter.h

```
void Move(const FInputActionValue& Value);
void Look(const FInputActionValue& Value);
void RunStart();
void RunCompleted();
void AimingBowEnable();
void AimingBowDisable();
void ShootBow();
```

та в функції SetupPlayerInputComponent, наслідувальній від батьківського класа Character (в фалі BaseCharacter.cpp) додамо до вхідних дій ці функції

```
EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Started, this, &ACharacter::Jump);
EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Completed, this, &ACharacter::StopJumping);

EnhancedInputComponent->BindAction(MoveAction, ETriggerEvent::Triggered, this, &ABase_Character::Move);
```

```
EnhancedInputComponent->BindAction(LookAction, ETriggerEvent::Triggered,
this, &ABase_Character::Look);

EnhancedInputComponent->BindAction(RunAction, ETriggerEvent::Started,
this, &ABase_Character::RunStart);
EnhancedInputComponent->BindAction(RunAction, ETriggerEvent::Completed,
this, &ABase_Character::RunCompleted);

EnhancedInputComponent->BindAction(RMBAction, ETriggerEvent::Triggered,
this, &ABase_Character::AimingBowEnable);
EnhancedInputComponent->BindAction(RMBAction,
ETriggerEvent::Completed, this, &ABase_Character::AimingBowDisable);

EnhancedInputComponent->BindAction(LMBAction, ETriggerEvent::Triggered,
this, &ABase_Character::ShootBow);
```

4.4.5 Оголошення змінних, функцій і структур для реалізації ігрової логіки

Спочатку реалізуємо структуру, назвемо її LvlStatsStruc, і оголосимо в ній декілька змінних: кількість необхідного досвіду до наступного рівня, максимальне значення здоров’я на цьому рівні, і бази пошкодження які наносить персонаж. Значення за замовчуванням ми будемо вказувати для всіх персонажів окремо, тому потрібно в макросі UPROPERTY додати параметр EditAnywhere, для редагування з рушія, і BlueprintReadWrite для використання значень в блупрінті де ми будемо відображати бар здоров'я, досвіду, і рівень.

```
UPROPERTY(EditAnywhere, BlueprintReadWrite)
float ExpToNextLvl;
UPROPERTY(EditAnywhere, BlueprintReadWrite)
float MaxHP;
```

```
UPROPERTY(EditAnywhere, BlueprintReadWrite)
float BaseDamage;
```

Далі в нашому BaseCharacter.h оголосимо змінні:
Змінна Arrow, буде зберігати посилання на блупрінт стріли, в якому буде сам статичний меш, і базовий компонент ProjectileMovement, відповідальний за переміщення об'єкта по заданим параметрам, її ми будемо вказувати у рушії, тому нам потрібен параметр EditAnywhere, і додамо її в категорію до інших компонетів

```
UPROPERTY(EditAnywhere, Category = Components)
UClass* Arrow;
```

Логічна змінна Aiming - для синхронізації анімації з діями персонажа, тому нам потрібен параметр BlueprintReadWrite, далі об'єднаємо всі характеристики в категорію Stats

```
UPROPERTY(BlueprintReadWrite, Category = Stats)
bool Aiming;
```

Логічна змінна Recoil - також потрібна здебільшого для анімації, перезарядки лука

```
UPROPERTY(BlueprintReadWrite, Category = Stats)
bool Recoil;
```

Массив LvlStats - буде зберігати значення які змінюються на різних рівнях персонажа, він має тип LvlStatsStruc що являється структурою, яку ми створили раніше, і буде налаштовуватися в рушії.

```
UPROPERTY(EditAnywhere, Category = Stats)
FLvlStatsStruc LvlStats[3];
```

CurrentHealth і CurrentExp це лічильники поточного значення здоров'я і досвіду персонажа , це будуть локальні змінні, тому їм не потрібен макрос UPROPERTY

```
float CurrentHealth;
float CurrentExp;
```

CurrentLvl - лічильник поточного рівня, ми будемо його відображати через блупрінт у інтерфейсі користувача, тому потрібен параметр BlueprintReadWrite

```
UPROPERTY(BlueprintReadWrite, Category = Stats)
int CurrentLvl;
```

Тепер необхідно оголосити функції які ми будемо використовувати
ShootBow - в цій функції ми будемо створювати нашу стрілу, і робити прорахунки влучення в ціль

```
void ShootBow();
```

LvlUp - підвищення рівня, і синхронізація нових характеристик

```
void LvlUp();
```

PrecentHp, PrecentExp - повертатимуть значення в процентах для бару здоров'я і досвіду, ці функції будемо визивати в блупрінті тому потрібен макрос UFUNCTION, з параметром BlueprintCallable

```
UFUNCTION(BlueprintCallable)
float PrecentHp();

UFUNCTION(BlueprintCallable)
float PrecentExp();
```

TakeExp - нарахування отриманого досвіду до поточного

```
UFUNCTION(BlueprintCallable)
void TakeExp(float exp);
```

TakeDamage це віртуальна успадкована від Character функція отримання пошкодження, нам треба її перевизначити для нашої ігрової логіки.

```
virtual float TakeDamage (float DamageAmount,struct FDamageEvent const& DamageEvent,class AController* EventInstigator, AActor* DamageCauser) override;
```

4.4.6 Реалізація ігрової логіки

Реалізуємо всі оголошенні нами раніше функції:

LvlUp - нам треба поточний рівень збільшити на один, обнулити лічильник поточного досвіду, і оновити дані здоровья. При підвищенні рівня, буде автолікування, до максимального значення здоровья.

```
void ABase_Character::LvlUp()
{
CurrentLvl++;
CurrentExp = 0;
```

```
 CurrentHealth = LvlStats[CurrentLvl].MaxHP;
}
```

PrecentHp і PrecentExp повертають відсоток залишку здоров'я від максимального, і відсоток до наступного рівня відповідно

```
float ABase_Character::PrecentHp()
{
 return CurrentHealth / LvlStats[CurrentLvl].MaxHP;
}

float ABase_Character::PrecentExp()
{
 return CurrentExp / LvlStats[CurrentLvl].ExpToNextLvl;
}
```

TakeExp - отримує значення досвіду, додає до поточного, і підвищує рівень в разі якщо поточний досвід перевищує необхідний для підвищення рівня

```
void ABase_Character::TakeExp(float exp)
{
 CurrentExp += exp;
 if (CurrentExp >= LvlStats[CurrentLvl].ExpToNextLvl) LvlUp();
}
```

TakeDamage - отримаємо значення нанесених нам пошкоджень, і віднімаємо його від поточного здоровья

```
float ABase_Character::TakeDamage(float DamageAmount, struct FDamageEvent const& DamageEvent, class AController* EventInstigator, AActor* DamageCauser)
{
float dmg = Super::TakeDamage(DamageAmount, DamageEvent,EventInstigator,DamageCauser);
CurrentHealth -= dmg;
return dmg;
}
```

Move - рахуємо в якому напрямку повинен переміститися наш персонаж відповідно натиснутої клавіши гравцем, і робимо переміщення.

```
void ABase_Character::Move(const FInputActionValue& Value)
{
FVector2D MovementVector = Value.Get<FVector2D>();

if (Controller != nullptr)
{
    const FRotator Rotation = Controller->GetControlRotation();
    const FRotator YawRotation(0, Rotation.Yaw, 0);

    const FVector ForwardDirection = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);

    const FVector RightDirection = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);
```

```
        AddMovementInput(ForwardDirection, MovementVector.Y);
        AddMovementInput(RightDirection, MovementVector.X);
    }
    }
```

Look- отримуємо направлення переміщення миші, і повертаємо персонажа у той бік.

```
    void ABase_Character::Look(const FInputActionValue& Value)
    {
     FVector2D LookAxisVector = Value.Get<FVector2D>();

     if (Controller != nullptr)
     {
        AddControllerYawInput(LookAxisVector.X);
        AddControllerPitchInput(LookAxisVector.Y);
     }
    }
```

RunStart - збільшуємо швидкість переміщення персонажа коли клавіши бігу утримуються

```
    void ABase_Character::RunStart()
    {
     if (!Aiming)
     {
        GetCharacterMovement()->MaxWalkSpeed = 500.f;
```

```
}
}
```

RunCompleted - повертаємо швидкість до базового значення коли кнопку бігу відпустили

```
void ABase_Character::RunCompleted()
{
 GetCharacterMovement()->MaxWalkSpeed = 300.f;
}
```

AimingBowEnable - активація прицілювання при утриманні клавіши прицілювання, і приближення камери до персонажу

```
void ABase_Character::AimingBowEnable()
{
 Aiming = true;
 CameraBoom->TargetArmLength = 200.f;
}
```

AimingBowDisable - деактивація прицілювання і повернення камери на базове значення

```
void ABase_Character::AimingBowDisable()
{
 Aiming = false;
 CameraBoom->TargetArmLength = 400.f;
}
```

ShootBow - якщо персонаж находится в режимі прицілювання, і зараз не перезарядка, створюємо стрілу, в координатах раніше створеного сокета “arrow_socket”, повертаємо стрілу в направлення погляду камери.

В цей момент стріла починає свій рух, ми змінюємо статус Recoil, і в цьому ж саме направленню, випускаємо луч зіткнення, який повертає перший об'єкт типу Actor з яким перетинається.

Так як в нашій грі об'єкти типу Actor це тільки персонажі гравця і супротивники, ми додаємо параметр який ігнорує персонажа на перевірку зіткнення. Таким чином луч зіткнення може повернути тільки Actor супротивників, і якщо луч повертає успішне зіткнення, ми запускаємо функцію TakeDamage, аналогічну той яку ми перевизначели в нашому персонажі.

```
void ABase_Character::ShootBow()
{

 if (Aiming && !Recoil)
 {
        FActorSpawnParameters SpawnInfo;
        SpawnInfo.Instigator = this;
        SpawnInfo.Owner = this;
        const FVector transfom_socket = GetMesh()->GetSocketLocation("arrow_socket");
        const FRotator rotator_camera = FollowCamera->GetComponentRotation();
        const FVector scale = FVector(1, 1, 1);
        FTransform transform = FTransform(rotator_camera, transfom_socket, scale);
```

```
        AActor* new_arrow = GetWorld()->SpawnActor(Arrow, &transform,
SpawnInfo);

        Recoil = true;

        FVector Start = GetMesh()->GetSocketLocation("arrow_socket");
        FVector End = Start + FollowCamera->GetForwardVector() * 2000.f;
        FHitResult HitResult;
        FCollisionQueryParams Params;
        Params.AddIgnoredActor(this);

        if (GetWorld()->LineTraceSingleByChannel(HitResult, Start, End,
ECollisionChannel::ECC_Camera, Params, FCollisionResponseParams()))
        {
                HitResult.GetActor()-
>TakeDamage(LvlStats[CurrentLvl].BaseDamage, FDamageEvent(),
GetInstigatorController(), this);
        }
}
}
```

4.4.7 Налаштування успадкування базового блупрінт класса від c++ класса

Відкриємо наш BP Third Person в рушії, переходемо у вкладку Class Settings, і в графі Parent Class замінемо стандартний клас на створений нами. Тепер ми можемо використовувати написані нами функції і змінні. Див. Рис.4.4.7

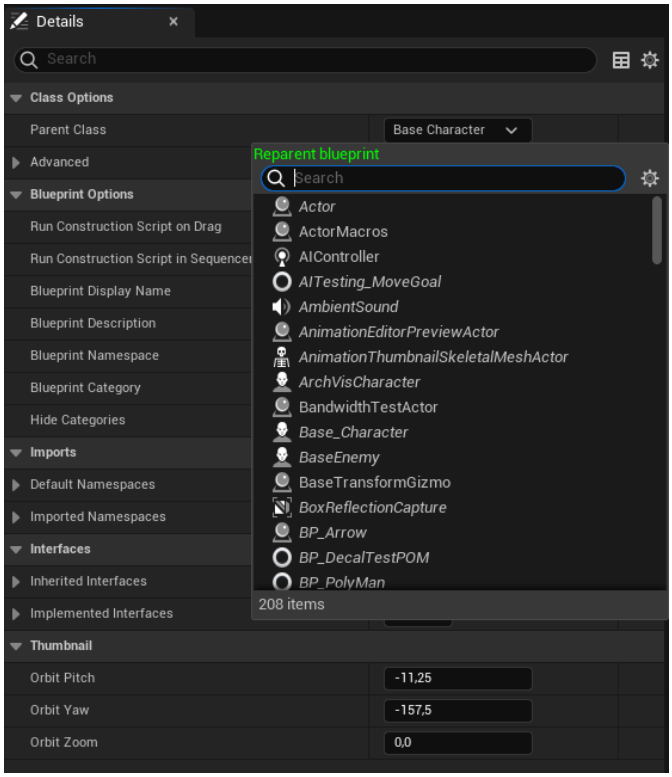


Рис. 4.4.7

У вкладці Class Defaults знайдемо нашу категорію Stats, і заповнемо характеристики відповідно до Рис. 4.4.8. Також вказуємо наш BP_Arrow в наш компонент Arrow.

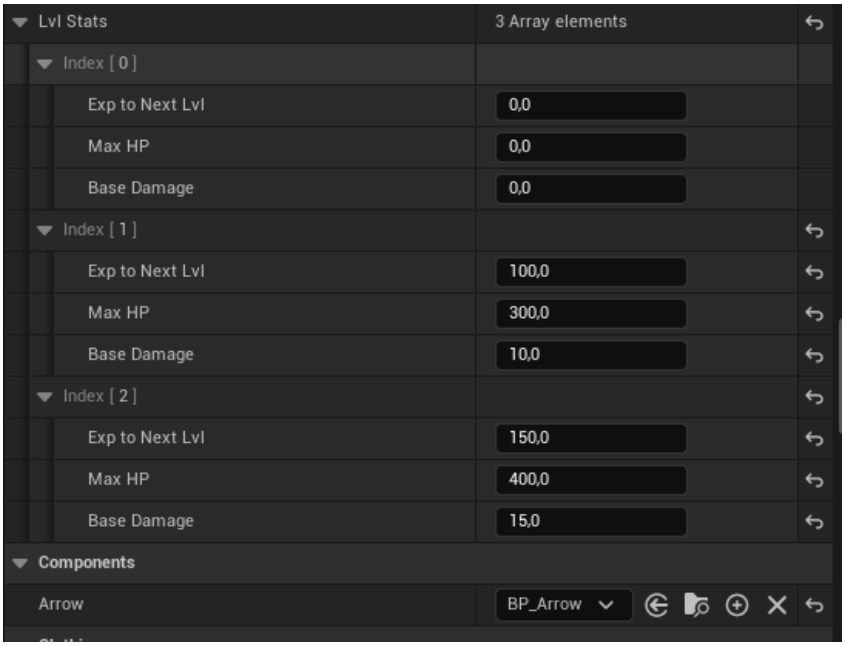


Рис. 4.4.8

4.4.8 Створення інтерфейсів користувача

Створимо два Widget Blueprint і додамо їх як компонент в BP Third Person:

W_ExpBar - з текстовим блоком lvl, і progress bar з назвою exp ba. (Рис.4.4.9)

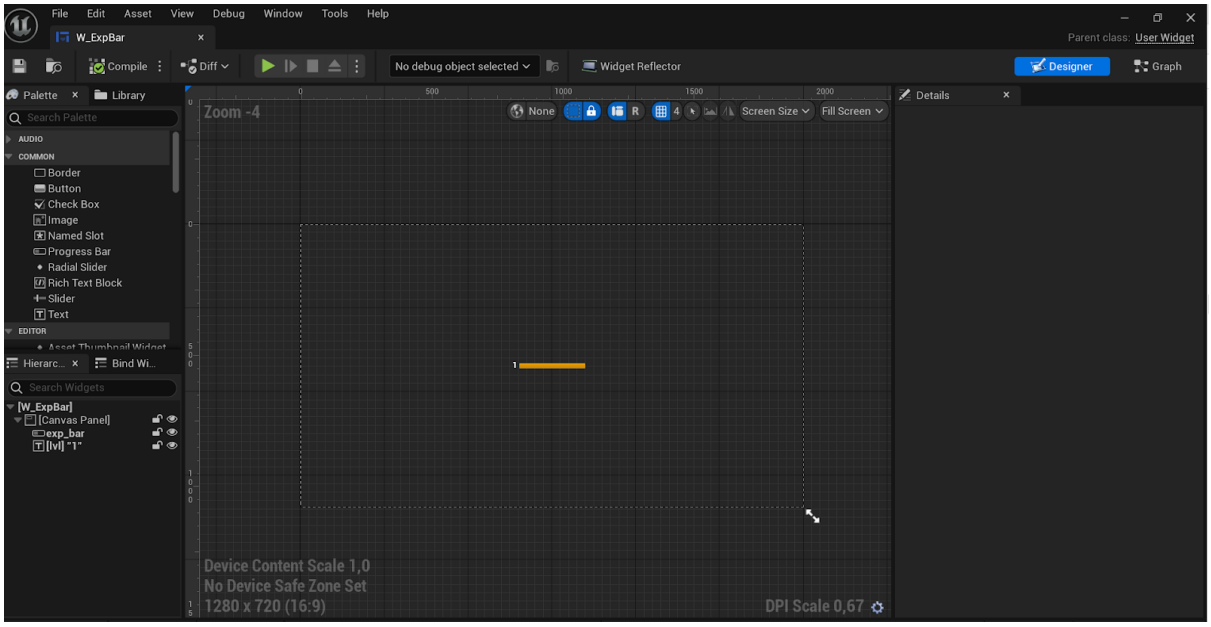


Рис. 4.4.9

W_HealthBar - progress bar з назвою hp bar . (Рис. 4.4.10)

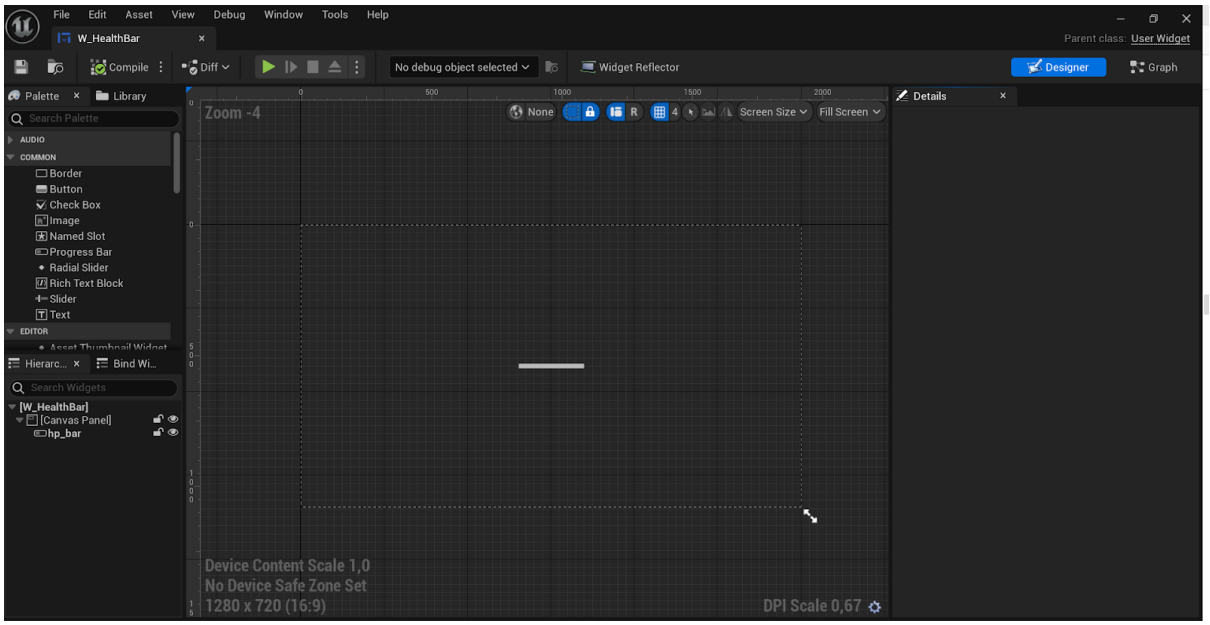


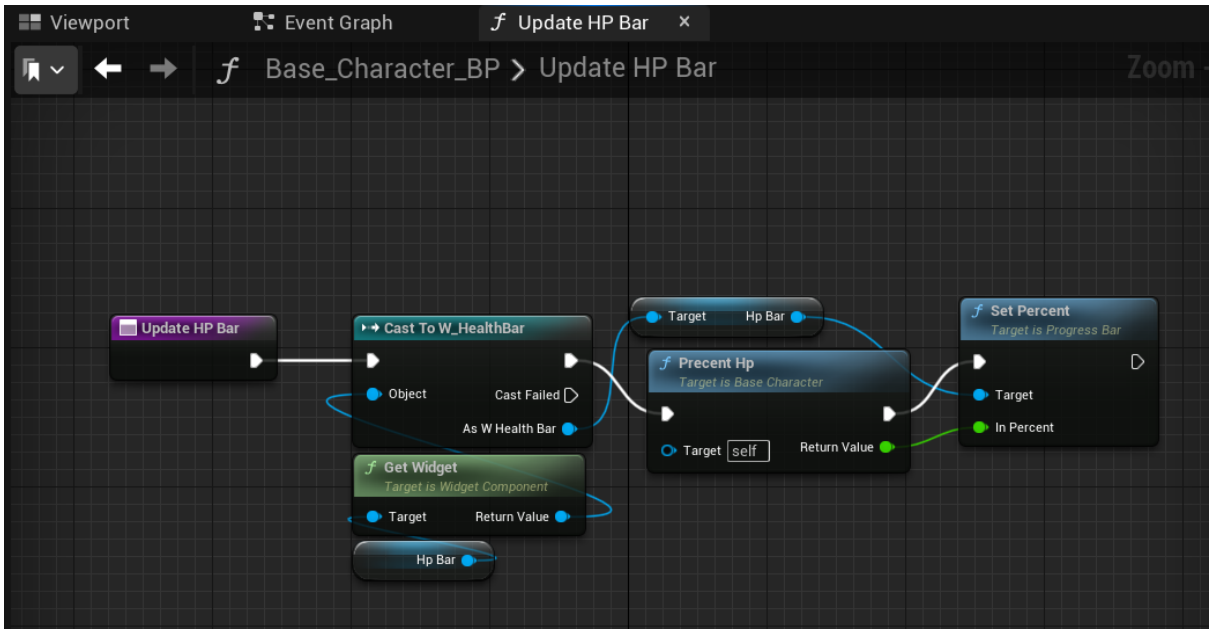
Рис. 4.4.10

4.4.9 Підключення інтерфейсів користувача до функцій c++ класса за допомогою блупрінта

Відкриємо наш BP Third Person, викликаємо Event Tick і створимо 2 функції Update HP Bar і Update Exp Bar. Див. Рис.4.4.11

Рис.4.4.11

В функціїї Update HP Bar, отримуємо посилання на компонент W_HealthBar, викликаємо функцію PercentHP яку ми написали у c++ класі Base Character, отриманні дані з цієї функції передаємо в функцію Set Percent, яка приймає значення від 0 до 100, і переміщує полоску progress bar на відповідне значення.(Рис.4.4.12)



(Рис.4.4.12)

У функціїї Update Exp Bar робимо аналогічну логіку для progress bar і передаємо значення поточного рівня в текстовий блок який буде відображати його на екрані (Рис. 4.4.13)

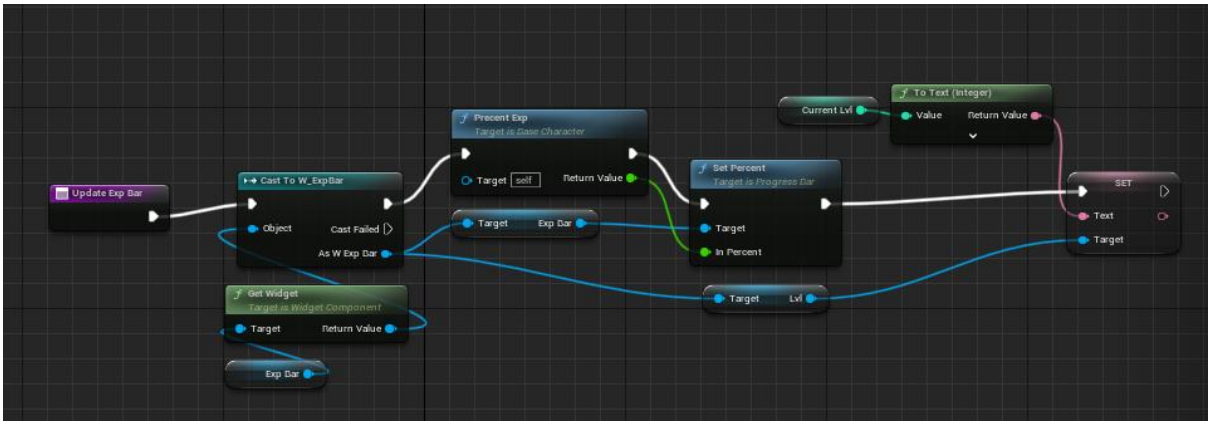


Рис. 4.4.13

4.4.10 Налаштування базового анімаційного блупрінта

В ABP Third Person створимо дві логічні зміні IsAiming та IsRecoil (Рис. 4.4.14)

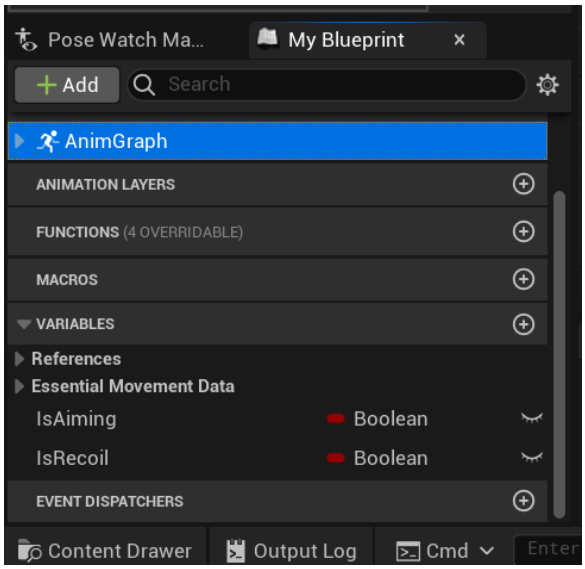


Рис. 4.4.14

Створимо два додаткові стани Aiming та Shoot з відповідними анімціями, та приєднаємо їх до стану Idle та Walk/Run як показано на рис.4.4.15.

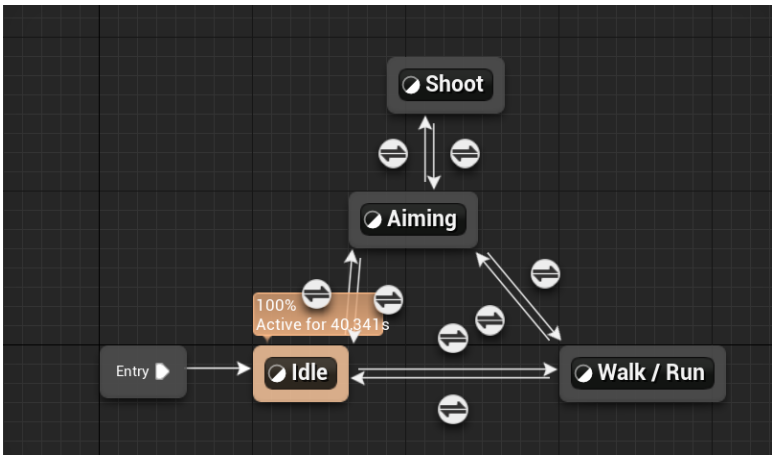


Рис. 4.4.15

На вхід до стану Aiming підключаємо зміну IsAiming, на вихід така сама перевірка але додаємо блок NOT, що повертає true якщо ми перестаємо прицілюватись. Між станом Aiming та Shoot робимо аналогічний зв'язок але зі зміною IsRecoil. Див. рис. 4.4.16 та рис. 4.4.17

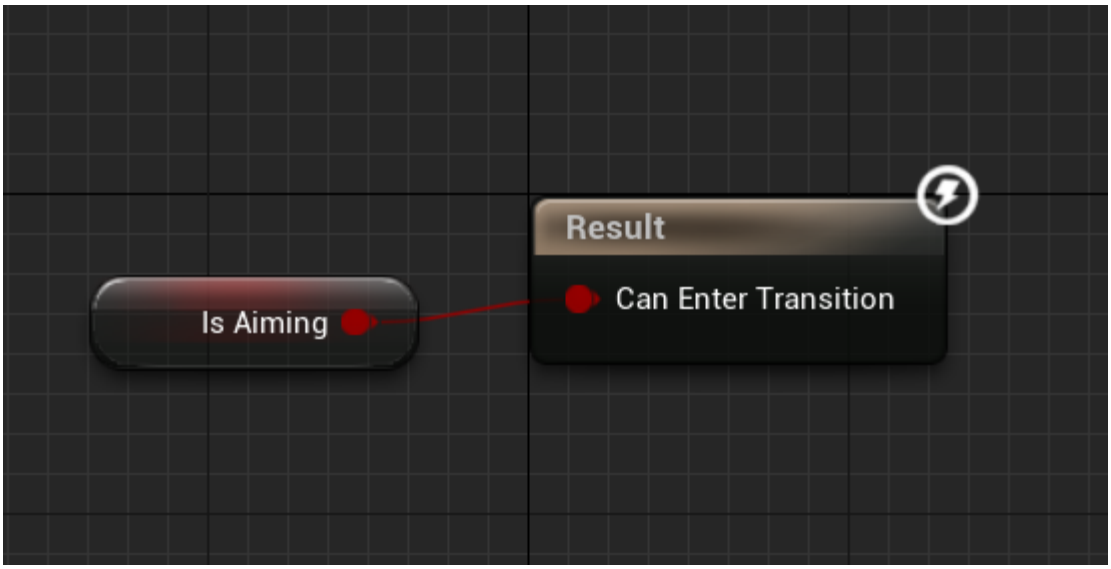


рис. 4.4.16

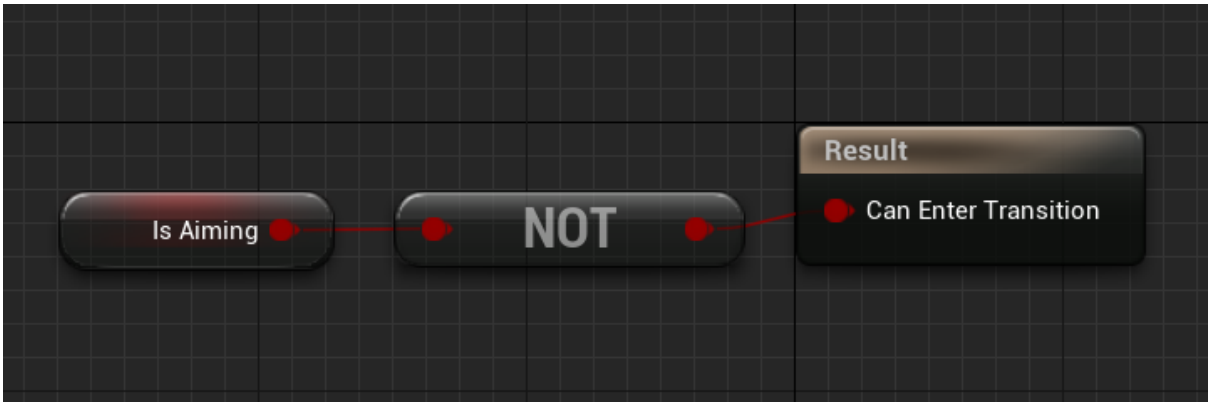


рис. 4.4.17

У графі EventGraph додаємо до шаблону ще два з'єднання, в яких оновлюємо змінні IsAiming та IsRecoil відповідними змінами з класу Base Character.

В анімації пострілу додамо подію Bow reload end, і перемістимо її на останній кадр як зображено на рис. 4.4.18

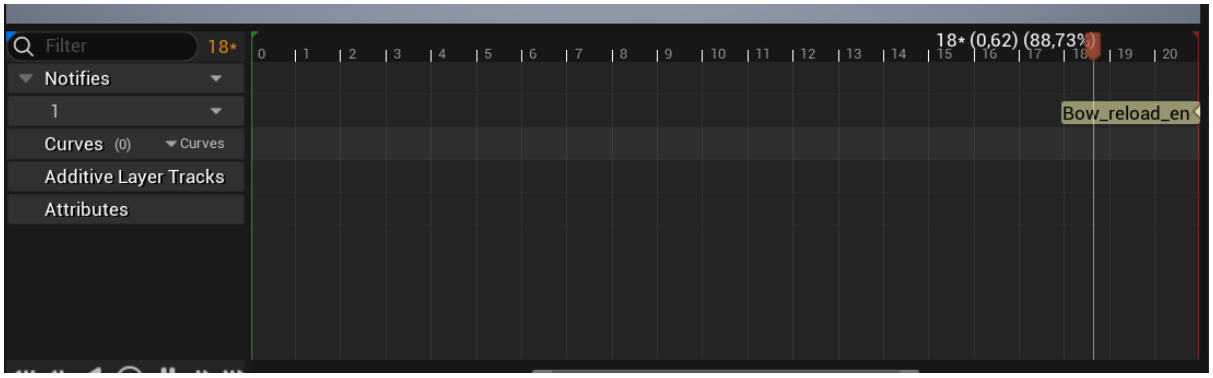


рис. 4.4.18

В графі EventGraph анімаційного блупрінта ABP Third Person, викличемо цю подію і посилаючись на клас Base Character, змінимо значення Recoil на false, закінчуючи стан перезарядки після завершення анімації пострілу. (Рис. 4.4.19)

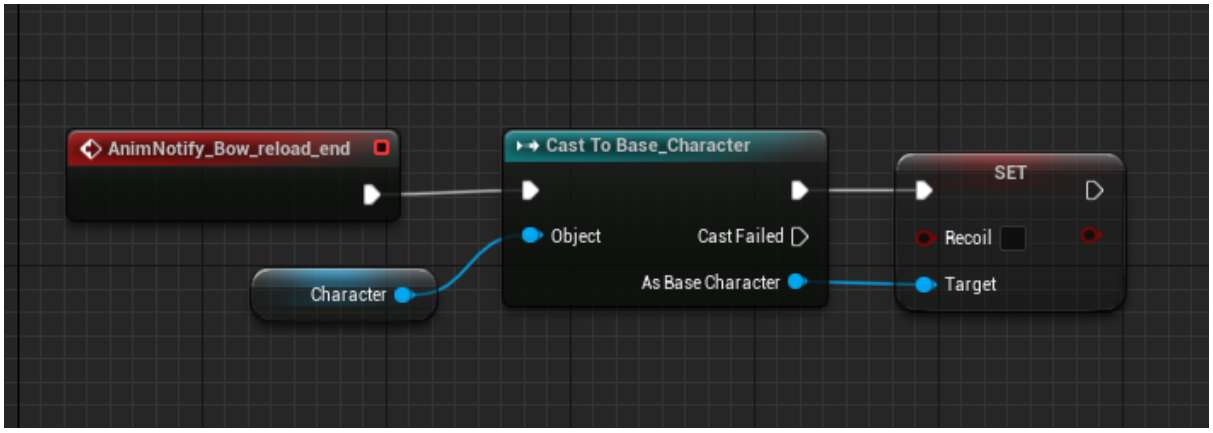


Рис. 4.4.19

4.5 Тестування комп’ютерної гри

Тестування комп'ютерної гри - це важлива частина розробки ігор, яка спрямована на виявлення помилок, вдосконалення геймплею та забезпечення якості продукту перед випуском. Ця дія включає виконання програмних компонентів, використовуючи ручні або автоматизовані інструменти для оцінки його властивостей. Головна мета - виявлення недоліків, прогавин або невідповідностей вимогам, а не лише співпадіння з вже наявними вимогами.

Існують різні аспекти тестування гри:

Функціональне тестування: Перевірка різних аспектів гри, таких як рух персонажів, реакція на дії гравця, правильність виконання завдань, обробка колізій тощо.

Тестування на стабільність: Перевірка на стійкість гри до можливих витоків пам'яті, падінь програми або непередбачуваних помилок.

Тестування на продуктивність: Визначення швидкодії гри на різних пристроях та у різних умовах.

Тестування сумісності: Перевірка роботи гри на різних платформах, ОС або пристроях.

Тестування на користувацький досвід (UX): Оцінка зручності управління, інтуїтивності інтерфейсу, а також загального задоволення гравців від гри.

Тестування на мережі: Якщо гра підтримує онлайн, тестування мережевої взаємодії, стійкості серверів, мережевих затримок тощо.

Важливість цього процесу полягає в тому, що виявлені помилки або недоліки можна виправити до того, як продукт потрапить до користувачів. Якщо програмне забезпечення перевірено належним чином, це гарантує його надійність, безпеку та ефективність, що в результаті приносить користь у вигляді економії часу, фінансових ресурсів та задоволення клієнтів.

ВИСНОВКИ

У дипломній роботі було вирішено актуальну задачу розробки ігрового застосунку у жанрі RPG. Під час виконання роботи був проведений аналіз предметної області, включаючи вимоги до розробки та обґрунтування вибору програмного забезпечення для завдання.

У розділі аналізу були описані вимоги до створення ігрового застосунку та обгрунтовано вибір мови програмування та середовища розробки. Аналіз можливих засобів реалізації допоміг обрати необхідний інструментарій для проекту. В результаті аналізу було зроблено висновок, що Unreal Engine 5 є найбільш підходящим для розробки гри.

Гра була створена на обраному двигуні Unreal Engine 5 без підключення до мережі Інтернет та можливістю гри проти штучного інтелекту.

Також було проведено аналіз предметної сфери розробки ігор у жанрі RPG. Розглянуто поняття " RPG " та визначено його значення і актуальність для проекту.

Для ігрового застосунку було створено технічне завдання, визначено основних учасників системи та основні вимоги до застосунку. Розроблено сценарій використання системи, а також побудовано діаграму прецедентів та станів, що створили структурну основу для системи та проілюстрували взаємодію класів, їх атрибутів та зв'язків.

Також було детально визначено поняття ігрового інтерфейсу. Наступним кроком була реалізація програмного забезпечення та його відповідне тестування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційний сайт Unreal Engine: https://www.unrealengine.com/

- Офіційний веб-сайт Unreal Engine містить багато ресурсів, документацію та підручники для розробки ігор на цьому двигуні.

2. Документація Unreal Engine: https://docs.unrealengine.com/en-US/index.html

- Офіційна документація Unreal Engine містить докладні пояснення та приклади для різних аспектів розробки ігор на цьому двигуні.

3. Unreal Engine YouTube-канал: https://www.youtube.com/user/UnrealDevelopmentKit

- Офіційний YouTube-канал Unreal Engine містить відеоуроки та підказки з розробки ігор на Unreal Engine.

4. Форум Unreal Engine: https://forums.unrealengine.com/

- Форум спільноти Unreal Engine, де ви можете знайти відповіді на питання та консультуватися з іншими розробниками.

5. Unreal Engine на GitHub: https://github.com/EpicGames/UnrealEngine

- GitHub-репозиторій Unreal Engine містить відкритий код двигуна та додаткові ресурси.

6. Unreal Engine 4 Documentation - Blueprint Visual Scripting: https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html

- Офіційна документація Unreal Engine щодо Blueprint, що дозволяє створювати ігри без програмування.

7. Книга "Learning C++ by Creating Games with Unreal Engine 4" автора William Sherif.

- Ця книга надає інформацію щодо програмування ігор на Unreal Engine з використанням мови програмування C++.

8. Unreal Engine Tutorials на YouTube: https://www.youtube.com/user/UnrealDevelopmentKit/playlists

- Велика кількість відеоуроків на YouTube, що охоплюють різні аспекти розробки ігор на Unreal Engine.

9. Документація з розробки ігор загалом: "The Art of Game Design" від Jesse Schell та "Game Design: Theory & Practice" від Richard Rouse III.

- Ці книги допоможуть зрозуміти принципи створення ігор та геймдизайну.

10. Unity vs. Unreal Engine: Choosing the Best Game Engine - Стаття на сайті Gamasutra: https://www.gamasutra.com/blogs/PhuongNguyen/20190603/343508/Unity_vs_Unreal_Engine_Choosing_the_Best_Game_Engine.php

- Ця стаття порівнює Unity і Unreal Engine та допоможе обрати підходящий двигун для проекту.

11. Gamasutra (https://www.gamasutra.com/): Веб-сайт, присвячений індустрії геймдеву, зі статтями та блогами від професіоналів галузі.
12. https://www.mixamo.com/#/

- mixamo сайт

13. https://terribilisstudio.fr/?section=MC

- програма mixamo converter

14. Unreal Engine [Електронний ресурс] / Wikipedia https://uk.wikipedia.org/wiki/Unreal_Engine
15. IGN - https://www.ign.com/

- Цей сайт забезпечує огляди, новини та інтерв'ю з розробниками ігор, а також відеоогляди та додатковий контент

16. Reddit - для спільноти low poly: https://www.reddit.com/r/low_poly/

- де користувачі діляться своїми роботами, порадами та питаннями щодо low poly творчості.

17. Polycount - https://polycount.com/

- Це спільнота художників та розробників ігор, де обговорюється створення 3D-моделей, включаючи low poly

18. ArtStation - https://www.artstation.com/

- Це платформа для художників, де можно знайти багато робіт у стилі low poly та поради від професіоналів

19. https://docs.unrealengine.com/5.0/en-US/creating-blueprint-classes-in-unreal-engine/

- An overview of the C++ Class Wizard in Unreal Engine.

20. cplusplus.com

- Тут можна знайти огляди, приклади коду, статті та форуми для обговорень.

21. cppreference.com

- Це онлайн-довідник, який містить багато інформації про функції, бібліотеки та основи мови програмування C++

ДОДАТОК

https://we.tl/t-owFxhfLHfe - лінка на завантаження гри **“ Arden Dungeon ”**

LvlStatsStruc - це структура (власний тип даних, як int, float і т.д), тут ми в блупрінті налаштовуємо основні характеристики персонажа такі як хп урон і досвід до іншого лвл.:

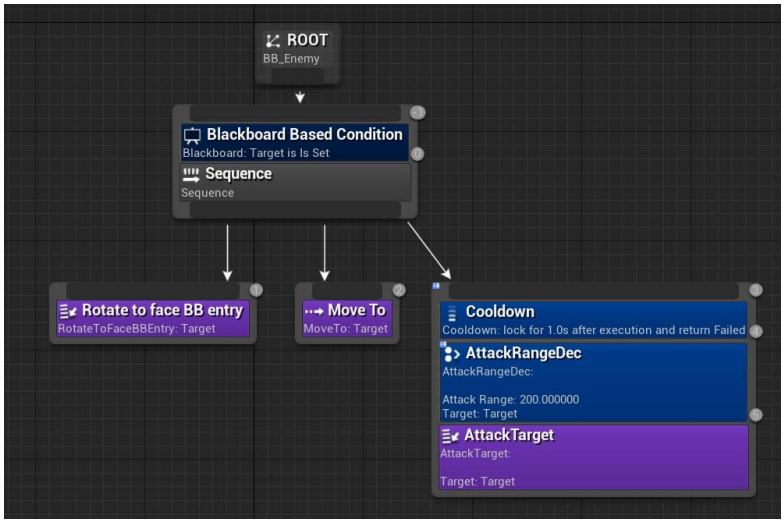
```
#include "CoreMinimal.h"
#include "UObject/NoExportTypes.h"
#include "LvlStatsStruc.generated.h"
USTRUCT(BlueprintType)
struct ARDENDUNGEON_API FLvlStatsStruc
{
	GENERATED_BODY()
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	float ExpToNextLvl;
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	float MaxHP;
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	float BaseDamage;
};
```

BaseEnemyAiController - це те що керує ворогом, у гравця це стандартний player controller з набором стандартних функцій які нам підходять, а у ворога ми задаємо свою логіку через дерево поведінки (Behaivior Tree)

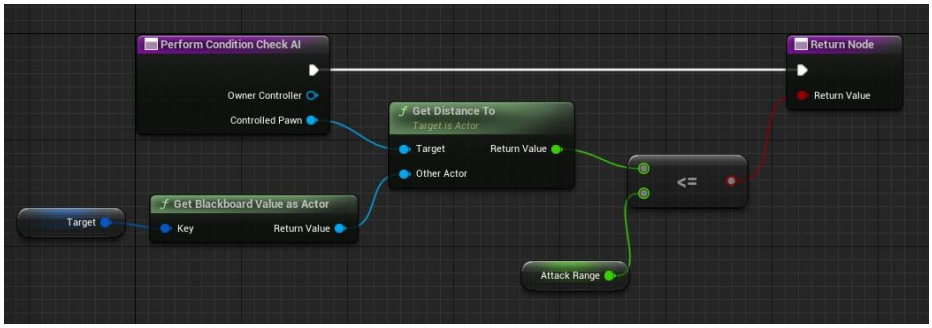
```
#include "CoreMinimal.h"
#include "AIController.h"
#include "Perception/AiPerceptionComponent.h"
#include "BehaviorTree/BehaviorTree.h"
#include "BehaviorTree/BlackboardComponent.h"
#include "BaseEnemyAiController.generated.h"
UCLASS()
class ARDENDUNGEON_API ABaseEnemyAiController : public AAIController
{
	GENERATED_BODY()
	virtual void BeginPlay() override;
	UPROPERTY(EditDefaultsOnly, Category = "Blackboard")
	UBlackboardData* BlackboardToUse;

	UPROPERTY(EditDefaultsOnly, Category = "Blackboard")
	UBehaviorTree* BehaviorTreeToUse;

	UPROPERTY()
	UBlackboardComponent* BB;
```

```
	UPROPERTY(EditDefaultsOnly, Category = "Blackboard")
	TSubclassOf<UAISense> SightSense;

	UPROPERTY(VisibleAnywhere, Category = "AI")
	UAIPerceptionComponent* Wolf_Perception;
	UFUNCTION()
	void SenseStuff(AActor* UpdatedActor, FAIStimulus Stimulus);
};
============================
#include "BaseEnemyAiController.h"

void ABaseEnemyAiController::BeginPlay()
{
	Super::BeginPlay();

	if (!ensure(BlackboardToUse)) { return; }
	else {
		UseBlackboard(BlackboardToUse, BB);
	}

	if (!ensure(BehaviorTreeToUse)) { return; }
	else {
		RunBehaviorTree(BehaviorTreeToUse);
	}

	Wolf_Perception = FindComponentByClass<UAIPerceptionComponent>();
	if (!ensure(Wolf_Perception)) { return; }
	else {
		Wolf_Perception->OnTargetPerceptionUpdated.AddDynamic(this,
&ABaseEnemyAiController::SenseStuff);
	}
}

void ABaseEnemyAiController::SenseStuff(AActor* UpdatedActor, FAIStimulus Stimulus)
{
	if (Stimulus.WasSuccessfullySensed())
	{
		if(UpdatedActor->ActorHasTag("Player"))	BB->SetValueAsObject("Target",
UpdatedActor);
	}
}
```

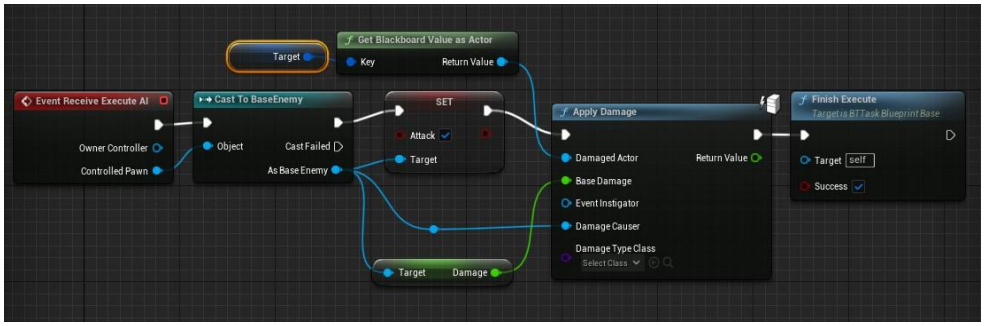
Behaivior Tree - дерево поведінкі ботів (встроєний функціонал), в нашому випадку, якщо значення дошки не пусте (аналог зміних в c++, тільки дошка привязанна до дерева), ми повертаєм ворога обличям до цілі (яку отримали в BaseEnemyAiController), біжемо до неї, і якщо дистанція до цілі 200 пунктів то ми атакуємо, затримка на атаку 1с.



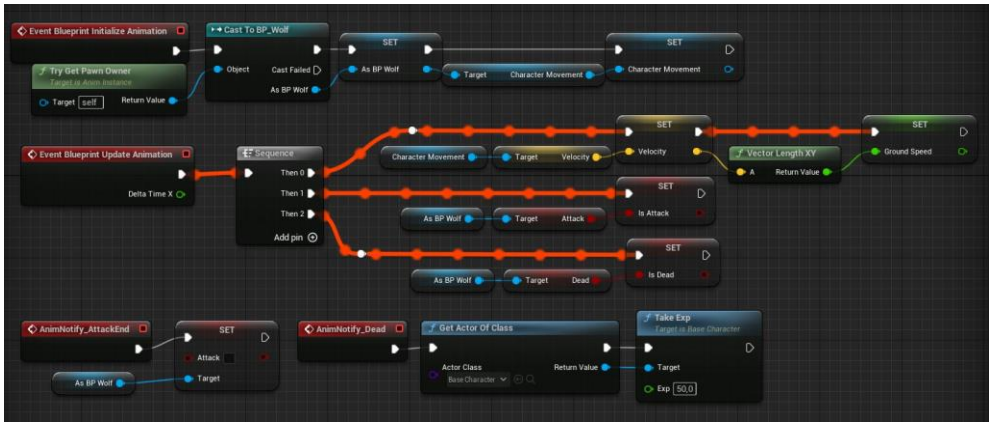
Behaivior Tree AttackRangeDec - перевірка дистанції до цілі



Behaivior Tree AttackTarget - атака цілі



Enemy Animation - подія дозволяє нам атакувати тільки після закінчення анімації атаки, друга подія в кінці анімації смерті ми даємо 50 досвіду персонажу.



BaseEnemy - основа наших ворогів (всіх)

```
#include "BaseEnemy.h"
#include "Perception/AiPerceptionComponent.h"
ABaseEnemy::ABaseEnemy()
{
	PrimaryActorTick.bCanEverTick = true;
}

void ABaseEnemy::BeginPlay()
{
	Super::BeginPlay();
}

float ABaseEnemy::TakeDamage(float DamageAmount, struct FDamageEvent const& DamageEvent, class
AController* EventInstigator, AActor* DamageCauser)
{
	float dmg = Super::TakeDamage(DamageAmount, DamageEvent, EventInstigator, DamageCauser);
	HP -= dmg;
	if (HP <= 0) Dead = true;
	return dmg;
}
```

==========

```
#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "BaseEnemy.generated.h"


UCLASS()
class ARDENDUNGEON_API ABaseEnemy : public ACharacter
{
	GENERATED_BODY()

public:

	ABaseEnemy();

protected:

	virtual void BeginPlay() override;

	UPROPERTY(BlueprintReadWrite)
	bool Attack;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	float Damage;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	float HP;

	UPROPERTY(BlueprintReadOnly)
```

```
	bool Dead;

	virtual float TakeDamage(float DamageAmount, struct FDamageEvent const& DamageEvent, class
AController* EventInstigator, AActor* DamageCauser) override;

};
```

BaseCharacter - це основа нашого персонажа, на основі цього класу робимо блупрінт

```
#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "Logging/LogMacros.h"
#include "Camera/CameraComponent.h"
#include "LvlStatsStruc.h"
#include "Base_Character.generated.h"

class USpringArmComponent;
class UCameraComponent;
class UInputMappingContext;
class UInputAction;
struct FInputActionValue;

DECLARE_LOG_CATEGORY_EXTERN(LogTemplateCharacter, Log, All);

UCLASS()
class ARDENDUNGEON_API ABase_Character : public ACharacter
{
	GENERATED_BODY()

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta =
(AllowPrivateAccess = "true"))
	USpringArmComponent* CameraBoom;

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

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess =
"true"))
	UInputMappingContext* DefaultMappingContext;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess =
"true"))
	UInputAction* JumpAction;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess =
"true"))
	UInputAction* MoveAction;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess =
"true"))
```

```
	UInputAction* LookAction;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess =
"true"))
	UInputAction* RunAction;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess =
"true"))
	UInputAction* RMBAction;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess =
"true"))
	UInputAction* LMBAction;

protected:
	UPROPERTY(BlueprintReadWrite)
	bool Aiming;

	UPROPERTY(BlueprintReadWrite)
	bool Recoil;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Components)
	UClass* Arrow;

	UPROPERTY(EditAnywhere, Category = Stats)
	FLvlStatsStruc LvlStats[5];

	float CurrentHealth;
	float CurrentExp;

	UPROPERTY(BlueprintReadWrite)
	int CurrentLvl;

	UPROPERTY(BlueprintReadWrite)
	bool Dead;

protected:
	virtual void BeginPlay() override;

	void Move(const FInputActionValue& Value);

	void Look(const FInputActionValue& Value);

	void RunStart();
	void RunCompleted();

	void AimingBowEnable();
	void AimingBowDisable();

	void ShootBow();

	void LvlUp();
```

```
	UFUNCTION(BlueprintCallable)
	float PercentHp();

	UFUNCTION(BlueprintCallable)
	float PercentExp();

	UFUNCTION(BlueprintCallable)
	void TakeExp(float exp);

	virtual float TakeDamage (float DamageAmount,struct FDamageEvent const& DamageEvent,class
AController* EventInstigator,    AActor* DamageCauser) override;

public:
	ABase_Character();

	virtual void Tick(float DeltaTime) override;

	virtual void SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent) override;

	FORCEINLINE class USpringArmComponent* GetCameraBoom() const { return CameraBoom; }

	FORCEINLINE class UCameraComponent* GetFollowCamera() const { return FollowCamera; }
};
```

```
#include "Base_Character.h"
#include "Components/CapsuleComponent.h"
#include "GameFramework/Actor.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "GameFramework/SpringArmComponent.h"
#include "GameFramework/Controller.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "EnhancedInputComponent.h"
#include "EnhancedInputSubsystems.h"
#include "InputActionValue.h"
#include "Components/SceneComponent.h"
#include "Kismet/KismetMathLibrary.h"
#include "Camera/CameraComponent.h"

DEFINE_LOG_CATEGORY(LogTemplateCharacter);

// Sets default values
ABase_Character::ABase_Character()
{
	// Set this character to call Tick() every frame.  You can turn this off to improve performance if you
don't need it.
	PrimaryActorTick.bCanEverTick = true;

	// Set size for collision capsule
	GetCapsuleComponent()->InitCapsuleSize(42.f, 96.0f);
```

```
	// Don't rotate when the controller rotates. Let that just affect the camera.
	bUseControllerRotationPitch = false;
	bUseControllerRotationYaw = false;
	bUseControllerRotationRoll = false;

	// Configure character movement
	GetCharacterMovement()->bOrientRotationToMovement = true; // Character moves in the direction of input...
	GetCharacterMovement()->RotationRate = FRotator(0.0f, 500.0f, 0.0f); // ...at this rotation rate
	GetCharacterMovement()->JumpZVelocity = 700.f;
	GetCharacterMovement()->AirControl = 0.35f;
	GetCharacterMovement()->MaxWalkSpeed = 500.f;
	GetCharacterMovement()->MinAnalogWalkSpeed = 20.f;
	GetCharacterMovement()->BrakingDecelerationWalking = 2000.f;
	GetCharacterMovement()->BrakingDecelerationFalling = 1500.0f;

	// Create a camera boom (pulls in towards the player if there is a collision)
	CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));
	CameraBoom->SetupAttachment(RootComponent);
	CameraBoom->TargetArmLength = 400.0f; // The camera follows at this distance behind the character
	CameraBoom->bUsePawnControlRotation = true; // Rotate the arm based on the controller

	// Create a follow camera
	FollowCamera = CreateDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"));
	FollowCamera->SetupAttachment(CameraBoom, USpringArmComponent::SocketName); // Attach the camera to the end of the boom and let the boom adjust to match the controller orientation
	FollowCamera->bUsePawnControlRotation = true; // Camera does not rotate relative to arm
}

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

	//Add Input Mapping Context
	if (APlayerController* PlayerController = Cast<APlayerController>(Controller))
	{
		if (UEnhancedInputLocalPlayerSubsystem* Subsystem = ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerController->GetLocalPlayer()))
		{
			Subsystem->AddMappingContext(DefaultMappingContext, 0);
		}
	}

	//up the level from 0 to 1
	if (CurrentLvl == 0)
	{
		LvlUp();
	}
```

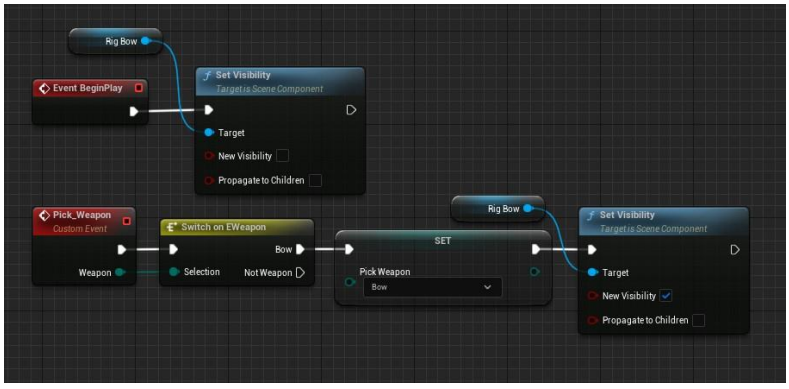
```
}

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

}

// Called to bind functionality to input
void ABase_Character::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
	// Set up action bindings
	if (UEnhancedInputComponent* EnhancedInputComponent =
Cast<UEnhancedInputComponent>(PlayerInputComponent)) {

		// Jumping
		EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Started, this,
&ACharacter::Jump);
		EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Completed, this,
&ACharacter::StopJumping);

		// Moving
		EnhancedInputComponent->BindAction(MoveAction, ETriggerEvent::Triggered, this,
&ABase_Character::Move);

		// Looking
		EnhancedInputComponent->BindAction(LookAction, ETriggerEvent::Triggered, this,
&ABase_Character::Look);

		//Run
		EnhancedInputComponent->BindAction(RunAction, ETriggerEvent::Started, this,
&ABase_Character::RunStart);
		EnhancedInputComponent->BindAction(RunAction, ETriggerEvent::Completed, this,
&ABase_Character::RunCompleted);

		//Aiming
		EnhancedInputComponent->BindAction(RMBAction, ETriggerEvent::Triggered, this,
&ABase_Character::AimingBowEnable);
		EnhancedInputComponent->BindAction(RMBAction, ETriggerEvent::Completed, this,
&ABase_Character::AimingBowDisable);

		//Shoot Bow
		EnhancedInputComponent->BindAction(LMBAction, ETriggerEvent::Triggered, this,
&ABase_Character::ShootBow);

	}
	else
	{
		UE_LOG(LogTemplateCharacter, Error, TEXT("'%s' Failed to find an Enhanced Input
component! This template is built to use the Enhanced Input system. If you intend to use the legacy system,
then you will need to update this C++ file."), *GetNameSafe(this));
```

```
	}
}

void ABase_Character::LvlUp()
{
	CurrentLvl++;
	CurrentExp = 0;
	CurrentHealth = LvlStats[CurrentLvl].MaxHP;
}

float ABase_Character::PercentHp()
{
	return CurrentHealth / LvlStats[CurrentLvl].MaxHP;
}

float ABase_Character::PercentExp()
{
	return CurrentExp / LvlStats[CurrentLvl].ExpToNextLvl;
}

void ABase_Character::TakeExp(float exp)
{
	CurrentExp += exp;
	if (CurrentExp >= LvlStats[CurrentLvl].ExpToNextLvl) LvlUp();
}

float ABase_Character::TakeDamage(float DamageAmount, struct FDamageEvent const& DamageEvent,
class AController* EventInstigator, AActor* DamageCauser)
{
	float dmg = Super::TakeDamage(DamageAmount, DamageEvent,EventInstigator,DamageCauser);
	CurrentHealth -= dmg;
	if (CurrentHealth <= 0) Dead = true;
	return dmg;
}

void ABase_Character::Move(const FInputActionValue& Value)
{
	// input is a Vector2D
	FVector2D MovementVector = Value.Get<FVector2D>();

	if (Controller != nullptr)
	{
		// find out which way is forward
		const FRotator Rotation = Controller->GetControlRotation();
		const FRotator YawRotation(0, Rotation.Yaw, 0);

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

		// get right vector
		const FVector RightDirection = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);
```

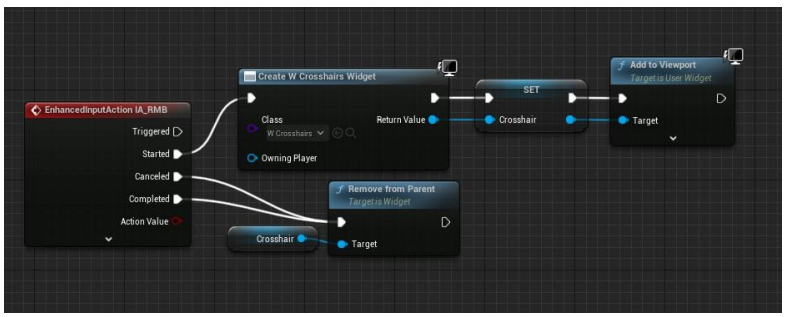
```
                // add movement
                AddMovementInput(ForwardDirection, MovementVector.Y);
                AddMovementInput(RightDirection, MovementVector.X);
        }
}

void ABase_Character::Look(const FInputActionValue& Value)
{
        // input is a Vector2D
        FVector2D LookAxisVector = Value.Get<FVector2D>();

        if (Controller != nullptr)
        {
                // add yaw and pitch input to controller
                AddControllerYawInput(LookAxisVector.X);
                AddControllerPitchInput(LookAxisVector.Y);
        }
}

void ABase_Character::RunStart()
{
        if (!Aiming)
        {
                GetCharacterMovement()->MaxWalkSpeed = 500.f;
        }
}

void ABase_Character::RunCompleted()
{
        GetCharacterMovement()->MaxWalkSpeed = 300.f;
}

void ABase_Character::AimingBowEnable()
{
        Aiming = true;
        CameraBoom->TargetArmLength = 200.f;
}

void ABase_Character::AimingBowDisable()
{
        Aiming = false;
        CameraBoom->TargetArmLength = 400.f;
}

void ABase_Character::ShootBow()
{
        if (Aiming && !Recoil)
        {
                Recoil = true;

                //create an arrow from the bow towards the center of the screen
                FActorSpawnParameters SpawnInfo;
```

```
                SpawnInfo.Instigator = this;
                SpawnInfo.Owner = this;
                const FVector transfom_socket = GetMesh()->GetSocketLocation("arrow_socket");
                const FRotator rotator_camera = FollowCamera->GetComponentRotation();
                const FVector scale = FVector(1, 1, 1);
                FTransform transform = FTransform(rotator_camera, transfom_socket, scale);
                AActor* new_arrow = GetWorld()->SpawnActor(Arrow, &transform, SpawnInfo);

                //create an trace from the bow towards the center of the screen
                FVector Start = GetMesh()->GetSocketLocation("arrow_socket");
                FVector End = Start + FollowCamera->GetForwardVector() * 5000.f;
                FHitResult HitResult;
                FCollisionQueryParams Params;
                Params.AddIgnoredActor(this);

                // Apply damage to target
                if (GetWorld()->LineTraceSingleByChannel(HitResult, Start, End,
ECollisionChannel::ECC_Camera, Params, FCollisionResponseParams()))
                {
                        HitResult.GetActor()->TakeDamage(LvlStats[CurrentLvl].BaseDamage,
FDamageEvent(), GetInstigatorController(), this);
                }
        }
}
```

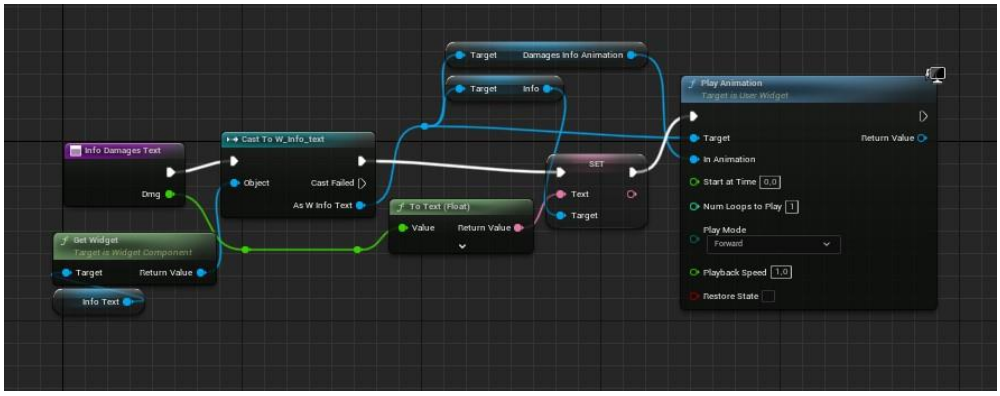
Blueprint BaseCharacter bow - робимо лук в руці невидимим, коли спрацьовує подія pick weapon ми перевіряємо яку зброю ми беремо і робимо його видимим



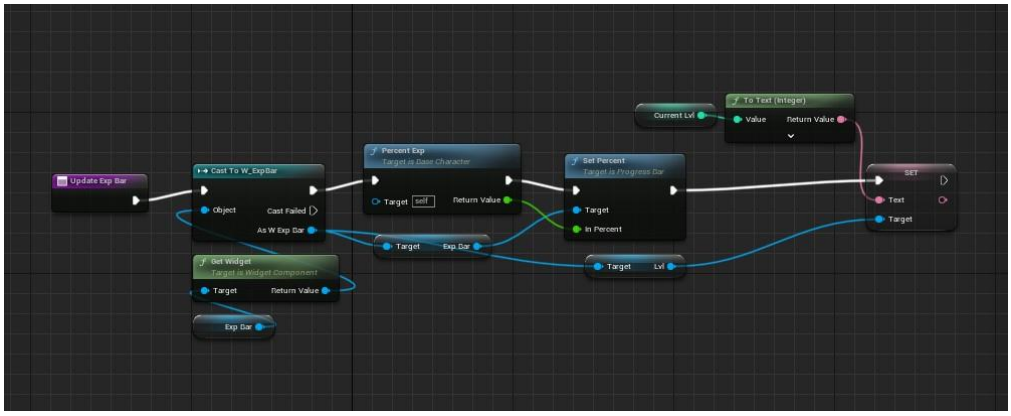
Blueprint BaseCharacter crosshair - відображаємо приціл коли нажимаємо праву кнопку миші



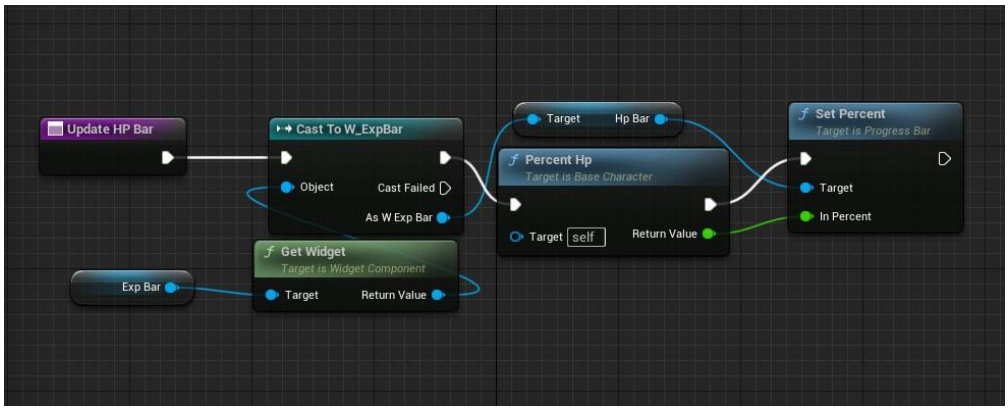
Blueprint BaseCharacter widget infotext - показуємо на екрані скільки отримали шкоди, і запускаємо анімацію переміщення тексту по екрану



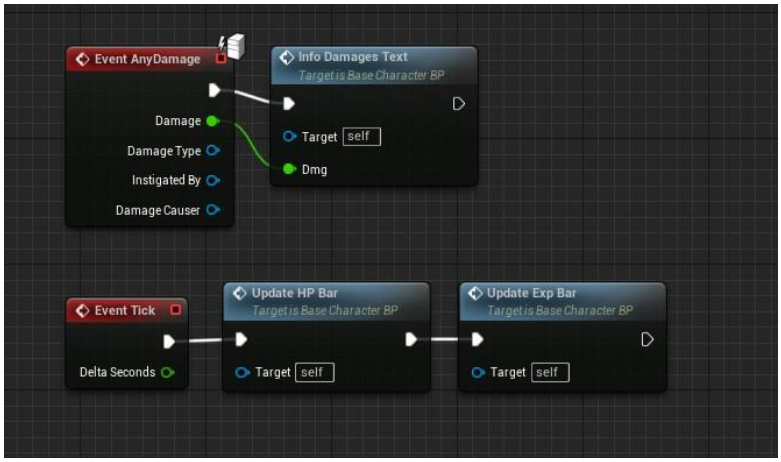
Blueprint BaseCharacter widget update exp bar - передаємо процентне значення (від 0 до 100) на прогрес бар (полоска заповнює бар залежно від проценту тобто значення 50 це середина бара) + відображаємо лвл в компоненті тексту



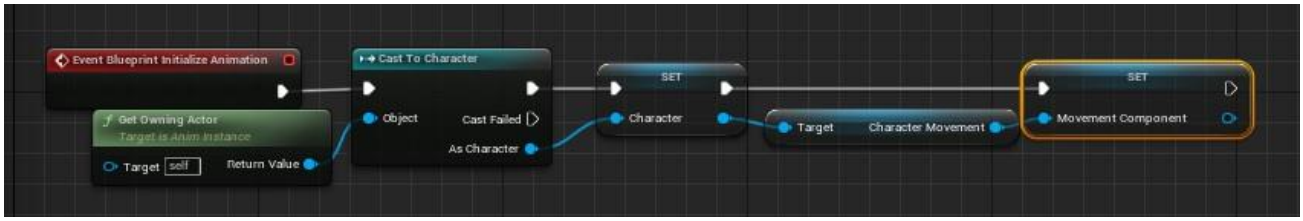
Blueprint BaseCharacter widget update hp bar - анологічно но для бара хп



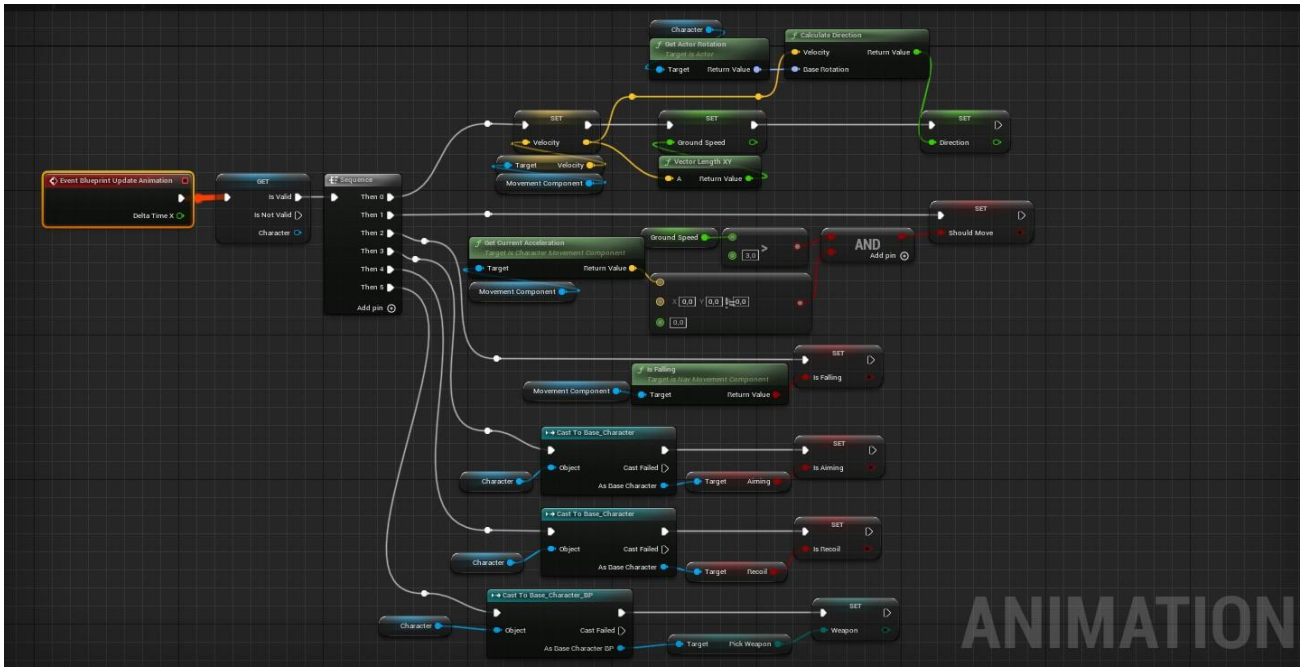
Blueprint BaseCharacter widget - тут ми викликаємо вище описані функції, info text коли отримуємо шкоду, hp і exp bar кожного кадру



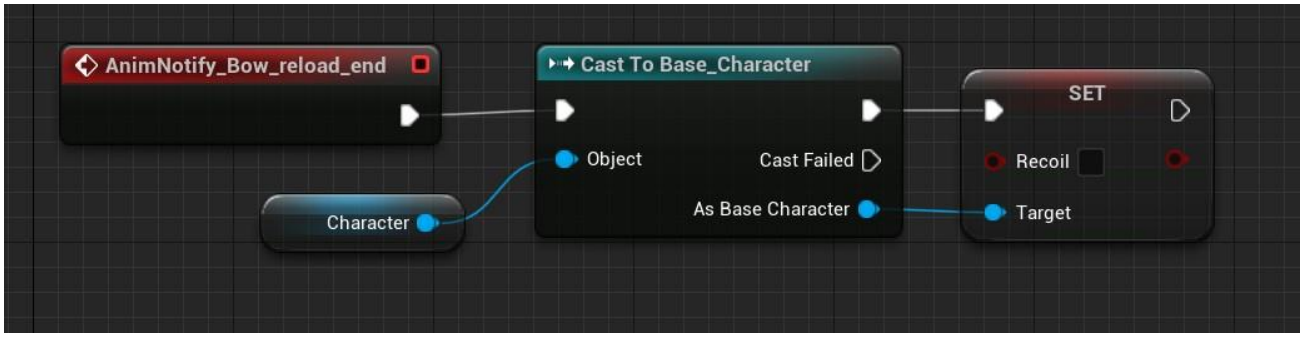
Animation component – це анім блупрінт, спочатку робимо посилання на блупрінт персонажу, і його компонент переміщення



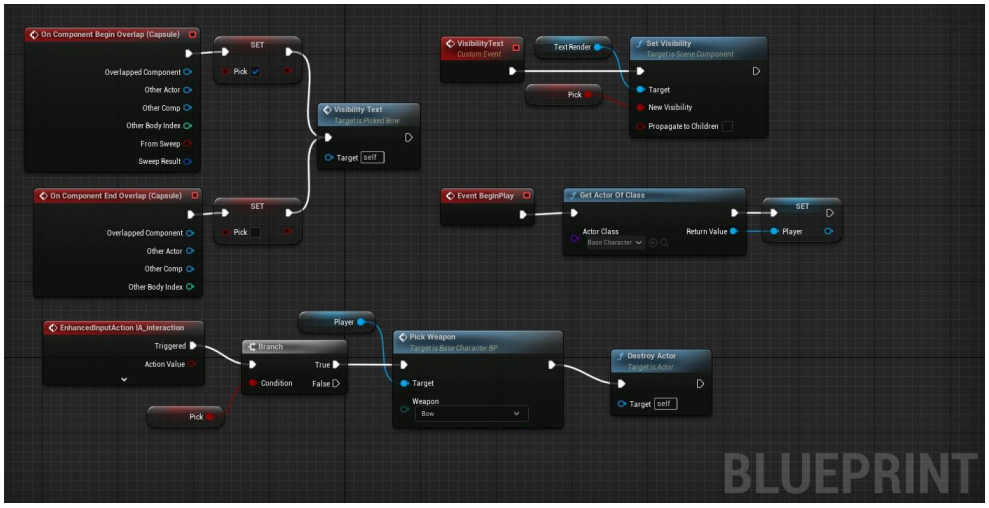
Animation get variable - беремо значення з блупрінта персонажа (кожного кадру), і зберігаємо їх в локальні зміні які відповідають за переключення анімації.



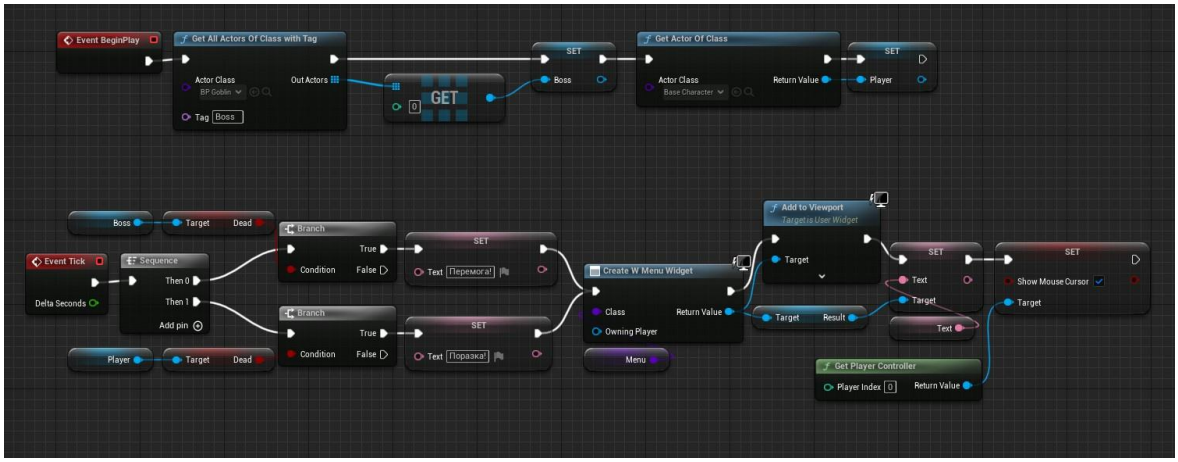
Animation notify - подія в кінці анімації пострілу, перемикає значення Recoil. Типа ми закінчили перезарядку лука, можна стріляти знову.



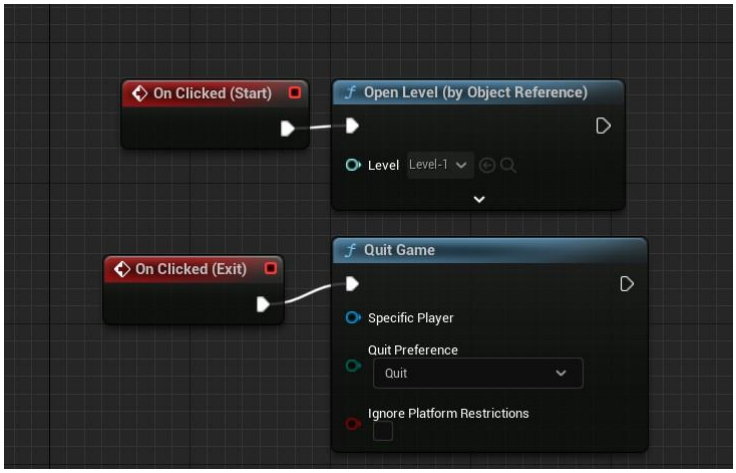
PickedBow - лук який ми підбираємо, якщо гравець зайшов в зону трігера, ми включаємо що лук можна підібрати, показуєм текст на яку кнопку жати, коли кнопку нажали, визиваємо подію pick weapon у персонажа, і видаляємо цей об’єкт.



MainGameMode - game mode загружається разом з левелом, і контролює гру. Також перевіряє умови гри, якщо бос або персонаж помер, то визиває меню і пише відповідний текст.



MainMenu - головне меню, це рівень з 2 кнопками, старт загружає наш рівень для гри, вихід відовідно вихід



EndGameMenu - меню кінця гри - загружає рівень головного меню