Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра програмних систем і технологій

УДК 004.925

На правах

рукопису

БР.ІПЗ - 31.00.00.00

ВИПУСКНА КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Тема: Візуалізація статичних і динамічних 3D об'єктів на основі векторної алгебри в графічних симуляціях

Спеціальність - 121 “Інженерія програмного забезпечення”

Виконав студент

ІПЗ-43_____/Валерій Старіков

(шифр групи)(підпис)(дата)(розшифровка підпису)

Науковий керівник

к.т.н._____/Максим Ткаченко

(посада) (підпис) (дата) (розшифровка підпису) s

Консультант з питань нормоконтролю

фахівець._____/Тамара ЧАПОВСЬКА

Допускається до захисту

з питань нормоконтролю

Завідувач кафедри

д.т.н., проф._____/Олексій БИЧКОВ

(посада) (підпис) (дата) (розшифровка підпису)

Київ – 2021

Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра програмних систем і технологій

Освітньо-кваліфікаційний рівень бакалавр

Спеціальність 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖЕНО

Зав.кафедри програмних систем і технологій

_____(Олексій БИЧКОВ)

(підпис) (прізвище та ініціали)

ЗАВДАННЯ

НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

Старікову Валерій Олександровичу

1. Тема бакалаврської роботи “Візуалізація статичних і динамічних 3D об'єктів на основі векторної алгебри в графічних симуляціях”, керівник проекту (роботи) Ткаченко Максим Васильович, к.т.н., асистент кафедри ПСТ затверджені наказом вищого навчального закладу від “11” листопада 2020 р. № 6

2. Строк подання студентом роботи _____

3. Вихідні дані до проекту (роботи) Концепції створення графічних симуляцій.

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

1. Ознайомлення з особливостями векторної алгебри для графічних симуляцій.
2. Створення графічної симуляції.
3. Практичне використання симуляції з елементами векторної алгебри.
4. Архітектура створеного програмного забезпечення.

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

1. Рисунок 2.3.1 (a) Вид орфографічної проекції зверху вниз і (b) отримане 2D-зображення на екрані

2. Рисунок 3.1.1 Загальний вигляд симуляції

3. Рисунок 3.1.2 Загальний вигляд симуляції (вид за персонажа)

4. Рисунок 3.1.2.1 Перша мапа

5. Рисунок 3.1.2.2 Друга мапа

6. Рисунок 3.1.3.1 Звуки

7. Рисунок 3.2.2.1 Модель персонажа

8. Рисунок 3.2.2.2 Анімація замаху для атаки

9. Рисунок 3.2.4.1 Реалізація робота анімацій

10. Рисунок 3.2.5.1 Інтерфейс 1

11. Рисунок 3.2.5.2 Інтерфейс 2

12. Рисунок 3.2.6.1 Blueprint шкала здоров’я

13. Рисунок 3.2.6.2 Blueprint шкала стаміни

14. Рисунок 3.3.1.1 Графічне представлення павука

15. Рисунок 3.3.1.2 Графічне представлення лицаря

16. Рисунок 3.3.3.1 Blueprint реалізація мобів

17. Рисунок 3.3.4.1 Навігація штучного інтелекту

18. Рисунок 3.3.5.1 Скріншот бою

19. Рисунок 3.4.2.1 Графічне представлення зброї

20. Рисунок 3.4.3.1 Графічне представлення плаваючих платформ

21. Рисунок 3.4.4.1 Графічне представлення таємних дверей

22. Рисунок 3.4.5.1 Графічне представлення простору спавну

23. Рисунок 3.5.1 Фінальний бос

24. Рисунок 4.1.1 Система c++ класів UE4

25. Рисунок 4.2.1 Діаграма c++ класів

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Ознайомлення з особливостями векторної алгебри для графічних симуляцій.	Ткаченко Максим Васильович		
Створення графічної симуляції.	Ткаченко Максим Васильович		
Використання векторної алгебри у практичній реалізації.	Ткаченко Максим Васильович		
Архітектура створеного погромного забезпечення.	Ткаченко Максим Васильович		

7. Дата видачі завдання 13 жовтня 2020 р.

Керівник _____ (Максим ТКАЧЕНКО)

Завдання прийняв до виконання _____ (Валерій СТАРІКОВ)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів бакалаврської роботи	Термін виконання етапів роботи	Примітка
1	Знаходження та вивчення теоретичних матеріалів		Виконано
2	Створення плану розробки програмного забезпечення		Виконано

3	Вивчення додаткових можливостей середовищ розробки		Виконано
4	Створення програмного забезпечення та додавання ігрових механік		Виконано
5	Кінцеві налаштування симуляції		Виконано
6	Затвердження пояснювальної записки роботи завідуючого кафедри		Виконано

Студент – бакалавр _____ (Валерій СТАРІКОВ)

Керівник роботи _____ (Максим ТКАЧЕНКО)

АНОТАЦІЯ

Випускна кваліфікаційна бакалаврська робота: 64 с., 25 рис., 1 додат., 7 джерел.

Тема: Візуалізація статичних і динамічних 3D об'єктів на основі векторної алгебри в графічних симуляціях.

Об’єкт дослідження: Використання векторної алгебри для графічних симуляціях та створення графічних симуляціях за допомогою рушія Unreal Engine 4.

Мета роботи: Дослідження проектування та розробки графічних симуляцій з практичної реалізацією у ігровій сфері та використанням векторної алгебри.

Предмет дослідження: Сучасний підхід до розробки графічних симуляцій та їх практична реалізація.

Результати дослідження: Були досліджені різні етапи да підходи сучасної розробки графічних симуляцій з практичною реалізацію у ігровій сфері та з використанням векторної алгебри.

Висновок

В даній роботі було проведено дослідження візуалізації різних типів об’єктів у графічних симуляціях на основі векторної алгебри та продемонстровано практичне застосування. Були використані сучасні інструменти для розробки ПЗ, яке створює графічну симуляцію з високою графікою, анімаціями, звуками, штучним інтелектом та елементами геймплею.

ГРАФІЧНІ СИМУЛЯЦІЇ, ВЕКТОРНАЯ АЛГЕБРА, ІГРОВИЙ РУШІЙ, UNREAL ENGINE 4, ВІДЕОГРА, ІГРОВІ МЕХАНІКІ, ГРАФІКА, АНІМАЦІЯ.

ANNOTATION

Final qualifying bachelor's thesis: 64 pages, 25 images, 1 appendix, 7 references.

Topic: Visualization of static and dynamic 3D objects based on vector algebra in graphical simulation.

Object of research: Using of vector algebra for graphical simulations and creating graphical simulations using the Unreal Engine 4.

Purpose: Research of design and development of graphic simulations with practical implementation in the game sphere and with using of vector algebra.

Subject of research: Modern approach to the development of graphic simulations and their practical implementation.

Research results: Different stages and approaches of modern development of graphic simulations with practical implementation in the game sphere using of vector algebra were investigated.

Conclusion

In this work, a study of the visualization of different types of objects in graphical simulations based on vector algebra was carried out and practical application was demonstrated. Modern tools have been used to develop software that creates graphical simulations with high graphics, animations, sounds, artificial intelligence and gameplay elements.

GRAPHIC SIMULATIONS, VECTOR ALGEBRA, GAME MOVEMENT, UNREAL ENGINE 4, VIDEO GAME, GAME MECHANICS, GRAPHICS, ANIMATION.

АННОТАЦИЯ

Выпускная квалификационная бакалаврская работа: 64 с., 25 рис., 1 дополнение., 7 источников.

Тема: Визуализация статических и динамических 3D объектов на основе векторной алгебры в графических симуляциях.

Объект исследования: Использование векторной алгебры для графических симуляциях и создания графических симуляциях с помощью двигателя Unreal Engine 4.

Цель работы: Исследования проектирования и разработки графических симуляций с практической реализацией в игровой сфере и использованием векторной алгебры.

Предмет исследования: Современный подход к разработке графических симуляций и их практическая реализация.

Результаты исследования: Были исследованы различные этапы да подходы современной разработки графических симуляций с практической реализации в игровой сфере и с использованием векторной алгебры.

Вывод

В данной работе было проведено исследование визуализации различных типов объектов в графических симуляциях на основе векторной алгебры и продемонстрировано практическое применение. Были использованы современные инструменты для разработки ПО, которое создает графическую симуляцию с высокой графикой, анимациями, звуками, искусственным интеллектом и элементами геймплея.

ГРАФИЧЕСКИЕ СИМУЛЯЦИИ, векторная АЛГЕБРА, игровой движок, UNREAL ENGINE 4, видеоигры, игровая механика, графика, анимация.

ЗМІСТ

РОЗДІЛ 1. ВСТУП 12
РОЗДІЛ 2. ТЕОРИТИЧНА ЧАСТИНА 15
2.1 Вектори 15
2.2 Матриці 17
2.3 Матриці камери та проекції 18
РОЗДІЛ 3. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ 22
3.1 Огляд графічної симуляції 22
3.1.2 Мапи 23
3.1.3 Звуки 24
3.1.4 Дельта час 24
3.2 Персонаж 25
3.2.1 Загальний опис 25
3.2.2 Зовнішній вигляд та анімації 26
3.2.3 C++ реалізація 27
3.2.4 Blueprint реалізація 28
3.2.5 Інтерфейс 29
3.2.6 Шкали здоров'я та стаміни 30
3.3 Моби 31
3.3.1 Зовнішній вигляд та анімація 31
3.3.2 C++ реалізація 32
3.3.3 Blueprint реалізація 33
3.3.4 Робота зі штучним інтелектом 34
3.3.5 Система бою 35
3.4 Ігрові механіки 36
3.4.1 Система взаємодії з об'єктами 36
3.4.2 Зброя 37
3.4.3 Плаваючі платформи 38
3.4.4 Таємні двері 39
3.4.5 Простір спавну 40
3.5 Сюжет 41

РОЗДІЛ 4. АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 42
4.1 Архітектура c++ класів UE4..... 42
4.2 Архітектура програмного забезпечення 44
ВИСНОВКИ 45
Література 46
Додаток А..... 47

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

UE4	-	Unreal Engine 4
ООП	-	об’єктивно орієнтоване програмування
ПЗ	-	програмне забезпечення
VS	-	Visual studio 2019

РОЗДІЛ 1. ВСТУП

Актуальність роботи

В наш час індустрія відеоігор дуже активно розвивається. Для цього використовують все новіші і новіші технології. Актуальність даної роботи полягає в тому, що вона збирає в одне ціле ці технології та демонструє їх практичний застосунок.

Мета і задачі дослідження

Метою дипломної візуалізація різних типів 3д об’єктів. В ході роботи буде проведено дослідження у використанні векторної алгебри для візуалізації. Також буде наведено практичне застосування опрацьованого матеріалу у вигляді графічної симуляції з елементами відеогри – дана є однією з найпопулярніших при роботі з графікою та гарно демонструю усі аспекти головної теми та використовую смужні області досліджень у графічних симуляціях.

Можна перерахувати основні математичні та фізичні поняття, які досліджуються у даній роботі:

- Вектори;
- Матриці;
- Лінійна трансформація матриць;
- Обертання об’єктів у 3Д просторі;
- Деякі геометричні примітиви;

Об’єктом дослідження

Об’єктом дослідження є література по використанню векторної алгебри для графічних симуляціях та література по створенню графічних симуляціях за допомогою рушія Unreal Engine 4.

Предметом дослідження

Сучасний підхід до розборки графічних симуляцій та їх графічна реалізація.

Методи дослідження

Основними мовами програмування було обрано C++ та Blueprint. Даний вибір був зумовлений простотою та зручністю використання даних мов для цілей даної роботи.

C++ являє собою усім відому та зручну мову для створення графіки та відеоігор. Її переваги перед іншими полягають в тому, що вона має ефективну роботу за пом'яють, що дуже важливо для відеоігор. Так як для графічних симуляцій прийнято було використовувати об'єктно-орієнтоване програмування (ООП) то нам дуже допоможуть засоби цієї мові: посилання, вказівники, множинне спадкування класів, зручне перевантаження операторів, тощо.

Bluerpint – це система візуального скриптинга, що представляє собою візуальний інтерфейс для створення елементів геймплея. Дана мова використовується в ігровому движку Unreal Engine. Вона дозволяє використовувати майже повний потенціал програмування та є дуже ефективною у поєднанні з C++.

В ПЗ буде наведено безліч прикладів використання цих мов разом. Наприклад, можна створювати змінні чи функції, які використовуються, як у C++ так і у Blueprint. Такий підхід дозволить нам змінювати рушій та створювати власні меню та панелі у його інтерфейсі.

Для отримання змогу використовувати повний потенціал UE4 використовуючи Visual Studio 2019(VS). Дане середовище розробки призначене для зборки C++ коду.

Практичне значення одержаних результатів

Програмне забезпечення (ПЗ), яке було створено для дипломної роботи, використовує графічний рушій Unreal Engine 4 (UE4). Даний рушій, на відміну від багатьох інших засобів для візуалізації, дозволяє нам самим запрограмувати правила рендерінгу, освітлення, руху, фізики, тощо. Тому цей рушій і було обрано для роботи.

За допомогою теоретичних знань векторної алгебри та рушія UE4 було створено ПЗ, яке має такі ключові характеристики:

- Повністю анімований керований 3д персонаж;
- велика карта, яка відкрита для досліджень;
- супротивники, які також повністю анімовано та керуються штучним інтелектом;
- система бою;
- система взаємодії персонажа з навколишніми об’єктами;
- графічний інтерфейс користувача.

Графічна складова також включає в себе роботу з різними типами текстур та динамітних анімацій, наприклад, диму, вогню, тощо; та роботу з анімацією персонажа.

Симуляція, яка була створена має елементи різних ігрових жанрів: від аркади до рпг. Сама гра має високу графіку, реалістичні анімації, фізику та багато різних ігрових механік.

Особистий внесок студента

У роботі були поєднані різні підходи до розборки графічних симуляцій: від векторної алгебри до поєднання мов програмування та їх спільного використання. Було продемонстровано практичну реалізацію у вигляді відеогри, яка має зайняти своє місце на ринку ігор.

Структура та обсяг роботи даних

Робота викладена на 68 сторінках друкованого тексту. Робота містить 12 таблиць, 32 рисунки та 3 додатки, обсягом 12 стор.

РОЗДІЛ 2. ТЕОРИТИЧНА ЧАСТИНА

2.1 Вектори

В даній роботі буде розглянута та частина векторної алгебри, яка безпосередньою використовуються у графічних симуляціях, а саме для модулювання усіх видів руху та для зручного використання лінійної кінематики.

Перше, що буде потрібно для роботи – це вектор. Переважно, в роботі використовуються тільки 3х вимірні вектори. В роботі використовуються такі операції над векторами: отримання розміру, нормалізація, перевірка на нормалізацію, векторний добуток, скалярний добуток, множення на скаляр, додавання та віднімання векторів [1] [2].

Для того, щоб реалізувати ці операції у програмування було створено спеціальний клас FVector. Формули потрібних операцій виглядають так:

- **вектор(V):**

$$V = (x;\ y;\ z) \tag{2.1.1}$$

- **отримання розміру (Size):**

$$Size = sqrt(x^2 + y^2 + z^2) \tag{2.1.2}$$

- **нормалізація:**

$$V_n = (x^2/Size + y^2/Size + z^2/Size) \tag{2.1.3}$$

- **перевірка на нормалізацію:**

$$IsN = (x^2/Size + y^2/Size + z^2/Size) \tag{2.1.4}$$

- **скалярний добуток:**

$$DotProduct = (a_x * b_x + a_y * b_y + a_z * b_{z;}) \quad (2.1.5)$$

$$DotProduct = (Size_a * Size_b * \cos(alpha)) \quad (2.1.6)$$

- **векторний добуток:**

$$CrossProduct = (Size_a * Size_b * sin(alpha)) \quad (2.1.7)$$

- **віднімання векторів:**

$$V_{a-b} = (a_x - b_x ; a_y - b_y; a_z - b_{z;}) \quad (2.1.8)$$

- **додавання векторів:**

$$V_{a+b} = (a_x + b_x ; a_y + b_y; a_z + b_{z;}) \quad (2.1.9)$$

Даний клас гарантує зручність у написанні подальшого коду: усі потрібні операції представлені у вигляді інтуїтивно-зрозумілих функції. Треба звернути увагу на те, що деякі оголошення не є стандартними функціями, а є перевантаженими операторами. Мова C++ дозволяє це зробити досить зручно і тепер можна використовувати оператори для операцій з векторами, що є дуже зручним та компактним.

2.2 Матриці

Матриці займають важливе місце для створення графіка, адже саме завдяки їх можна легко реалізувати переміщення, обертання та масштабування графічних об’єктів.

Для зручного використання матриць було старено клас FPlane, який дозволить далі створювати матрицю з трьома площинами.

Тепер можемо перейти до операцій з матрицями. Для програмного забезпечення, яке було створено достатньо трьох операцій [1] [2]:

- **Переміщення:**

$$T(a,b,c) = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{matrix} \quad (2.2.1)$$

- **Масштабування:**

$$T(S_x, S_y, S_z) = \begin{matrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \quad (2.2.2)$$

- **Поворот:**

$$R_x = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & -\sin(a) & 0 \\ 0 & \sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \quad (2.2.3)$$

$$R_y = \begin{matrix} \cos(a) & 0 & \sin(a) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(a) & 0 & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \quad (2.2.4)$$

$$R_z = \begin{matrix} \cos(a) & -\sin(a) & 0 & 0 \\ \sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \quad (2.2.5)$$

2.3 Матриці камери та проекції

Матриця камера утворюється за допомогою трьох параметрів: положення камери; цільове положення, на яке дивиться камера та направлення вгору. [1]

Тому для початку потрібно обчислити 4 різні вектори»

$$k = \frac{(target - camera)}{|target - camera|} \quad (2.3.1)$$

$$i = \frac{(up * k)}{|up * k|} \quad (2.3.2)$$

$$j = \frac{(k * i)}{|k * i|} \quad (2.3.3)$$

$$t = (-i * camera;\ -j * camera; -k * camera) \quad (2.3.4)$$

Виходячи з цих векторів матриця камера буде мати вигляд:

$$C = \begin{matrix} i_x & j_x & k_x & 0 \\ i_y & j_y & k_y & 0 \\ i_z & j_z & k_z & 0 \\ t_x & t_y & t_z & 1 \end{matrix} \quad (2.3.5)$$

Матриця проекції визначає як 3Д об’єктиперетворюються на 2д, які відображаються на екрані. В ортографічній проекції предмети, розташовані далі від камери, мають такі ж розміри, що й предмети, розташовані ближче до камери. Це означає, що гравці не зможуть сприймати, розташовані об'єкти ближче або далі від камери. Тому білшість ігор використовують перспективну проекцію.

У перспективній проекції об'єкти, віддалені від камери, менші, ніж ближчі. Таким чином, гравці сприймають глибину сцени.

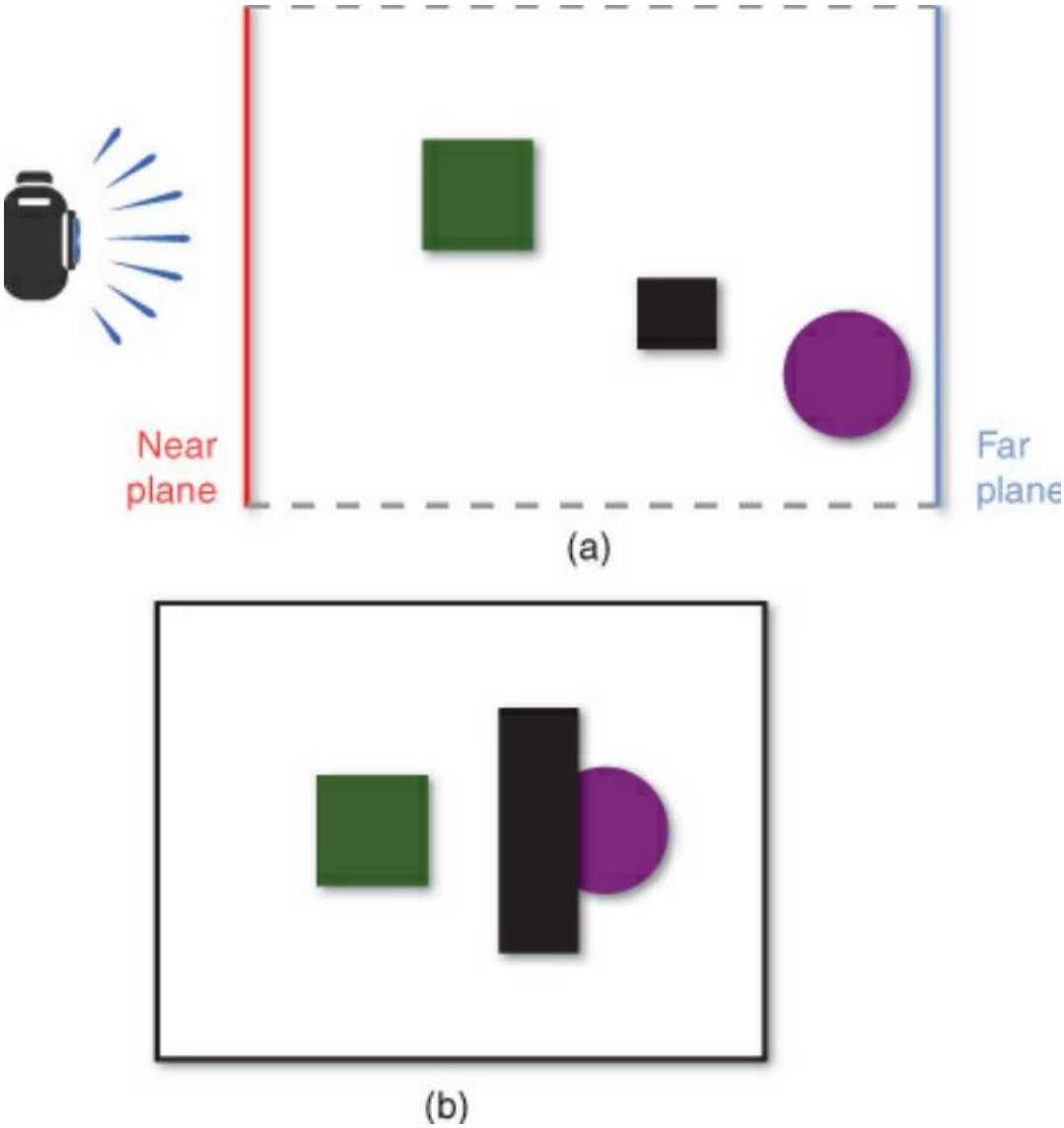

Рисунок 2.3.1 (a) Вид орфографічної проекції зверху вниз і (b) отримане 2D-зображення на екрані [1]

Кожна з цих проекцій має близьку та далеку площину. Близький літак, як правило, знаходиться дуже близько до камери. На екрані не видно нічого між камерою та ближньою площиною. Ось чому в іграх предмети частково зникають, якщо камера наближається до них. Так само далекий літак знаходиться далеко від камери, і нічого повз нього не видно. Ігри іноді дозволяють гравцеві зменшити "дистанцію розіграшу" для покращення продуктивності. Це часто просто питання затягування в дальню площину.

Матриця орфографічної проекції має чотири параметри: ширину виду, висоту виду, відстань до ближньої площини та відстань до дальньої

площини. Враховуючи ці параметри, орфографічна матриця проекції виглядає так:

$$O = \begin{matrix} 2/w & 0 & 0 & 0 \\ 0 & 2/h & 0 & 0 \\ 0 & 0 & 1/(f-n) & 0 \\ 0 & 0 & n/(n-f) & 1 \end{matrix} \quad (2.2.2)$$

В UE4 вже існує клас, який називається `FMatrix ProjectionMatrix;` Тому можна його використовувати для налаштування камери, або змінювати для написання своєї камери.

РОЗДІЛ 3. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Огляд графічної симуляції

В результаті виконання дипломної роботи була створена графічна симуляція з елементами геймплею. Користувач може керувати повністю анімованим персонажем, досліджувати карту, збирати очки та перемагати супротивників. Загальний вигляд симуляції:

Рис. 3.1.1 Загальний вигляд симуляції

Та вигляд від третього лиця, граючи за персонажа:

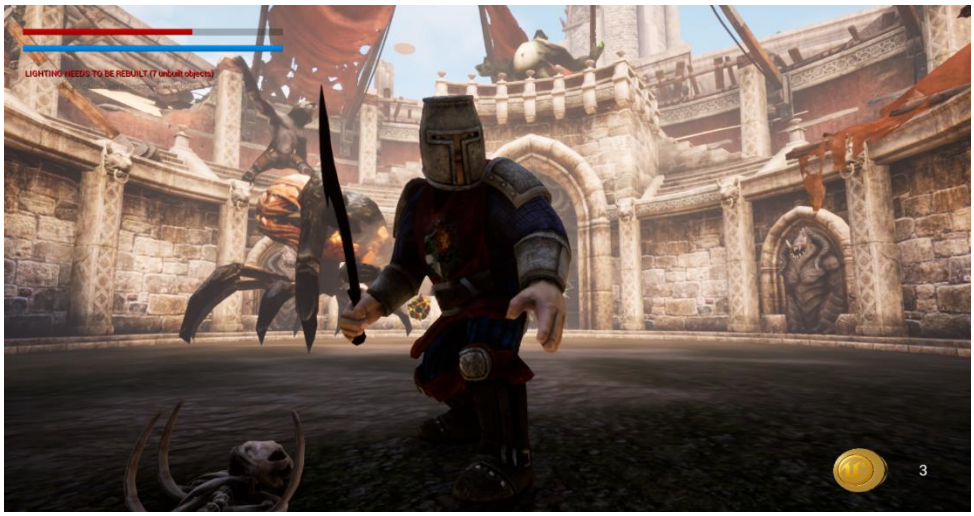

Рис. 3.1.2 Загальний вигляд симуляції (вид за персонажа)

3.1.2 Мапи

Дана симуляція має 2 мами. Перша графічна мапа називається – «Храм Сонця» [8] [9]. Це невелика мапа, яка являє собою споруду, де є два зали та одна велика тераса. Освітлення, яке присутнє на мапі виправдовую її назву:

Рис. 3.1.2.1 Перша мапа

Дана мапа включає в себе понад 500 елементів, як динамічних, так і статичних об’єктів: вікна, колони, освітлення, моби, факели, автодвері, тощо. Більшість об’єктів мають різні типи колізій колізії, що дозволяє персонажу та мобам взаємодіяти з ними. Більшість колізій представлені геометричними примітивами: паралелепіпед, сфера, тощо.

Друга мапа називається «Руїни». Вона більша за розміром та має специфічний рельєф. Вона включає в себе зруйновані фортеці, пам’ятники, гори, тощо. Для потрапляння з першої мапи на другу був реалізований телепорт [8] [9]. Більше детально про це буде написано в розділі «Ігрові механіки».

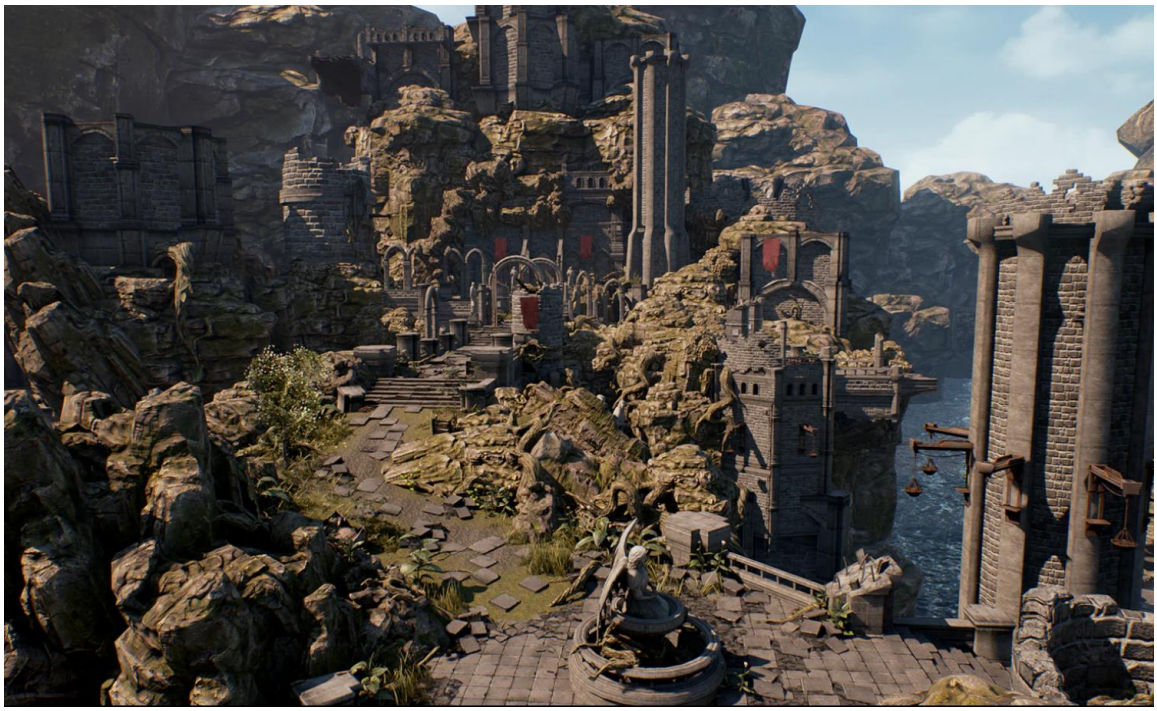

Рис. 3.1.2.2 Друга мапа

3.1.3 Звуки

Симуляція має звукове супроводження. UE4 дозволяє завантажувати свої mp3 файли та перетворювати їх на ігрові звуки – cue файли:

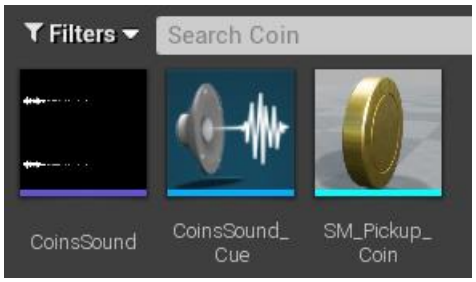

Рис. 3.1.3.1 Звуки

Таким чином були підібрані різні звуки для різних ігрових механік: вибух бомб, взаємодія з монетою, атака зброї, тощо.

3.1.4 Дельта час

Концепція дельта часу відіграю важливу роль у графічному відображенні динамічних об’єктів. Усі типи графічних переміщень

виконуються кожен кадр. Але частота кадрів (fps) є різною на різних пристроях.

Саме тому була придумана концепція дельта часу. Дельта час – це час, який пройшов з моменту відображення останнього кадру. Перемноживши швидкість переміщення на дельта час отримаємо саме ту величину, яка буде відповідати встановленій швидкості у реальному житті. [3] [14] [15]

Наприклад, персонаж рухається зі швидкість 10 м/с. Якщо fps дорівнює 30, то дельта час дорівнює 0,03. Перемножимо швидкість 10 м/с на дельта час 0.03 і отримаємо 0.3 м/кадр. Тепер перемножимо 0.3 м/кадр на fps і отримаємо 10 м/с. Якщо взяти 60 fps дельта тайм буде 0.15 і переміщення м/кадр буде частішим, але швидкість за секунду все одно буде 10 м/с.

Концепція дельта часу допомагає розробникам роботи ігри, яке не залежать від частості кадрів окремих пристроїв. В даній роботі також використовується ця концепція.

3.2 Персонаж

3.2.1 Загальний опис

Даний персонаж є повністю анімований, він має такі типи руху: біг, стрибок, швидкій біг та атака. Основними атрибутами є шкала здоров’я та стаміна – дані атрибути відображаються в інтерфейсі.

Для взаємодії (колізії) з різними типами об’єктів була створена різна реалізація. Наприклад, якщо об’єкт, з я ким взаємодія персонаж – є зброєю, то він має вибір – взяти цю зброю чи ні. Якщо ж це бомба, то при колізії бомба вибухає та відіймає здоров’я персонажа. При швидкому бігу відіймається стаміна; якщо стаміна буде дорівнювати нулю, то персонаж певний час не зможе швидко бігати. Стаміна регенерується через певний час.

3.2.2 Зовнішній вигляд та анімації

Як вже було зазначено: в симуляцій прикутій повністю анімований 3Д персонаж. Так виглядає його модель [16]:

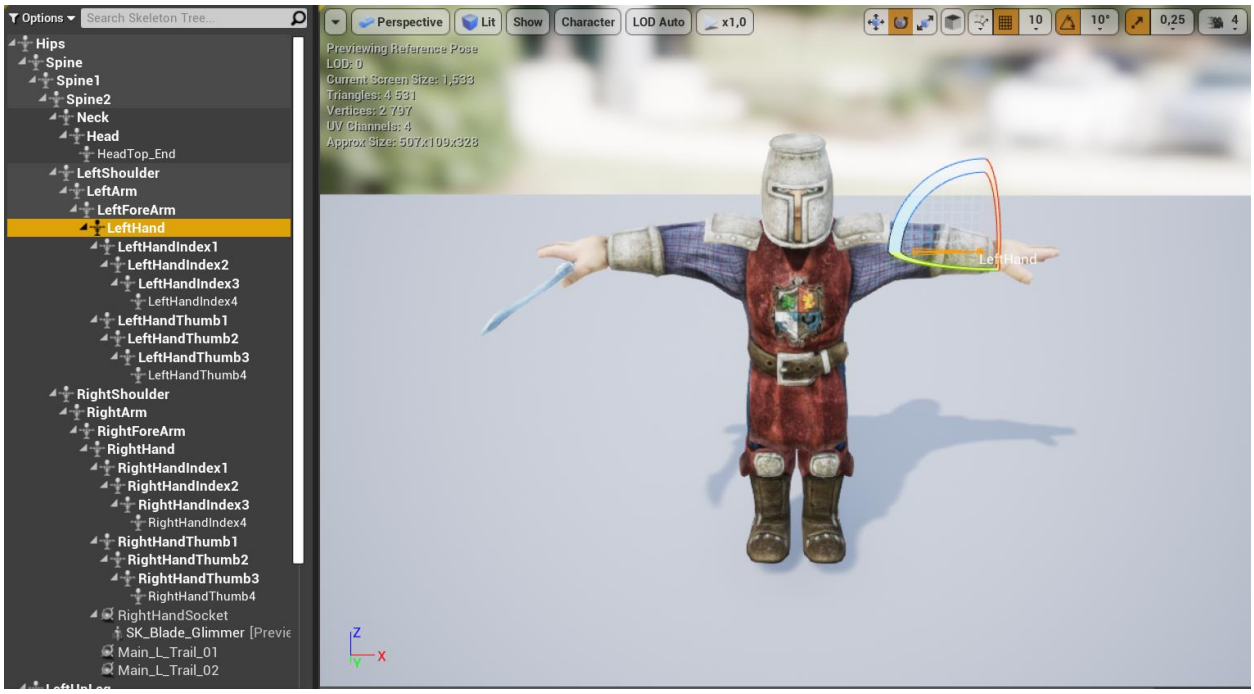

Рис. 3.2.2.1 Модель персонажа

Персонаж являє собою об’єднання багатьох слоїв: модель (текстура), скелет, анімація, фізика, код c++ та скриптинг Blueprint.

На рисунку 3.3.1 зображено також меню скелета персонажа. Як бачимо, зліва обрано кістку з назвою LeftHand і вона відображається на моделі персонажа. Кожна кістка може набувати різних положень, що дозволяє ефективно працювати з різними анімаціями. Усього налічується біля 50 кісток.

Тепер розглянемо анімації, які має цей персонаж. Усього налічується 15 анімацій. Деякі з них об’єднані у спеціальні блоки. Наприклад: анімація спокою, ходьби та бігу об’єднані у спеціальний блок. Якщо персонаж має нульову швидкість, то відіграється анімація спокою, але якщо він починає рух, то анімація спокою поступово переходить у ходьбу, а потім і у біг. Даний підхід дозволяє отримувати плавний перехід між анімаціями.

Рис. 3.2.2.2 Анімація замаху для атаки

3.2.3 C++ реалізація

Весь код двох файлів Main.h та Main.cpp, які є реалізацію персонажа на мові c++, буде наведено у додатку. В цьому розділі буде проведено аналіз основних частин даної реалізації.

У файлі Main.h запрограмовані оголошення функції та змінних. Майже кожна функція чи змінна має спеціальні макроси: `UPROPERTY()` та `UFUNCTION()`. Дані макроси – допоможуть працювати з c++ об'єктами у інтерфейсі редакторі та у Blueprint. [3] [4]

Розглянемо основні функції та групи функцій класу персонажу:

- Перша функція – це конструктор класу, там були налаштовані загальні параметри персонажа та також створена камера.
- Друга функція виглядає так: `void AMain::Tick(float DeltaTime);` Це функція для оновлення персонажу кожного кадру; як бачимо – тут присутня змінна з назвою DeltaTime – саме тут і можна реалізувати концепцію Дельта часу. В цій функції реалізована робота з рухом персонажа.

- Третя функція - `void AMain::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)` – це функція для налаштування керування персонажем за допомогою клавіатури.
- Четверта група функцій - `void AMain::Move(float Value)`, `void AMain::Jump()`, `void AMain::TurnAtRate(float Rate)` - ця функції використовується для різних типів руху персонажа. Вони використовується у функції `SetupPlayerInputComponent`. Саме тут і можна використати клас FMatrix, який було описано у теоретичній частині.
- П’ята група функцій - `void AMain::Attack()`, `void AMain::UpdateCombatTarget()`, `void AMain::SetEquippedWeapon(AWeapon* WeaponToSet)` – це функції, які відповідають за бойові дії персонажа.

Друга частина c++ реалізація знаходиться у файлах MainAnimInstance.h та MainAnimInstance.cpp. Даний клас зберігає в собі швидкість руху персонажа, ця змінна буде використовуватися діла у класі Blueprint, який будет створено на основі MainAnimInstance.

3.2.4 Blueprint реалізація

Перша частина реалізації на Blueprint знаходиться у файлі Main_BP. Друга частина реалізації на Blueprint заходиться у файлі, якій виповідає за анімацію персонажа MainAnim_BP, так як саме для цього і зручно використовувати Blueprint. [6]

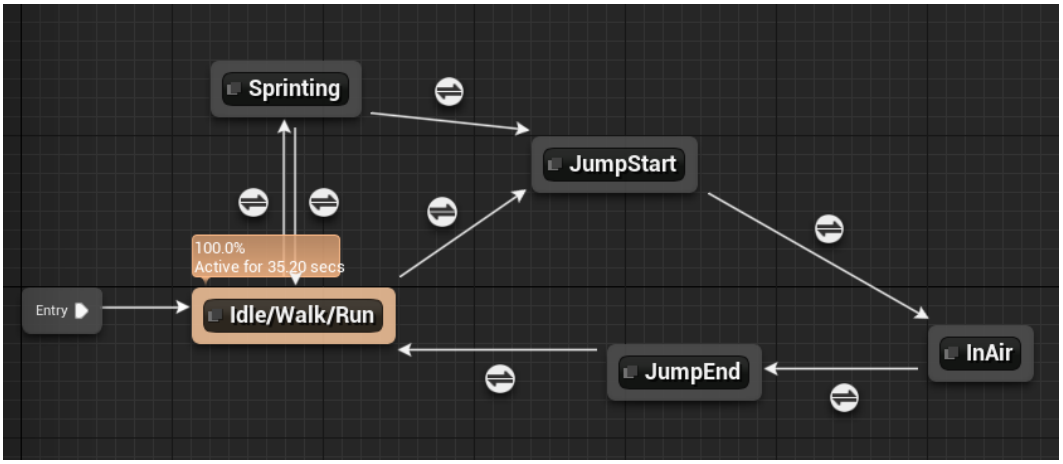

Рис. 3.2.4.1 Реалізація робота анімацій

На даному рисунку можемо побачити блоки, які являють собою анімації та зв’язки, які встановлені між ними. Перехід від однієї анімації до іншої виконується при виконані певних умов. Наприклад, щоб перейти до анімації JumpStart змінна bIsInAir у класі MainAnimInstance має набути значення true. [7] [8]

3.2.5 Інтерфейс

Інтерфейс гри складається з 3х компонентів: шкала здоров’я – вона позначена червоним кольором на рисунку; шкала стаміна – синій колір та кількість монет, які персонаж може збирати на протязі гри.

В лівому верхньому куті екрану знаходяться шкали здоров’я та стаміна; у правому нижньому – кількість монет. Графічна реалізація зроблена за допомогою інструментів UE4 для роботи з інтерфейсом [5]:

Рис. 3.2.5.1 Інтерфейс 1

Рис. 3.2.5.2 Інтерфейс 2

3.2.6 Шкали здоров’я та стаміни

Один з прикладів ефективної взаємодії c++ та Blueprint є реалізація шкал здоров’я та стаміни. Так виглядає код на Blueprint:

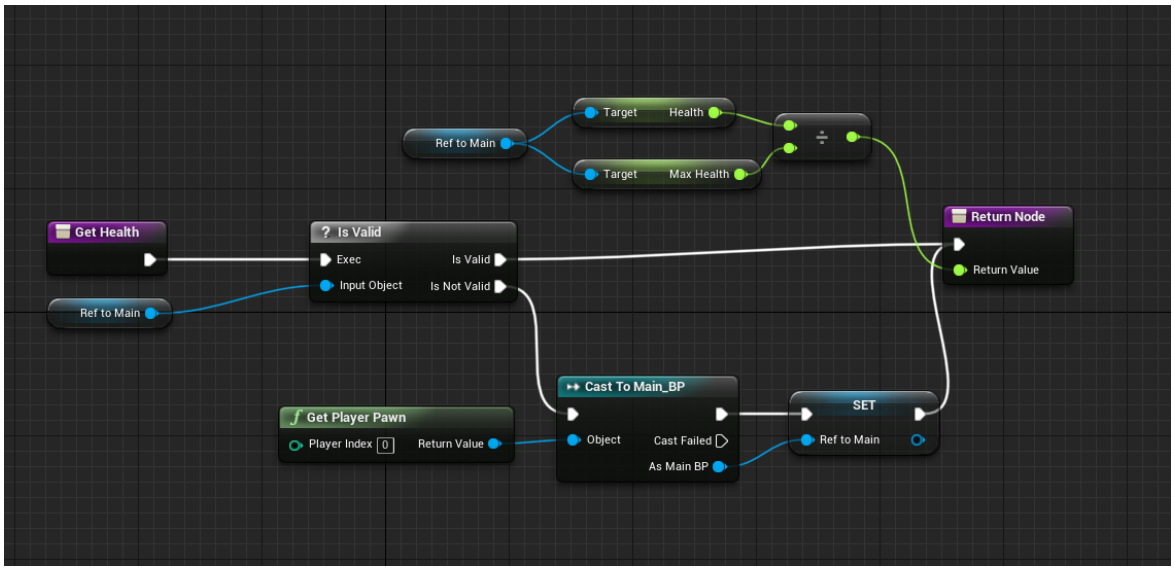

Рис. 3.2.6.1 Blueprint шкала здоров’я

Як бачимо в цьому файлі було використано посилання на клас Main (Ref to Main). Таким чином отримується доступ до певних полів та методів класа у c++. Для шкали здоров’я було отримано поля Health та MaxHealth. Потім виконується ділення та отримується процент, який становить Health від MaxHealth. Такі операції дозволяють нам зафарбовувати червоним відповідну частину шкали здоров’я.

Для стаміни було створено схожий код, але складніший. В ньому так само використовується посилання на клас Main, але отримується поле StaminaStatus. Складність реалізації станіни полягає в тому, що колір шкали змінюється відповідно до кількості стаміни. Так виглядає реалізація:

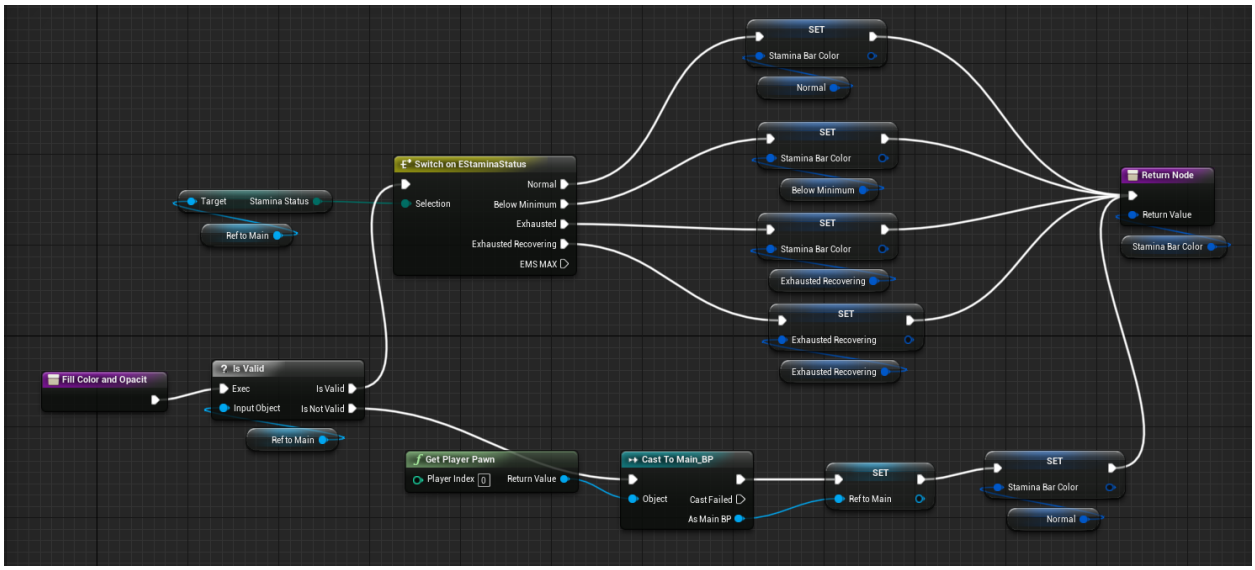

Рис. 3.2.6.2 Blueprint шкала стаміни

3.3 Моби

3.3.1 Зовнішній вигляд та анімація

Моби – це ворожі персонажі під управління штучного інтелекту.

Усього було створено 2 типи мобів [8] [9]:

Павук:

Рис. 3.3.1.1 Графічне предствлення павука

Лицарі:

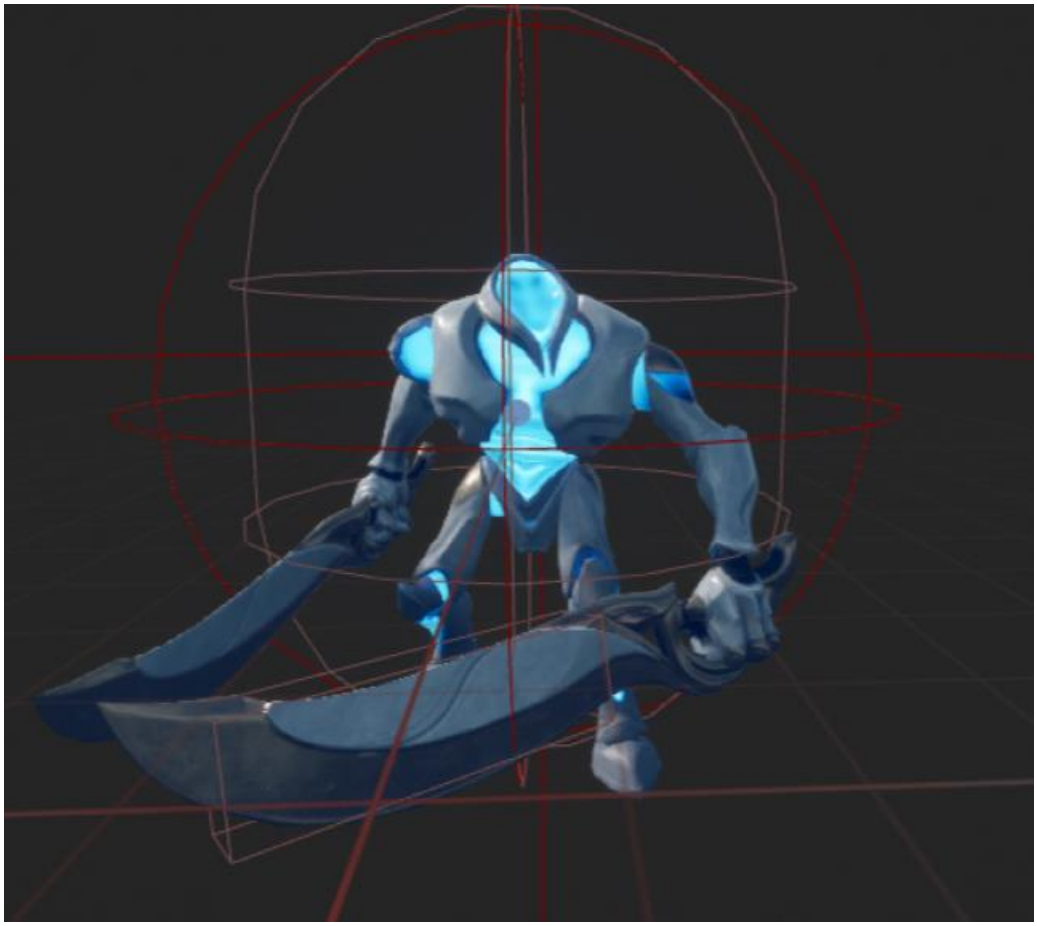

Рис. 3.3.1.2 Графічне предствлення лицаря

На рисунку зображений один з типів лицарів. Всього їх існую 3. Кожен тип має свої властивості: текстуру, здоров’я, швидкість, урон.

3.3.2 C++ реалізація

Частина поведінки мобів запрограмована на c++. Файли Enemy.h та Enemy.cpp містять першу частину реалізації на c++. [10]

Концепція, яка була використана при реалізації мобів полягало в наступному: створюється c++ клас, який містить основні параметри, які має мати моб. Потім від цього класу створюються 3 окремі Blueprint класи, які может змінювати ті параметри, які були запрограмовані на c++, наприклад,

урон, швидкість, тощо. Даний підхід знову демонструє ефективність використання c++ та Blueprint разом для розробки.

Розглянемо основні функції та групи функцій класу ворога(моба):

- Перша функція – це конструктор класу, там були налаштовані загальні параметри моба. Наприклад: сферу колізій, атаки чи початку нападу. Розміри сфер для кожного моба будуть змінені у Blueprint. Сфера колізії створена для пришвидшення роботи програми, так як прорахунок кожної кінцівки чи складок одежі моба буде дуже затратним.
- Друга функція: `void AEnemy::BeginPlay()`. Тут програмується поведінка при зіткнені з різними типам об'єктів.
- Третя група функцій: `AEnemy::AgroSphereOnOverlapBegin()`, `void AEnemy::AgroSphereOnOverlapEnd()`, `void AEnemy::CombatSphereOnOverlapBegin()`, `void AEnemy::CombatSphereOnOverlapEnd()` – дана група функцій задає поведінку при взаємодії персонажа та моба.
- П'ята група функцій – це реалізації бойової системи: рух до цілі, атака, урон та смерть: `void AEnemy::Attack()`, `void AEnemy::Attack()`, `float AEnemy::TakeDamage()`,`void AEnemy::Die()`.

Робота з анімацією мобів дуже схожа на роботу з анімацією персонажа, тому писати про це другий раз немає сенсу [4] [5].

3.3.3 Blueprint реалізація

Тепер перейдемо до Blueprint реалізації. Так виглядає клас:

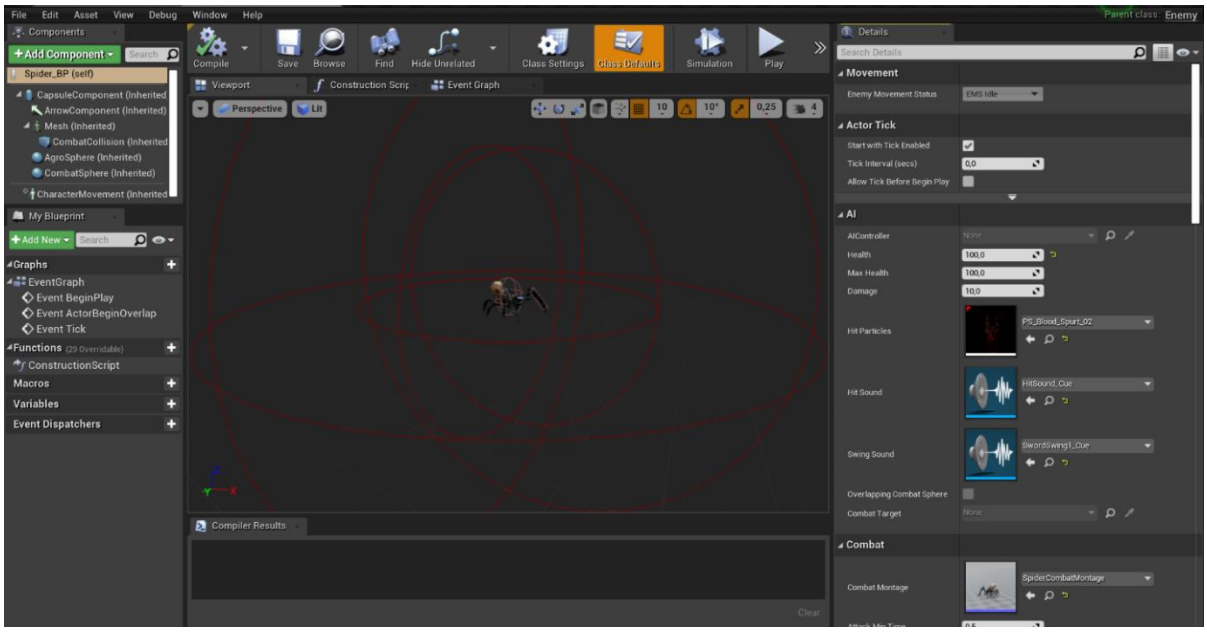

Рис. 3.3.3.1 Blueprint релалізація мобів

Як можна побачити на панелі Components є усі сфери взаємодії, які були запрограмовані, більш того тепер можна змінювати їх радіус, що дозволяє на створювати персонажів з різною поведінкою.

На панелі Details можемо побачити різні поля, які представляють собою змінні c++ класу. Тут можна обрати зовнішній вигляд, розмір моба, налаштувати звуки, тощо.

3.3.4 Робота зі штучним інтелектом

Для програмування поведінки моба було також використано підхід роботи зі штучним інтелектом у UE4. Суть цього підходу полягає в тому, що UE4 надає там готовий штучний інтелект з реалізованим алгоритмом пошуку напоротого шляху до цілі. [7]

У c++ класі у функції `void AEnemy::MoveToTarget()` строюється змінна `FNavPathSharedPtr NavPath;` яка і буде шляхом до цілі, яка буду встановлена. У випадку даної графічної симуляції – це персонаж.

Для того, щоб штучний інтелект працював у самому редакторі строюється об’єкт, який називається Nav Mesh – це площина зеленого кольору, яка покриває простір, по якому може рухатися моб. Виглядає це так:

Рис. 3.3.4.1 Навігація штучного інтелекту

Звичайно, при запуску самої симуляції ця зелена площина зникає. Її можна побачити лише налаштовуючи симуляцію.

Тепер розглянемо ще одну частину штучного інтелекту, яка була реалізована. Це сфери, які вже частково було написано у c++ реалізації. Для поведінки моба створено 3 сфери. Сфера зору – якщо персонаж потрапляє у неї, то моб починає рухатися до нього, відповідно до алгоритму штучного інтелекту. Сфера атаки – якщо персонаж потрапляє у цю сферу це означає, що моб його наздогнав та може атакувати. Сфера колізій – сфера, яка є фізичним представленням моба, вона використовуються штучним інтелектом для прорахунку колізій.

3.3.5 Система бою

Систему бою реалізована у класі персонажа та у класі мобів. Розглянемо спочатку реалізацію с боку персонажа. Ця система полягає в

тому, шо ціль автоматично визначається, в залежності від того яка з цілей ближча. Це значно спрощує управління.

З боку мобів була створена додаткова колізія, у вигляді паралелепіпеда, який прив’язується до кінцівки(у випадку павука) чи до зброї(у випадку лицарів). Дана колізія створена для роботи з атакою. Як моби так і персонаж можуть наносити один одному урон, якщо ж здоров’я одного з них стає менше нуля, то програється анімація загибелі.

Рис. 3.3.5.1 Скріншот бою

3.4 Ігрові механіки

3.4.1 Система взаємодії з об’єктами

Об’єкти, з якими може взаємодіяти персонаж представлені c++ класом Item. Даний клас містить дві основні віртуальні функції, які будуть перезаписані у класах нащадках. Ці дві функції створені для того, щоб програмувати поведінку об’єктів, з якими взаємодіє персонаж.

Перший клас нащадок Item – це бомба, поведінка, яка представлена c++ класом Explosive. Цей клас зберігає указник на клас персонажу Main. Даний указник використовується для зменшення здоров’я персонажа при взаємодії з

бомбою – коли таке трапляється, то бомба вибухає та зникає з симуляції. Така поведінка запрограмована у двох функціях, які Explosive наслідує від Item.

Другий клас нащадок Item – це монета, яка представлена c++ класом Pickup. Архітектора поведінки аналогічна з класом Explosive. Сама поведінка являє собою то, що при взаємодії з монетою кількість монет в показнику в інтерфейсі збільшується.

3.4.2 Зброя

Реалізація зброї представлена c++ класом Weapon. Даний клас є нащадком Item, тому у ньому були перевизначені функції, які виповідають за взаємодію з персонажем. В результаті, якщо персонаж піддійте до зброї та натисне ліву кнопку миші – зброя буде екіпірована. Функція Equip(), яка відповідає за екіпірування персонажа поточної зброї.

Важливим елементом є те, що після екіпірування нової зброї – стара зброя, яка була у руці персонажа, має зникнути з гри. Саме тут ми і можемо використати роботу з пам’яттю у c++ та через вказівник видалити більше не потрібний об’єкт. Також для зброї була реалізована анімація частинок. Тепер на кожну зброї можна помістити вогонь, дим, лід чи інші анімовані частинки.

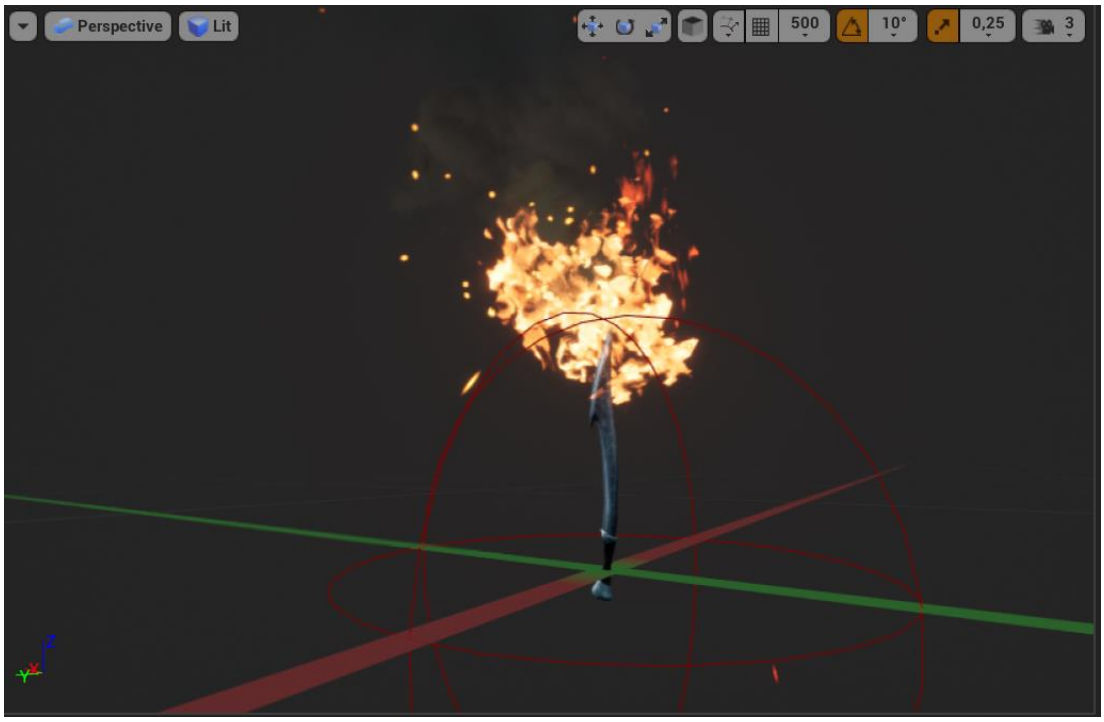

Рис. 3.4.2.1 Графічне представлення зброї

3.4.3 Плаваючі платформи

Рис. 3.4.3.1 Графічне представлення плаваючих платформ

Плаваючі платформи – це ігрова механіка, яка створена для різноманітності геймплею. Дана механіка являє собою платформи, які плавно рухаються у просторі та можуть перевозити гравця з одного місяці на інше.

Для цього була використана інтерполяція та обмін векторів місцями. Таким чином отримуємо циклічну механіку, яка робить геймплей значно цікавішим.

3.4.4 Таємні двері

Таємні двері – це ігрова механіка, яка відображає вплив гравця на ігровий простір. Дана механіка являє собою 2 об’єкта. Перший – перемикач, якщо персонаж наступає на нього, то другий об’єкт – двері – підіймуться і дозволять гравцю пройти далі.

Дана механіка реалізована на c++, де запрограмовані функції взаємодії з механікою та на Blueprint, де налаштовується плавне підіймання дверей за допомогою спеціальної системи таймерів UE4.

Рис. 3.4.4.1 Графічне представлення таємних дверей

Після того як персонаж відійшов від перемикача – двері через 2 секунди знову опустяться.

3.4.5 Простір спавну

Простір спавну – це ще одна ігрова механіка, яка має на меті спавн конкретних мобів в тій області, яка була позначена на карті. Простір спавну є паралелепіпедом:

Рис. 3.4.5.1 Графічне представлення простору спавну

Основна реалізація даної механіки знаходиться у c++ класі SpawnVolume. Така реалізація дозволяє вибирати вже у інтерфейсі UE4(нам не потрібно змінювати код) тип мобів, які будуть спавнитися у конкретному просторі спавну.

Далі у Blueprint налаштовуються ефекти-анімації, які будуть програватися під час спавну моба. Для цього функція спавну у c++ була зазначена як `BlueprintNativeEvent` – така реалізація дозволяє комбінувати поведінку функції у c++ та у Blueprint.

3.5 Сюжет

В створеній симуляції також присутній і сюжет. Його суть в тому, що гравець має здолати всіх мобів на першій мапі, знайти таємні двері та через плаваючі платформи дістатися до телепорту, який переведе гравця на іншу маму.

На другій мапі моби стають більше небезпечними: вони швидше рухаються та мають більший урон. А в кінці другої мапи на гравця чекає бос, який представлений великим павуком, який дуже повільно рухається, але має дуже високий урон. Завдання гравця – подолати боса та забрати нову зброю, яку бос охороняє.

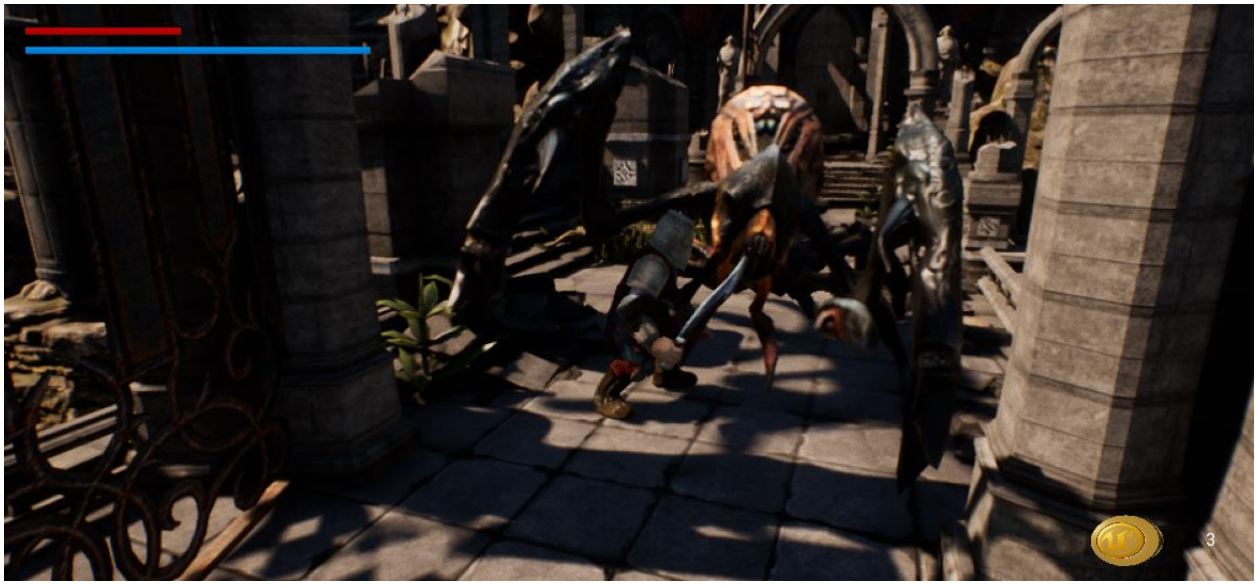

Рис. 3.5.1 Фінальний бос

РОЗДІЛ 4. АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Архітектура c++ класів UE4

UE4 маж свою систему c++ класів, яка була також використана при створені ПЗ. [3] [4] [5]

- Перший клас, від якого наслідуються всі інші класи – це Object. Його функції полягають лише у збереженні інформації. Він не може буду поміщений до графічної симуляції.
- Другий клас – це Actor. Наслідується від Object. Він може бути поміщений до симуляціях та має графічне представлення. Зазвичай, акторами виступають різні ігрові механіки, типу плаваючих платформ чи таємних дверей.
- Третій клас – Pawn. Наслідується від Actor. Даний клас має усі функції актора, але він також може керуватися гравцем.
- Четвертий клас – Character. Наслідується від Pawn та має компонент руху.

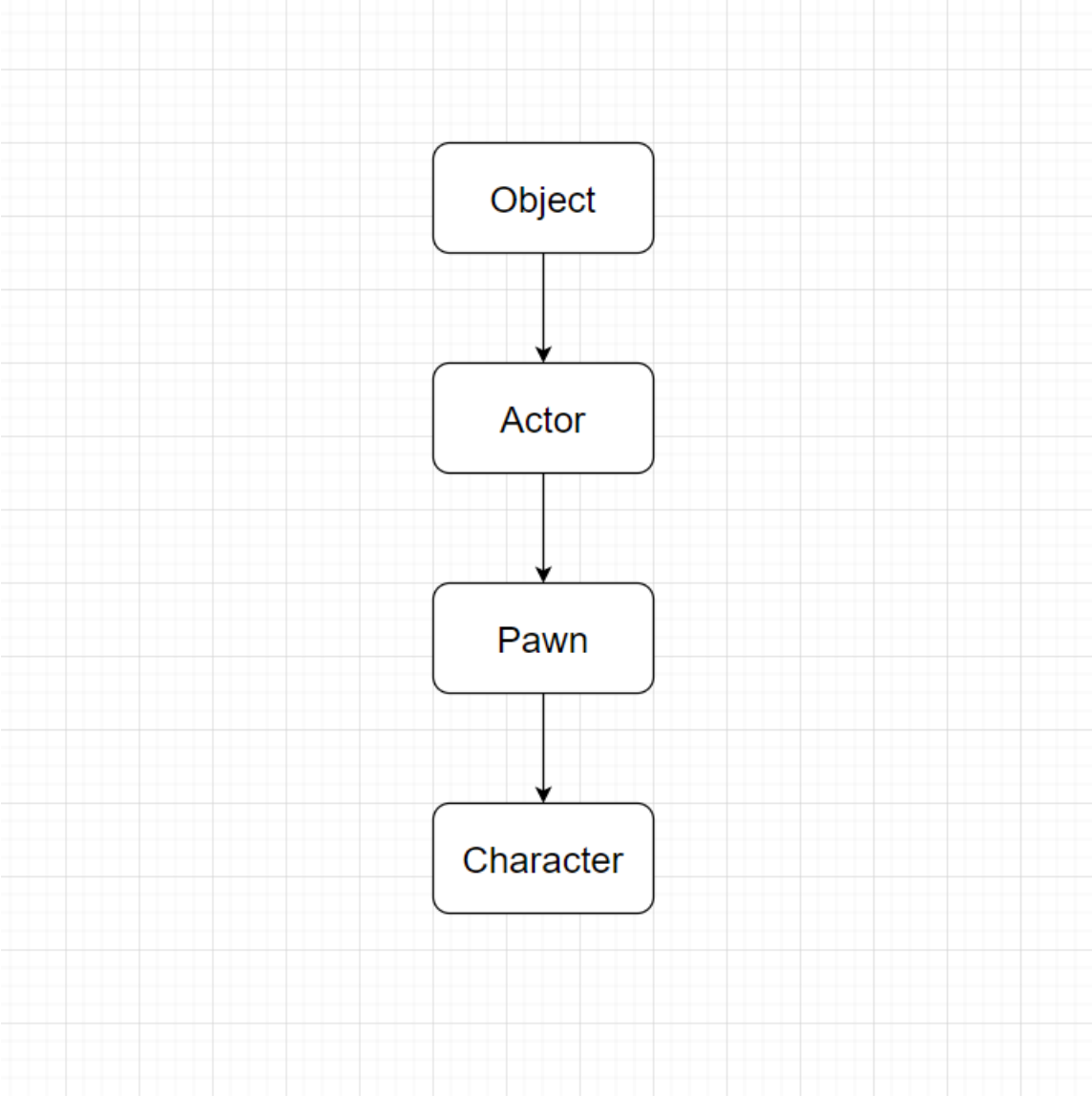

Рис. 4.1.1 Система c++ класів UE4

4.2 Архітектура програмного забезпечення

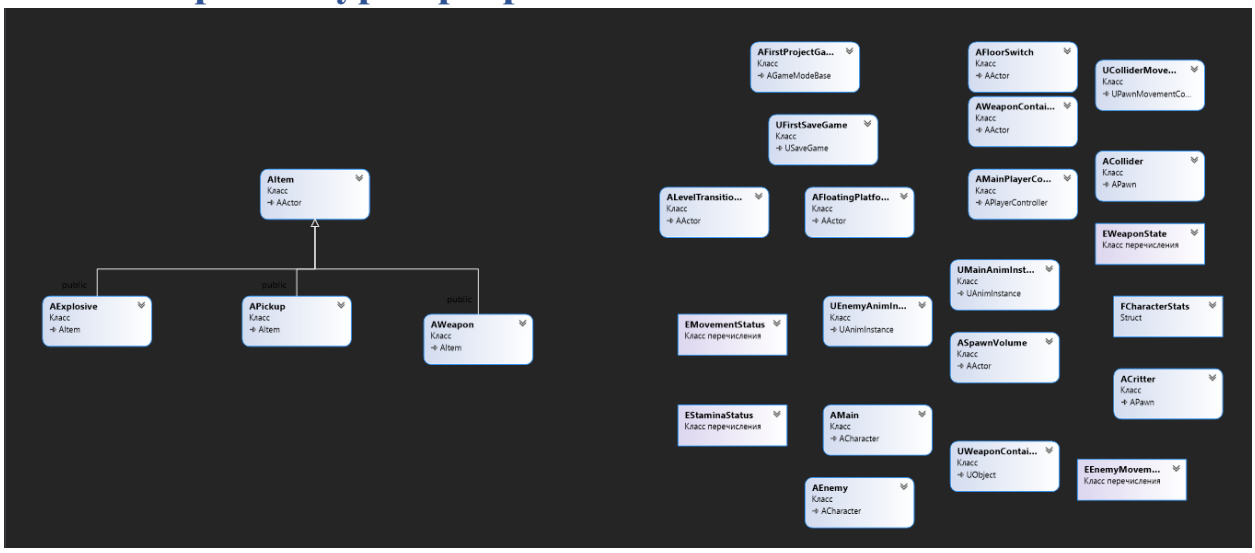

Рис. 4.2.1 Діаграма c++ класів

Так як програмне забезпечення дипломної роботи було розроблено за допомогою рушія UE4, то архітектура, яка була запроваджена, має такі властивості, яких вимагає даний рушій. Наприклад, програма має основний клас Main, записані основні налаштування симуляції.

Архітектура ПЗ повністю знаходися у парадигмі об’єктно-орієнтованого програмування. Деякі класи мають графічне представлення та Blueprint реалізація, деякі створені для внутрішніх налаштувань ПЗ.

Наслідування та асоціація – це основні зв’язки, які існують між класами. Наслідування представлене класами, які наслідуються від AItem. Це класи AExplosive, APickup та EWeapon. Агреграція має багато представлень, один з прикладів – це знаходження у класі зброї вказівника на головного персонажа.

ВИСНОВКИ

В даній роботі було проведено дослідження візуалізації різних типів об’єктів у графічних симуляціях на основі векторної алгебри та продемонстровано практичне застосування. Були використані сучасні інструменти для розробки ПЗ, яке створює графічну симуляцію з високою графікою, анімаціями, звуками, штучним інтелектом та елементами геймплею.

Код ПЗ побудований так, що програма може легко розширюватися для подальшого розвитку. В майже кожну строчку коду був доданий коментар для легкого розуміння. Усі змінні та функції були названі так як цього вимагає стандарт написання c++ на UE4, тому кожен розробник зможе зрозуміти як працює програма.

Завдяки рушію UE4 було створено сучасний продукт, який може зайняти свою нішу на ринку розробки ігор. В результаті проведеної роботи можна зазначити, що було використано векторну алгебру, c++, Blueprint та UE4 разом, що, на мою думку, складає новий етап у розробці відеоігор, так як ці компоненти дають дуже широкий спектр можливостей, коли вони поєднані в одне ціле.

Література

1. Game Programming in C++: Creating 3D Games

2. Mathematics for 3D Game Programming and Computer Graphics

3. Unreal Engine C++ Developer: Learn C++ and Make Video Games

4. Beginning Unreal Game Development

5. Game Development Projects with Unreal Engine

6. Blueprints Visual Scripting for Unreal Engine

7. Hands-On Artificial Intelligence with Unreal Engine

8. https://www.unrealengine.com/marketplace/en-US/store

9. https://docs.unrealengine.com/4.26/en-US/

10. Unreal Engine 4 for Design Visualization: Developing Stunning Interactive Visualizations, Animations, and Renderings (Game Design)

11. Master the Art of Unreal Engine 4 – Blueprints

12. An Introduction to Unreal Engine 4

13. Beginning Unreal Game Development: Foundation for Simple to Complex Games Using Unreal Engine 4

14. https://www.unrealengine.com/en-US/?lang=en_US&state=%2Findex.html

15. https://answers.unrealengine.com/index.html

16. https://www.mixamo.com/

Додаток А

Main

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "Main.generated.h"

UENUM(BlueprintType)
enum class EMovementStatus : uint8
{
	EMS_Normal UMETA(DisplayName = "Normal"),
	EMS_Sprinting UMETA(DisplayName = "Sprinting"),

	EMS_MAX UMETA(Display = "DefaultMAX")
};

UENUM(BlueprintType)
enum class EStaminaStatus : uint8
{
	ESS_Normal UMETA(DisplayName = "Normal"),
	ESS_BelowMinimum UMETA(DisplayName = "BelowMinimum"),
	ESS_Exhausted UMETA(DisplayName = "Exhausted"),
	ESS_ExhaustedRecovering UMETA(DisplayName = "ExhaustedRecovering"),

	EMS_MAX UMETA(Display = "DefaultMAX")
};

UCLASS()
class SUNTEMPLEGAME_API AMain : public ACharacter
{
	GENERATED_BODY()

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

	////////////////////////////////////////////////////////////////////////
	////CAMERA
	// CameraBoom positioning the camera behind the player
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Camera", meta =
(AllowPrivateAccess = "true"))
	class USpringArmComponent* CameraBoom;
	// FollowCamera
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Camera", meta =
(AllowPrivateAccess = "true"))
	class UCameraComponent* FollowCamera;
	// Base turn rate to scale turning functions for the camera
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Camera")
	float BaseTurnRate;
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Camera")
	float BaseLookUpRate;

	////////////////////////////////////////////////////////////////////////
	////PLAYER STATS
	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Player stats")
	float MaxHealth;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Player stats")
	float Health;
```

```
	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Player stats")
	float MaxStamina;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Player stats")
	float Stamina;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Player stats")
	int32 Coins;
	// Called to decrement health
	void DecrementHealth(float Amount);
	// Called to die
	void Die();
	// Called to incremt coins
	void IncrementCoins(int32 Amount);

	////////////////////////////////////////////////////////////////////////
	////MOVEMENT
	// Movement status
	UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "Enums")
	EMovementStatus MovementStatus;
	// Set movement status and speed
	FORCEINLINE void SetMovementStatus(EMovementStatus Status);
	// Running speed
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Running")
	float RunningSpeed;
	// Sprinting speed
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Running")
	float SprintingSpeed;
	// Is shift key down?
	bool bShiftKeyDown;
	// Shift key is down
	void ShiftKeyDown();
	// Shift key is up
	void ShiftKeyUp();

	////////////////////////////////////////////////////////////////////////
	////STAMINA
	// Stamina status
	UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "Enums")
	EStaminaStatus StaminaStatus;
	// Set stamina status
	FORCEINLINE void SetStaminaStatus(EStaminaStatus Status) { StaminaStatus = Status;
}
	// Stamina drain rate
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Running")
	float StaminaDrainRate;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Running")
	// Min sprint stamina
	float MinSprintStamina;

	////////////////////////////////////////////////////////////////////////
	////DEBUG SPHERES
	// Pickup locations
	TArray<FVector> PickupLocations;
	// Show all pickups locations
	UFUNCTION(BlueprintCallable)
	void ShowPickupLocations();

	////////////////////////////////////////////////////////////////////////
	////WEAPON
	// Weapon
	UPROPERTY(EditDefaultsOnly , BlueprintReadOnly, Category = "Items")
	class AWeapon* EquippedWeapon;
	// Set EquippedWeapon
	FORCEINLINE void SetEquippedWeapon(AWeapon* Weapon) { EquippedWeapon = Weapon; }
	// ActiveOverlapingItem
```

```
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Items")
	class AItem* ActiveOverlapingItem;
	// Set ActiveOverlapingItem
	FORCEINLINE void SetActiveOverlapingItem(AItem* Item) { ActiveOverlapingItem =
Item; }


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

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

	// Called to bind functionality to input
	virtual void SetupPlayerInputComponent(class UInputComponent*
PlayerInputComponent) override;

	// Called for forward/backward input
	void MoveForward(float Value);
	// Called for side to side input
	void MoveRight(float Value);

	// Called via input to turn at a given rate
	// @param Rate - is a normilized rate, i.e. 1.0f means 100% of desired turn rate
	void TurnAtRate(float Rate);
	// Called via input to look up/down at a given rate
	// @param Rate - is a normilized rate, i.e. 1.0 means 100% of desired look/up rate
	void LookUpAtRate(float Rate);

	// Is LMB down?
	bool bLMBDown;
	// If LMB is down
	void LMBDown();
	// If LMB is up
	void LMBUp();

	// Getters/Setters
	FORCEINLINE class USpringArmComponent* GetCameraBoom() const { return CameraBoom;
}
	FORCEINLINE class UCameraComponent* GetFollowCamera() const { return FollowCamera;
}

};



// Fill out your copyright notice in the Description page of Project Settings.

#include "Main.h"
#include "GameFramework/SpringArmComponent.h"
#include "Camera/CameraComponent.h"
#include "GameFramework/Controller.h"
#include "Engine/World.h"
#include "Components/InputComponent.h"
#include "Components/CapsuleComponent.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "Kismet/KismetSystemLibrary.h"
#include "Weapon.h"

// Sets default values
AMain::AMain()
{
```

```
	// 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;

	// Create CameraBoom (pulls towards the player if there's a collision)
	CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));
	CameraBoom->SetupAttachment(GetRootComponent()); // bind to root
	CameraBoom->TargetArmLength = 600.0f; // Camera follows at this distance
	CameraBoom->bUsePawnControlRotation = true; // ??? Rotate arm based on controller

	// Create FollowCamera
	FollowCamera = CreateDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"));
	FollowCamera->SetupAttachment(CameraBoom, USpringArmComponent::SocketName); //
bind to CameraBoom
	FollowCamera->bUsePawnControlRotation = false; // ???

	// Set size for collision capsule
	GetCapsuleComponent()->SetCapsuleSize(30.f, 105.f);

	// Don't rotate when the controller rotates
	bUseControllerRotationYaw = false;
	bUseControllerRotationPitch = false;
	bUseControllerRotationRoll = false;

	// Configure character movement:
	// charater moves in the direction of input
	GetCharacterMovement()->bOrientRotationToMovement = true;
	GetCharacterMovement()->RotationRate = FRotator(0.f, 540.0f, 0.f); // add
rotationn rate
	GetCharacterMovement()->JumpZVelocity = 650.0f;
	GetCharacterMovement()->AirControl = 0.2f; // control in air

	// Set up turn rates for input
	BaseTurnRate = 65.f;
	BaseLookUpRate = 65.f;

	// Player stats
	MaxHealth = 100.f;
	Health = 65.f;
	MaxStamina = 150.f;
	Stamina = 120.f;
	Coins = 0;

	// Movement
	MovementStatus = EMovementStatus::EMS_Normal;
	RunningSpeed = 650.f;
	SprintingSpeed = 950.f;
	bShiftKeyDown = false;

	// Stamine
	StaminaStatus = EStaminaStatus::ESS_Normal;
	StaminaDrainRate = 25.f;
	MinSprintStamina = 50.f;

	// LMB
	bLMBDown = false;
}

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

	// Debug sphere
```

```
	UKismetSystemLibrary::DrawDebugSphere(this, GetActorLocation() + FVector(0.f, 0.f,
75.f), 25.f, 12, FLinearColor::Green, 5.f, 0.5f);
}

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

	float DeltaStamina = StaminaDrainRate * DeltaTime;
	switch (StaminaStatus)
	{
	case EStaminaStatus::ESS_Normal:
		if (bShiftKeyDown)
		{
			Stamina -= DeltaStamina;
			if (Stamina <= MinSprintStamina)
			{
				SetStaminaStatus(EStaminaStatus::ESS_BelowMinimum);
			}
			SetMovementStatus(EMovementStatus::EMS_Sprinting);
		}
		else // shift key up
		{
			if (Stamina < MaxStamina) {
				Stamina += DeltaStamina;
			}
			SetMovementStatus(EMovementStatus::EMS_Normal);
		}
		break;

	case EStaminaStatus::ESS_BelowMinimum:
		if (bShiftKeyDown)
		{
			if (Stamina <= 0.f)
			{
				SetStaminaStatus(EStaminaStatus::ESS_Exhausted);
				SetMovementStatus(EMovementStatus::EMS_Normal);
			}
			else
			{
				Stamina -= DeltaStamina;
				SetMovementStatus(EMovementStatus::EMS_Sprinting);
			}
		}
		else // shift key up
		{
			if (Stamina >= MinSprintStamina)
			{
				Stamina += DeltaStamina;
				SetStaminaStatus(EStaminaStatus::ESS_Normal);
			}
			else
			{
				Stamina += DeltaStamina;
			}
			SetMovementStatus(EMovementStatus::EMS_Normal);

		}
		break;

	case EStaminaStatus::ESS_Exhausted:
		if (bShiftKeyDown)
		{
```

```
                        Stamina = 0.f;
                }
                else // shift key up
                {
                        SetStaminaStatus(EStaminaStatus::ESS_ExhaustedRecovering);
                        Stamina += DeltaStamina;
                }
                SetMovementStatus(EMovementStatus::EMS_Normal);
                break;

        case EStaminaStatus::ESS_ExhaustedRecovering:
                Stamina += DeltaStamina;
                if (Stamina >= MinSprintStamina)
                {
                        SetStaminaStatus(EStaminaStatus::ESS_Normal);
                }
                SetMovementStatus(EMovementStatus::EMS_Normal);
                break;
        }
}

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

        PlayerInputComponent->BindAction("Jump", IE_Pressed, this, &ACharacter::Jump);
        PlayerInputComponent->BindAction("Jump", IE_Released, this,
&ACharacter::StopJumping);
        PlayerInputComponent->BindAction("Sprint", IE_Pressed, this,
&AMain::ShiftKeyDown);
        PlayerInputComponent->BindAction("Sprint", IE_Released, this, &AMain::ShiftKeyUp);
        PlayerInputComponent->BindAction("LMB", IE_Pressed, this, &AMain::LMBDown);
        PlayerInputComponent->BindAction("LMB", IE_Released, this, &AMain::LMBDown);


        PlayerInputComponent->BindAxis("MoveForward", this, &AMain::MoveForward);
        PlayerInputComponent->BindAxis("MoveRight", this, &AMain::MoveRight);

        PlayerInputComponent->BindAxis("Turn", this, &APawn::AddControllerYawInput);
        PlayerInputComponent->BindAxis("LookUp", this, &APawn::AddControllerPitchInput);
        PlayerInputComponent->BindAxis("TurnRate", this, &AMain::TurnAtRate);
        PlayerInputComponent->BindAxis("LookUpRate", this, &AMain::LookUpAtRate);

}

// Called to move forward/back
void AMain::MoveForward(float Value)
{
        if ((Controller != nullptr) && (Value != 0.f))
        {
                const FRotator Rotation = Controller->GetControlRotation();
                const FRotator YawRotation(0.f, Rotation.Yaw, 0.f);
                const FVector Direction =
FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);
                AddMovementInput(Direction, Value);
        }
}
// Called to move right/left
void AMain::MoveRight(float Value)
{
        if ((Controller != nullptr) && (Value != 0.f))
        {
```

```
			// Find out wich way is forward
			const FRotator Rotation = Controller->GetControlRotation();
			const FRotator YawRotation(0.f, Rotation.Yaw, 0.f);
			const FVector Direction =
FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);
			AddMovementInput(Direction, Value);
		}
}

// Called to turn right/left
void AMain::TurnAtRate(float Rate)
{
		AddControllerYawInput(Rate * BaseTurnRate * GetWorld()->GetDeltaSeconds());
}
// Called to look up/down
void AMain::LookUpAtRate(float Rate)
{
		AddControllerPitchInput(Rate * BaseLookUpRate * GetWorld()->GetDeltaSeconds());
}

// If LMB is down
void AMain::LMBDown()
{
		bLMBDown = true;
		// ActiveOverlapingItem is valid
		if (ActiveOverlapingItem)
		{
			// Cast ActiveOverlapingItem to AWeapon. If it's valid - it
			// returns pointer, if not valid, it returns null
			AWeapon* Weapon = Cast<AWeapon>(ActiveOverlapingItem);
			if (Weapon)
			{
				// Equip weapon
				Weapon->Equip(this);
				// Set ActiveOverlapingItem to nulptr
				SetActiveOverlapingItem(nullptr);
			}
		}
}
// If LMB is up
void AMain::LMBUp()
{
		bLMBDown = false;
}

// Called to decrement health
void AMain::DecrementHealth(float Amount)
{
		Health -= Amount;
		if (Health <= 0) { Die(); }
}
// Called to die
void AMain::Die()
{

}
// Called to incremt coins
void AMain::IncrementCoins(int32 Amount)
{
		Coins += Amount;
}

// Set movement status
void AMain::SetMovementStatus(EMovementStatus Status)
```

```
	{
		MovementStatus = Status;
		if (MovementStatus == EMovementStatus::EMS_Sprinting)
		{
			GetCharacterMovement()->MaxWalkSpeed = SprintingSpeed;
		}
		else
		{
			GetCharacterMovement()->MaxWalkSpeed = RunningSpeed;
		}
	}

	// Shift key is down
	void AMain::ShiftKeyDown()
	{
		bShiftKeyDown = true;
	}
	// Shift key is up
	void AMain::ShiftKeyUp()
	{
		bShiftKeyDown = false;
	}

	// Show all pickups locations
	void AMain::ShowPickupLocations()
	{
		// Go through all location of pickups
		for (FVector currentLocation : PickupLocations)
		{
			// Debug sphere
			UKismetSystemLibrary::DrawDebugSphere(this, currentLocation, 25.f, 12,
	FLinearColor::Green, 5.f, 0.5f);
		}
}
```

Weapon

```
	// Fill out your copyright notice in the Description page of Project Settings.

	#pragma once

	#include "CoreMinimal.h"
	#include "Item.h"
	#include "Weapon.generated.h"

	/**
	 *
	 */
	UCLASS()
	class SUNTEMPLEGAME_API AWeapon : public AItem
	{
		GENERATED_BODY()

	public:
		// Constructor
		AWeapon();

		// Skeletal mesh
		UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "Skeletal Mesh")
		class USkeletalMeshComponent* SkeletalMesh;

		// Without FUNCTION() cause it's inherited from Item
```

```
	// Called to begin overlaping
	virtual void OnOverlapBegin(
			UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
UPrimitiveComponent* OtherComp,
			int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
override;
	// Called to end overlaping
	virtual void OnOverlapEnd(
			UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
			UPrimitiveComponent* OtherComp, int32 OtherBodyIndex) override;

	// Functio for equipping
	void Equip(class AMain* Main);

	// Equipping sound
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item | Sound")
	class USoundCue* OnEquipSound;

	// Is particles on a weapon?
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item | Particles")
	bool bWeaponParticles;
};



// Fill out your copyright notice in the Description page of Project Settings.

#include "Weapon.h"
#include "Components/SkeletalMeshComponent.h"
#include "Main.h"
#include "Engine/SkeletalMeshSocket.h"
#include "Sound/SoundCue.h"
#include "Kismet/GameplayStatics.h"
#include "Particles/ParticleSystemComponent.h"


// Constructor
AWeapon::AWeapon()
{
	SkeletalMesh =
CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("SkeletalMesh"));
	SkeletalMesh->SetupAttachment(GetRootComponent());

	bWeaponParticles = false;
}

// Called to begin overlaping
void AWeapon::OnOverlapBegin(
	UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent*
OtherComp,
	int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
	// Call functionality from parent class
	Super::OnOverlapBegin(
		OverlappedComponent, OtherActor, OtherComp,
		OtherBodyIndex, bFromSweep, SweepResult);

	// If OtherActor is valid
	if (OtherActor)
	{
		// Cast other actor to AMain. If it's valid - it
		// returns pointer, if not valid, it returns null
		AMain* Main = Cast<AMain>(OtherActor);
		// Check if Main is valid
```

```
			if (Main)
			{
				Main->SetActiveOverlapingItem(this);
			}
		}
}

// Called to end overlaping
void AWeapon::OnOverlapEnd(
		UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
		UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
{
		// Call functionality from parent class
		Super::OnOverlapEnd(
			OverlappedComponent, OtherActor,
			OtherComp, OtherBodyIndex);

		// If OtherActor is valid
		if (OtherActor)
		{
			// Cast other actor to AMain. If it's valid - it
			// returns pointer, if not valid, it returns null
			AMain* Main = Cast<AMain>(OtherActor);
			// Check if Main is valid
			if (Main)
			{
				Main->SetActiveOverlapingItem(nullptr);
			}
		}
}


void AWeapon::Equip(class AMain* Main)
{
		// If character is valid
		if (Main)
		{
			// Ignore camera
			SkeletalMesh->SetCollisionResponseToChannel(ECollisionChannel::ECC_Camera,
ECollisionResponse::ECR_Ignore);
			// Ignore pawn
			SkeletalMesh->SetCollisionResponseToChannel(ECollisionChannel::ECC_Pawn,
ECollisionResponse::ECR_Ignore);
			// Stop simulating physics
			SkeletalMesh->SetSimulatePhysics(false);

			// Get the RightHandSocket
			const USkeletalMeshSocket* RightHandSocket = Main->GetMesh()-
>GetSocketByName("RightHandSocket");
			// if RightHandSocket is valid
			if (RightHandSocket)
			{
				// Attach the socket to the character mesh
				RightHandSocket->AttachActor(this, Main->GetMesh());
				// Stop rotate
				bRotate = false;
				// Set the equipped weapon in Main
				Main->SetEquippedWeapon(this);

				// If OnEquipSound is valid
				if (OnEquipSound)
				{
					// Play sound
					UGameplayStatics::PlaySound2D(this, OnEquipSound);
```

```
                        }

                        // If not bWeaponParticles
                        if (!bWeaponParticles)
                        {
                                IdleParticlesComponent->Deactivate();
                        }
                }
        }
}
```

Item

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Item.generated.h"

UCLASS()
class SUNTEMPLEGAME_API AItem : public AActor
{
        GENERATED_BODY()

public:
        // Sets default values for this actor's properties
        AItem();


        // REMARK
        // The different between UParticleSystemComponent and UParticleSystem
        // is that the first one can onlye be created only with default subobject.
        // But the second one doesn't need subobject. It works for all classes
        // that have or doesn't have word "Component" int the end.


        // Base shape collision
        UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "Item | Collision")
        class USphereComponent* CollisionVolume;

        // Base mesh component
        UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "Item | Mesh")
        class UStaticMeshComponent* Mesh;

        // Particles component
        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item | Particles")
        class UParticleSystemComponent* IdleParticlesComponent;

        // Overlap particles
        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item | Particles")
        class UParticleSystem* OverlapParticles;

        // Overlap sound
        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item | Sounds")
        class USoundCue* OverlapSound;

        // Rotation
        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item | Properties")
        bool bRotate;
```

```
	// Rotation rate
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Item | Properties")
	float RotationRate;


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

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

	// Called to begin overlaping
	UFUNCTION()
	virtual void OnOverlapBegin(
			UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
UPrimitiveComponent* OtherComp,
			int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult);
	// Called to end overlaping
	UFUNCTION()
	virtual void OnOverlapEnd(
			UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
			UPrimitiveComponent* OtherComp, int32 OtherBodyIndex);

};



// Fill out your copyright notice in the Description page of Project Settings.

#include "Item.h"
#include "Components/SphereComponent.h"
#include "Components/StaticMeshComponent.h"
#include "Particles/ParticleSystemComponent.h"
#include "Kismet/GameplayStatics.h"
#include "Engine/World.h"
#include "Sound/SoundCue.h"

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

	// CollisionVolume
	CollisionVolume =
CreateDefaultSubobject<USphereComponent>(TEXT("CollisionVolume"));
	RootComponent = CollisionVolume;

	// Mesh
	Mesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Mesh"));
	Mesh->SetupAttachment(GetRootComponent());

	// IdleParticlesComponent
	IdleParticlesComponent =
CreateDefaultSubobject<UParticleSystemComponent>(TEXT("IdlePartcilesComponent"));
	IdleParticlesComponent->SetupAttachment(GetRootComponent());

	// Rroperties
	bRotate = false;
	RotationRate = 45.f;
```

```
	}

	// Called when the game starts or when spawned
	void AItem::BeginPlay()
	{
		Super::BeginPlay();
		CollisionVolume->OnComponentBeginOverlap.AddDynamic(this, &AItem::OnOverlapBegin);
		CollisionVolume->OnComponentEndOverlap.AddDynamic(this, &AItem::OnOverlapEnd);
	}

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

		if (bRotate)
		{
			FRotator Rotation = GetActorRotation();
			Rotation.Yaw += RotationRate * DeltaTime;
			SetActorRotation(Rotation);
		}
	}

	// Called to begin overlaping
	void AItem::OnOverlapBegin(
		UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent*
	OtherComp,
		int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
	{
		if (OverlapParticles)
		{
			UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), OverlapParticles,
	GetActorLocation(), FRotator(0.f), true);
		}
		if (OverlapSound)
		{
			UGameplayStatics::PlaySound2D(this, OverlapSound);
		}
	}

	// Called to end overlaping
	void AItem::OnOverlapEnd(
		UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
		UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
	{
}
```

FMatrix

```
	FORCEINLINE FRotationTranslationMatrix::FRotationTranslationMatrix(const FRotator& Rot,
	const FVector& Origin)
	{
	#if PLATFORM_ENABLE_VECTORINTRINSICS

		const VectorRegister Angles = MakeVectorRegister(Rot.Pitch, Rot.Yaw, Rot.Roll,
	0.0f);
		const VectorRegister HalfAngles = VectorMultiply(Angles,
	GlobalVectorConstants::DEG_TO_RAD);

		union { VectorRegister v; float f[4]; } SinAngles, CosAngles;
		VectorSinCos(&SinAngles.v, &CosAngles.v, &HalfAngles);
```

```
	const float	SP	= SinAngles.f[0];
	const float	SY	= SinAngles.f[1];
	const float	SR	= SinAngles.f[2];
	const float	CP	= CosAngles.f[0];
	const float	CY	= CosAngles.f[1];
	const float	CR	= CosAngles.f[2];

#else

	float SP, SY, SR;
	float CP, CY, CR;
	FMath::SinCos(&SP, &CP, FMath::DegreesToRadians(Rot.Pitch));
	FMath::SinCos(&SY, &CY, FMath::DegreesToRadians(Rot.Yaw));
	FMath::SinCos(&SR, &CR, FMath::DegreesToRadians(Rot.Roll));

#endif

	M[0][0]	= CP * CY;
	M[0][1]	= CP * SY;
	M[0][2]	= SP;
	M[0][3]	= 0.f;

	M[1][0]	= SR * SP * CY - CR * SY;
	M[1][1]	= SR * SP * SY + CR * CY;
	M[1][2]	= - SR * CP;
	M[1][3]	= 0.f;

	M[2][0]	= -( CR * SP * CY + SR * SY );
	M[2][1]	= CY * SR - CR * SP * SY;
	M[2][2]	= CR * CP;
	M[2][3]	= 0.f;

	M[3][0]	= Origin.X;
	M[3][1]	= Origin.Y;
	M[3][2]	= Origin.Z;
	M[3][3]	= 1.f;
}

FORCEINLINE FScaleMatrix::FScaleMatrix( const FVector& Scale )
	: FMatrix(
		FPlane(Scale.X,	0.0f,	0.0f,	0.0f),
		FPlane(0.0f,	Scale.Y,	0.0f,	0.0f),
		FPlane(0.0f,	0.0f,	Scale.Z,	0.0f),
		FPlane(0.0f,	0.0f,	0.0f,	1.0f)
	)
{ }
```

MainAnimInstance

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "Animation/AnimInstance.h"
#include "MainAnimInstance.generated.h"

/**
 *
```

```
 */
UCLASS()
class SUNTEMPLEGAME_API UMainAnimInstance : public UAnimInstance
{
	GENERATED_BODY()

public:
	virtual void NativeInitializeAnimation() override;

	UFUNCTION(BlueprintCallable, Category = AnimationProperties)
	void UpdateAnimationProperies();

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Movement)
	float MovementSpeed;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Movement)
	bool bIsInAir;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Movement)
	class APawn* Pawn;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Movement)
	class AMain* Main;

};



	// Fill out your copyright notice in the Description page of Project Settings.

#include "MainAnimInstance.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "Main.h"

void UMainAnimInstance::NativeInitializeAnimation()
{
	if (Pawn == nullptr)
	{
		Pawn = TryGetPawnOwner();
		if (Pawn)
		{
			Main = Cast<AMain>(Pawn);
		}
	}

}

void UMainAnimInstance::UpdateAnimationProperies()
{
	if (Pawn == nullptr)
	{
		Pawn = TryGetPawnOwner();
	}

	if (Pawn)
	{
		FVector Speed = Pawn->GetVelocity();
		FVector LateralSpeed = FVector(Speed.X, Speed.Y, 0.f);
		MovementSpeed = LateralSpeed.Size();
		bIsInAir = Pawn->GetMovementComponent()->IsFalling();

		if (Main == nullptr)
		{
			Main = Cast<AMain>(Pawn);
```

```
                }
            }
}
```

FloatingPlatform

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "FloatingPlatform.generated.h"

UCLASS()
class SUNTEMPLEGAME_API AFloatingPlatform : public AActor
{
	GENERATED_BODY()

public:
	// Sets default values for this actor's properties
	AFloatingPlatform();

	// Mesh for the platform
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Platform")
	class UStaticMeshComponent* Mesh;

	UPROPERTY(EditAnywhere)
	FVector StartPoint;

	// Create End point
	UPROPERTY(EditAnywhere, meta = (MakeEditWidget = "true"))
	FVector EndPoint;

	// Interp speed
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Platform")
	float InterpSpeed;

	// Interp speed
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Platform")
	float InterpTime;

	// Is interping or not
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Platform")
	bool bInterping;

	float Distance;

	// Timer
	FTimerHandle InterpTimer;

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

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

	// Called
	void ToggleInterping();
```

```
	// Called
	void SwapVectors(FVector& VecOne, FVector& VecTwo);

};



// Fill out your copyright notice in the Description page of Project Settings.

#include "FloatingPlatform.h"
#include "Components/StaticMeshComponent.h"
#include "TimerManager.h"

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

	Mesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Mesh"));
	RootComponent = Mesh;

	StartPoint = FVector(0.f);
	EndPoint = FVector(0.f);
	InterpSpeed = 4.f;
	InterpTime = 2.f;
	bInterping = false;
	Distance = 0.f;
}

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

	StartPoint = GetActorLocation();
	EndPoint += StartPoint;
	GetWorldTimerManager().SetTimer(InterpTimer, this,
&AFloatingPlatform::ToggleInterping, InterpTime);
	Distance = (EndPoint - StartPoint).Size();
}

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

	if (bInterping)
	{
		FVector CurrentLocation = GetActorLocation();
		// Smooth move
		FVector Interp = FMath::CInterpTo(CurrentLocation, EndPoint, DeltaTime,
InterpSpeed);
		SetActorLocation(Interp);

		float DistanceTraveled = (GetActorLocation() - StartPoint).Size();
		if (Distance - DistanceTraveled < 1.f)
		{
			ToggleInterping();
			GetWorldTimerManager().SetTimer(InterpTimer, this,
&AFloatingPlatform::ToggleInterping, InterpTime);
			SwapVectors(StartPoint, EndPoint);
		}
```

```
	}
}

void AFloatingPlatform::ToggleInterping()
{
	bInterping = !bInterping;
}

void AFloatingPlatform::SwapVectors(FVector& VecOne, FVector& VecTwo) {
	FVector Temp = VecOne;
	VecOne = VecTwo;
	VecTwo = Temp;
}
```