Міністерство освіти і науки України

Відокремлений структурний підрозділ «Тернопільський фаховий коледж Тернопільського національного технічного університету імені Івана Пулюя»
(повне найменування вищого навчального закладу)

Відділення телекомунікацій та електронних систем
(назва відділення)

Циклова комісія комп'ютерної інженерії
(повна назва циклової комісії)

ПОЯСНЮВАЛЬНА ЗАПИСКА
до кваліфікаційної роботи

бакалавра
(освітній ступінь)

на тему: Розробка комп'ютерної гри «Impartial Automaton» з використанням рушія Unreal Engine та мови C++

Виконав: студент VI курсу, групи КІб-602

Спеціальності **123 Комп'ютерна інженерія**
(шифр і назва, спеціальності)

Руслан ПОПОВИЧ
(ім'я та прізвище)

Керівник Ігор КАПАЦІЛА
(ім'я та прізвище)

Рецензент
(ім'я та прізвище)

Тернопіль – 2025

ВІДОКРЕМЛЕНИЙ СТРУКТУРНИЙ ПІДРОЗДІЛ
«ТЕРНОПІЛЬСЬКИЙ ФАХОВИЙ КОЛЕДЖ
ТЕРНОПІЛЬСЬКОГО НАЦІОНАЛЬНОГО ТЕХНІЧНОГО УНІВЕРСИТЕТУ
імені ІВАНА ПУЛЮЯ»

Відділення телекомунікацій та електронних систем
Циклова комісія комп'ютерної інженерії
Освітній ступінь бакалавр
Освітньо-професійна програма: Комп'ютерна інженерія
Спеціальність: 123 Комп'ютерна інженерія
Галузь знань: 12 Інформаційні технології

ЗАТВЕРДЖУЮ
Голова циклової комісії
комп'ютерної інженерії
_____ Андрій ЮЗЬКІВ
"08" травня 2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Поповичу Руслану Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи Розробка комп'ютерної гри "Impartial Automaton" з використанням рушія Unreal Engine та мови C++

керівник роботи Капаціла Ігор Богданович
(прізвище, ім'я, по батькові)

затверджені наказом Відокремленого структурного підрозділу «Тернопільський фаховий коледж Тернопільського національного технічного університетут імені Івана Пулюя» від 07.05.2025 р №4/9-224.

2. Строк подання студентом роботи: 21 червня 2025 року.

3. Вихідні дані до роботи: мова програмування C++, технічне завдання на розробку комп'ютерної гри, стандарти IEEE 29148-2018, IEEE 29119

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): Загальний розділ. Розробка технічного та робочого проєкту. Спеціальний розділ. Економічний розділ. Безпека життєдіяльності, основи охорони праці.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
   - діаграма використання програми;
   - діаграма класів програми;
   - діаграма послідовності дій;
   - текст методу;
   - блок схема алгоритму методу;
   - схема структурна.

6. Консультанти розділів роботи

Розділ	Ім'я, прізвище та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний розділ	Оксана РЕДЬКВА заст. директора з НВР		
Охорона праці, техніка безпеки та екологічні вимоги	Володимир ШТОКАЛО викладач		

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Отримання і аналіз технічного завдання	08.05	
2	Збір і узагальнення інформації	20.05	
3	Написання першого розділу	24.05	
4	Розробка технічного та робочого проекту	28.05	
5	Написання спеціального розділу	3.06	
6	Розрахунок економічної частини	5.06	
7	Написання розділу охорони праці	7.06	
8	Виконання графічної частини	10.06	
9	Оформлення проекту	14.06	
10	Погодження нормоконтролю	17.06	
11	Попередній захист роботи	21.06	
12	Захист кваліфікаційної роботи		

7. Дата видачі завдання: 08 травня 2025 року

Студент _____ Руслан ПОПОВИЧ
(підпис) (ім'я та прізвище)

Керівник роботи _____ Ігор КАПАЦІЛА
(підпис) (ім'я та прізвище)

АНОТАЦІЯ

Попович Р. Ю. Розробка комп'ютерної гри "Impartial Automaton" з використанням рушія Unreal Engine та мови C++: кваліфікаційна робота на здобуття освітнього ступеня бакалавр, за спеціальністю 123 Комп’ютерна інженерія. Тернопіль: ВСП «ТФК ТНТУ», 2025. 121с.

Дослідження присвячене розробці комп’ютерної гри «Impartial Automaton» з використанням рушія Unreal Engine та мови програмування C++. Об’єктом дослідження є процеси, що пов’язані із організацією та створенням ігрового застосунку. Предметом — застосування можливостей Unreal Engine і C++ для реалізації архітектури, геймплею та користувацького інтерфейсу. У результаті огляду і аналізу сучасних ігрових рушіїв показано, що одним із найефективніших підходів є розробка комп’ютерної гри з використанням Unreal Engine та мови програмування C++. Розроблено системи гри «Impartial Automaton» на базі Unreal Engine, що передбачає інтеграцію логіки, графіки та інтерфейсу користувача з можливістю подальшого розширення. Розроблено взаємодію між компонентами C++ і Blueprint для створення гнучкої ієрархії класів і об’єктів, що дозволяє підключати і змінювати модулі без порушення цілісності системи. Описано алгоритми роботи ключових компонентів гри та написано відповідне програмне забезпечення на мовах C++ та Blueprint. Проведено реалізацію інтерфейсу користувача (UI), обробку подій та передачу даних між компонентами системи. Проведено тестування реалізованих механік, перевірено їх продуктивність та стабільність роботи.

Ключові слова: комп’ютерна гра, Unreal Engine, C++, Blueprint, модульна архітектура, інтерфейс користувача, тестування, графічний рушій.

ANNOTATION

Ruslan POPOVYCH. Graduation Thesis on Topic Development of the computer game "Impartial Automaton" using the Unreal Engine and the C++ language: qualification work for obtaining a Bachelor's degree in Computer Engineering. Ternopil: SSS «TPC TNTU», 2025. 121 p.

The study is dedicated to the development of the computer game "Impartial Automaton" using the Unreal Engine and the C++ programming language. The object of qualification work is the processes associated with the organization and creation of a game application. The subject is the use of the capabilities of Unreal Engine and C++ to implement architecture, gameplay, and user interface. As a result of the review and analysis of modern game engines, it is shown that one of the most effective approaches is the development of a computer game using Unreal Engine and the C++ programming language. The systems of the game “Impartial Automaton” were developed using Unreal Engine, integrating game logic, graphics, and user interface in a modular architecture that supports future expansion. The interaction between C++ and Blueprint components has been designed to create a flexible hierarchy of classes and objects, allowing modules to be connected and modified without breaking the integrity of the system. The algorithms for the operation of key components of the game are described and the corresponding software is written in C++ and Blueprint. The user interface (UI), event processing, and data transfer between system components were implemented. The implemented mechanics were tested, their performance and stability were verified.

Keywords: computer game, UnrealEngine, C++, Blueprint, modular architecture, user interface, testing, graphics engine.

ЗМІСТ

ВСТУП

Сфера розваг пройшла великий шлях, музика, кіно (театр) та ігри супроводжували людство на кожному етапі його безупинного розвитку у тій чи іншій формі. З плином часу змінювлися методи та засоби, які використовувалися для їхнього створення чи взаємодії, в деяких випадках змінювалася й сама форма, як от на придачу до театральної сцени прийшов кінематограф. Проте, суть залишалась однією – дати людині цікавий досвід.

Завдяки комп'ютеризації процеси в нашому житті стали простішими, в тоу числі й ті, шо пов'язані з кіно, музикою й само собою іграми, які отримали нову форму у вигляді комп'ютерних ігор. Вони ж у свою чергу почали ставати більш гнучкими та технологічними і в результаті важливаою частинною життя багатьох людей.

У наш час комп'ютерні ігри використовуються не тільки для роваг, але й для тренування, симуляцій тощо. Наприклад, пілоти Формули-1 використовують автомобільні симулятори для тренування перед заїздами, адже в них можна налаштовувати різні погодні умови, пори року, конфігурації автомібілів тощо.

Метою кваліфікайної роботи є проектування та розробка комп'ютерної гри на рушії Unreal Engine 5, яка буде поєднювати у собі стилістику ігор жанру «souls» та механіки глибокого налаштування персонажа.

1 ЗАГАЛЬНИЙ РОЗДІЛ

1.1 Аналітичний огляд існуючих рішень

В недовгій, але насиченій, історії розвитку відеоігор було чимало хороших та поганих представників: деякі з них здобули світове визнання, у той час як інші залишилися незаміченими й забутими. Деколи стилістику гри виносять в окремий жанр, це можна назвати найвищим ступенем успіху, оскільки трапляється це рідко і такі прецеденти можна перерахувати на пальцях двох рук. Одним з них є жанр «souls-like».

Стилістика жанру «souls-like» полягає у складності ситуацій та чесності стосовно до гравця. Говорячи про складність, мається на увазі, що гравцю постійно потрібно бути уважним, підлаштовуватися під ворогів, шукати їхні вразливості та моменти для відповіді на їхні дії, а не просто збільшувати кількість здоров’я та шкоди недругів, як це роблять більшість розробників. Такий підхід дозволяє гравцю отримувати відчуття перемоги над випробуванням, яке він пройшов дійсно своїми силами. Варто зазначити, що в іграх цього жанру, як правило, немає вибору складності.

Щодо чесності, то тут варто відзначити, що попри свою високу складність, вони не є несправедливими. Усі вороги діють за прогнозованими шаблонами: запам’ятовуючи їхню поведінку та паттерни босів, можна їх перемогти. Механіки та правила зазвичай максимально прозорі, і не буває такого, що гра сама їх порушує. Наприклад, гравець чітко розуміє, як працюють ухиляння, блок, атака тощо. Користувач, налаштовуючи персонажа, сам підбирає, який стиль гри йому більше до вподоби, і гра не обмежує його одним-єдиним способом проходження.

Можна сказати, що «souls-like» більше навіть схоже на філософію створення гри, ніж просто на жанр.

Щодо до самих ігор, то можна відзначити такі існуючі рішення:

На рисунку 1.1 зображено обкладинку гри Dark Souls III.

Рисунок 1.1 – Обкладинка гри Dark Souls III

Першопроходцями у цьому жанрі є ігри серії Demon’s та Dark Souls. Розроблені японською компанією FromSoftware за керівництвом Хідетаки Міядзакі, якому і належить ідея створення жанру. На рисунку 1.2 зображено фото Хідетаки Міядзакі.

Рисунок 1.2 – Хідетака Міядзакі

Dark Souls III – третя й фінальна частина трилогії Dark Souls. Розроблена студією FromSoftware. Випущена 12 квітня 2016 року на PlayStation 4, Xbox One та Microsoft Windows. На рисунку 1.3 зображено скріншот з Dark Souls III.

Рисунок 1.3 - Скріншот з Dark Souls III

Переваги гри:

– хороший саундрек;
– цікаві вороги та боси;
– красиві та різноманітні локації;
– глибокий лор та сюжет;
– прописані неігрові персонажі;
– різноманітні ігрові ситуації;
– кооператив;
– великий арсенал зброї та обладунків;
– гнучке налаштування персонажа;
– перегравання.

Dark Souls III можна назвати іконою жанру, та варто брати до уваги рішення, прийняті під час її розробки, і в своїх іграх. Беручи це до уваги, досить складно назвати недоліки, оскільки вона доведна практично до ідеалу.

Lies of P – гра розроблена студією Round8 Studio та випущена 18 вересня 2023 року на PlayStation 4 та 5, Xbox Series X/S, Xbox One, Microsoft Windows та Mac OS. На рисунку 1.4 зображено обкладинку гри Lies of P.

Рисунок 1.4 - Обкладинка гри Lies of P

Гра бере персонажів з казки італійського письменника Карло Коллоді «Пригоди Піноккіо» та гармонійно їх вписує у зовсім новий світ, змінюючи їхні характери та мотиви на більш дорослі. На рисунку 1.5 зображено скріншот з гри Lies of P.

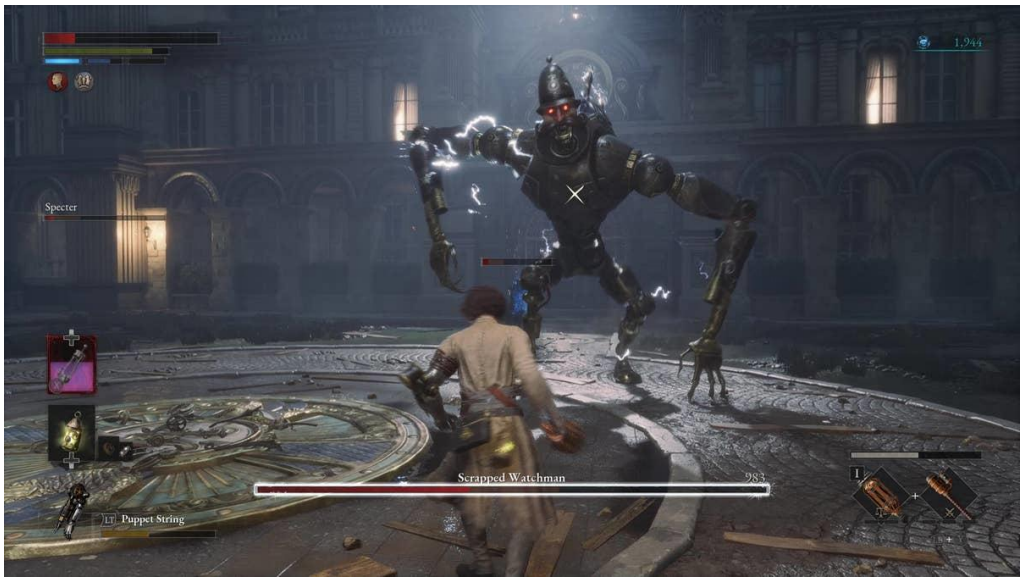

Рисунок 1.5 - Скріншот з гри Lies of P

Переваги гри:

– сюжет;

– саундтрек;

– прописані неігрові персонажі;

– механіка збірки зброї – гра дозволяє змінювати клинок та руків’я зброї, що є досить унікальним.

Недоліки:

– більшість босів та ворогів – присутні моменти, коли вони є досить несправедливими до гравця, хоча і не непереможні;

– прокачування навичок персонажа – не зрозуміло чому, але постійні предмети для відновлення здоров’я персонажа, які в інших іграх покращуються спеціальними предметами, помістили для прокачування у дереві навичків, як і нормальне ухиляння.

Незважаючи на усі недоліки, Lies of P все ще залишається хорошою грою, яка приносить задоволення своїми персонажами, історією та механіками.

Після аналітичного огляду існуючих на ринку аналогів, можна приступати до розробки технічного завдання, яке буде враховувати усі недоліки та сильні сторони проаналізованих варіантів.Технічне завдання повинно визначити наступні аспекти:

– найменування проекту;

– область застосування;

– призначення розробки;

– функціональні та нефункціональні вимоги програмного продукту;

– вимоги до документації;

– стадії розробки програмного забезпечення;

– техніко-економічні показники;

– правила тестування проекту.

1.2 Технічне завдання

1.2.1 Найменування та область застосування

Цей технічний документ стосується розробки комп’ютерної гри під назвою «Impartial Automaton», реалізованої мовою програмування C++.

Сфера використання: загальне застосування на персональних комп’ютерах.

1.2.2 Призначення розробки

Метою розробки є створення техно-демонстраційного варіанту гри у жанрі «souls-like». Призначений для загального застосування або у якості посилання в інших роботах з відповідним зазначенням першоджерела.

Засоби реалізації:

– редактор для створення 3D моделей – Blender;

– ігровий рушій – Unreal Engine 5.5;

– IDE для написання коду з встановленими пакетами для розробки на C++ та Unreal Engine – Visual Studio.

1.2.3 Вимоги до програмного забезпечення

Кінцевий результат комп’ютерної гри має задовільняти наступні вимоги(назви клавіш, кнопок тощо згадані англійською мовою):

1. Головне меню:

1.1 Головне меню має містити наступні кнопки для переходів: «Start», «About», «Exit».

1.2 Для взаємодії з головним меню повинна використовуватися комп’ютерна миша.

1.3 Для підтвердження має застосовуватися ліва кнопки комп’ютерної миші.

1.4 При взаємодії з кнопкою «Exit» – гра повинна припиняти свою діяльність та закриватись.

1.5 При виборі кнопки «About» – повинно відкриватися вікно, де описано хто та для чого розробив цю гру. Текст опису: «Made for qualification work by Ruslan Popovych».

1.6 Під час натискання кнопки «Start» – гра має завантажувати ігровий рівень з персонажем, ворогами та іншими об’єктами взаємодії.

1.7 У верхній частині головного меню має розміщуватися скорочена назва гри – «I. A.».

1.8 Клавіша мають мати наступний стиль: білі надписи та червоний задній фон, колір заднього фону кнопок – 760000FF (Hex Linear).

1.9 Дизайн головного меню не повинен викликати поганого самопочуття користувача.

2. Ігровий процес:

2.1 Гра повинна містити один рівень, на якому гравець може взаємодіяти з NPC та іншими інтерактивними об’єктами.

2.2 Під час ігрового процесу інтерфейс користувача має складатися лише з однією стрічки здоров’я.

2.3 Стрічка здоров’я має мати колір, який буде асоціюватися з здоров’ям.

2.4 Стрічка здоров’я має розміщуватися у лівому верхньому кута екрану та не має мішати ігровому процесу.

2.5 Гравець має мати клавіші для взаємодії з головним ігровим персонажем та світом.

2.6 Для переміщення головного ігрового персонажа мають використовуватися клавіші: «W» для переміщення вперед, «A» – вліво, «S» – назад, «D» – вправо.

2.7 Під час вільного переміщення головний ігровий персонаж завжди має дивитися в сторону, в яку він переміщується.

2.8 Гравець має мати можливість зациклити камеру на ворогу.

2.9 Для зациклення повинна використовуватися «Middle Mouse Button».

2.10 Для зняття зациклення також має використовуватися «Middle Mouse Button».

2.11 Під час зациклення, камера має обертатися навколо ворога та переміщуватися з головним ігровим персонажем.

2.12 Під час зациклення головний ігровий персонаж повинен переміщуватися відносно ворога та постійно дивитися в його сторону.

2.13 Під час зациклення мають використовуватися інші анімації переміщення, які призначенні для цього.

2.14 Якщо гравець відбігає на певну відстань від об’єкту зациклення – зациклення спадає і він повертається до вільного переміщення.

2.15 Гравець має мати змогу ухилятися від атак, нажаттям клавіші «L Shift».

2.16 Під час ухиляння, головний ігровий персонаж має бути невразливим до вхідної шкоди.

2.17 Під час ухиляння, головний ігровий персонаж не має мати змоги атакувати.

2.18 Під час ухиляння, головний ігровий персонаж не має мати змоги повторити ухиляння, якщо він вже його виконує.

2.19 Ухиляння повинні виконуватися у тому напрямку, який задає користувач.

2.20 Гравець має мати змогу наносити шкоду ворогам, за допомогою «Left Mouse Button».

2.21 Під час натискання «Left Mouse Button», повинна програватися анімація атаки, яка залежить від екіпірованої зброї на головному ігровому персонажі.

2.22 Під час атаки, мають програватися усі анімації, які є для поточної зброї, з можливістю обірвати програвання між переходами від однієї анімації до наступної.

2.23 Гравець має мати доступ до інвентарю, за допомогою клавіші «I».

2.24 Інвентар має розміщуватися у правій частині екрану.

2.25 Інвентар має містити усі предмети, які доступні на даний момент гравцю.

2.26 При взаємодії, предмет, якщо це зброя, повинен екіпіруватися на головного ігрового персонажа.

2.27 Гравець має мати змогу змінювати зброю, яку головний ігровий персонаж має у руках.

2.28 Гравець має мати змогу змінювати елементи тіла головного ігрового персонажа.

2.29 Під час зміни зброї, мають мінятися анімації атаки.

2.30 При відкритому інвентарі, у лівій нижній частині екрану має знаходитися кнопка для виходу з гри.

2.31 При повторному натисканні клавіші «I», при відкритому інвентарі, інвентар має закриватися.

2.32 Взаємодія з інвентарем має відбуватися за допомогою курсора миші.

2.33 Гравець має мати змогу для перемикатися між уже екіпірованої зброєї.

2.34 Для перемикання зброї на правій руці має використовуватися клавіша «Q» та «L» для лівої.

2.35 Під час перемикання активної зброї, неактивна зброя має переміщуватися на свій «Resort» слот.

2.36 Якщо атаки гравця попадає по ворогу, здоров’я ворога повинно опускатися та має програватися відповідна анімація реакції на попадання.

2.37 Якщо головний ігровий персонаж отримує шкоду, здоров’я має зменшуватися та це має відображатися на стрічці здоров’я.

2.38 На рівні мають бути розташовані «контрольні точки».

2.39 При взаємодії з контрольними точками, здоров’я гравця має відновлюватися та має змінюватися позиція відродження гравця на координати контрольної точки.

2.40 Коли здоров’я гравця падає до нуля – головний ігровий персонаж програє відповідну анімацію та переміщується до останньої контрольної точки, які він відвідав.

2.41 В кінці анімації реакції на падіння здоров’я до нуля, на екрані гравця має появлатися надпис «Fail» та не зникати до відродження на контрольній точці.

2.42 Під час вільного переміщення, гравець може вільно крутити камерою, при зацикленні він цього не має мати змогу робити.

3. Штучний інтелект неігрових персонажів:

3.1 Для розробки логіки поведінки неігрових персонажів необхідно використати «Behavior Tree».

3.2 Неігрові персонажі, якщо не мають головного ігрового персонажа в ролі зору, мають патрулювати по своїх заданих на рівні позиціях.

3.3 Під час патрулювання швидкість переміщення має бути змаеншена та має програватися анімація ходьби.

3.4 При досягненні позиції, неігровий персонаж має зупинитись та зачекати деякий час, після чого почати переміщення до наступної позиції.

3.5 Поточна позиція до якої прямує персонаж вибираєть випадково з заданих йому.

3.6 Якщо неігровий персонаж має у своємо полі зору головного ігрвого персонажа, він перестає патрулювати та починає переслідувати гравця.

3.7 Коли неігровий персонаж находиться у межах атаки до граця – він атакує його.

3.8 Після проведення атаки у неігрового персонажа починається недовга затримка перед наступною атакою.

3.9 Якщо гравець відбіг на достатню відстань від ворога, останній втрачає інтерес та повертається до патрулювання.

3.10 Якщо здоров’я неігрового персонажа падає до нуля – програється відповідна анімація, вимикаються його колайдери та він залишається у відповідній позі.

4. Продуктивність:

4.1 Необхідно забезпечити максимальну оптимізацію гра, для стабільного запуску та роботи на широкому спектрі різноманітних комплектуючих.

4.2 Необхідно забезпечити швидкий відклик між нажатою клавішею та результатом на екрані.

4.3 Забезпечити щоб гра використовувала якомога менше ресурів комп’ютера, зменшити навантаження на систему.

4.4 Уникати невиправдано складних логічних блоків, які можуть створити потенційні проблеми під час роботи програми.

5. Безпека:

5.1 Розповсюдження як інсталятор.

5.2 Забезпечити захист від зловмисного зараження файлів.

6. Сумісність:

6.1 Забезпечити сумісність з операційною системою – Windows 10.

6.2 Підтрика вводу з клавіатури та комп'ютерної миші.

6.3 Мінімальні системні вимоги:

- CPU: Intel Core i3 – 4160;
- RAM: 500 MB;
- GPU: GeForce GTX 1050Ti;
- HDD/SSD: 2GB.

1.2.4 Вимоги до програмної документації

Документація повинна містити супутні документи до програмного забезпечення, що охоплюють основні аспекти, необхідні для належної підготовки до його використання. Вона повинна описувати: правильне застосування програмного забезпечення та ключові аспекти.

Документація складається з:

– пояснювальної записки;

– технічного завдання;

– опису програми, що містить інформацію про логічну структуру та функціонування проєкту;

– кодів програми;

– специфікації, яка включає перелік та призначення файлів програми;

– програм та методик випробувань.

До документації висуваються наступні вимоги:

– відповідність ДСТУ;

– чіткість та зрозумілість;

– відсутність неоднозначностей.

1.2.5 Техніко-економічні показники

Для роботи обрано наступні засоби: редактор для 3D моделювання Blender, редактор для написання коду Microsoft Visual Studio 2022 з відповідними встановленими пакетами для роботи з C++ та Unrel Engine та ігровий рушій Unreal Engine 5.5. Для їхньої коректної роботи обрано персональний комп’ютер з наступними характеристиками:

– OS: Windows 10;

– CPU: AMD Ryzen 7 3700X 3.60 GHz;

– RAM: 64 GB;

– GPU: NVIDIA GeForce RTX 4060.

Часові ресурси виділені на роботу: трудові – 2 людино-місяці, машинні – 500 годин машинного часу.

Тип інтерфейсу ПЗ: графічний;

Мова інтерфейсу: англійський.

Інтерфейс програми мусить містити головне меню, в якому обов’язково мають бути наступні пункти: «Start», «About», «Exit».

1.2.6 Стадії та етапи розробки

Розробка включає в себе наступні етапи:

1. Аналітичний огляд наявних на ринку подібних рішень.
2. Формування та анліз вимог до комп’ютерної гри.
3. Розробка та аналіз технічного завдання.
4. Моделювання концепції гри, її особливостей.
5. Дизайн та аналіз ігрових механік.
6. Вибір інструментів реалізації згідно прийнятих рішень на минулих етапах.
7. Дизайн персонажів та локацій.
8. Створення 3D моделей та анімацій для них.
9. Імпорт моделей та анімацій в ігровий рушій.
10. Написання коду програми.
11. Створення інтерфейсу користувача.
12. Написання тестів, тестування та усунення виявлених проблем.
13. Збірка гри.
14. Виготовлення інсталятора.
15. Розробка супутної документації.
16. Введення в експлуатацію.

1.2.7 Порядок контролю та прийому

Тестування комп’ютерної гри відбувається за допомогою Unit-тестів та метод «чорної скриньки» для певних елементів. Поєднання даних методів дає змогу отримати максимальний результат.

Unit-тести дозволяють перевірити внутрішню логіку програми, компонентів тощо на предмет присутності помилок та жорстких залежностей. Повторне використання одного й того ж тесту забезпечує, що усе виконується при тих ж умовах, що і минуле тестування. Також вони дозволяють створювати певні умови, які було б складно відтворити при інших методах тестування.

Метод «чорної скриньки» дозволяє швидко перевірити певні елементи, написання тестів до яких було б не зовсім доцільно – наприклад, перевірка перевірка роботи оновлення стрічки здоров’я гравця.

Після проведення тестування виявленні помилки виправляються та повторно тестуються. Лише після успішного проходження тестування, гру можна вважати готовою, і тільки в цьому випадку можна розпочинати створення інсталятор та оформлення програмної документацію.

2 РОЗРОБКА ТЕХНІЧНОГО ТА РОБОЧОГО ПРОЄКТУ

2.1 Розробка загальної структури і варіантів використання програми

Структура програми – це те, як програмний продукт функціонує під капотом, який модулі використовуються, як обмінюється інформація між ними, опрацьовується тощо [2]. В Unreal Engine проект складається з окремих модулів, які користувач може сам добавляти чи забирати. По своїй суті, структура дуже нагадує архітектурний шаблон MVC (Model-View-Controller), але з певними особливостями.

На рисунку 2.1 зображено відношення елементів MVC між собою та користувачем.

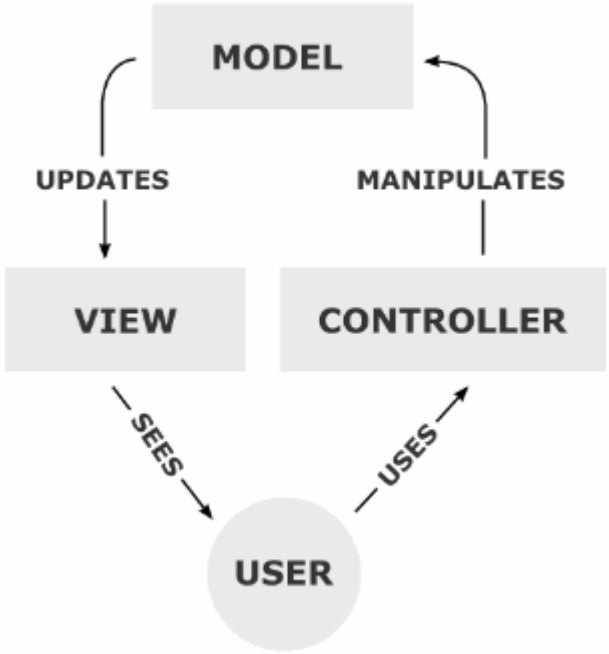

Рисунок 2.1 - Відношення елементів MVC між собою та користувачем

MVC (Model-View-Controller) – архітектурний шаблон, який визначає поділ системи на три компоненти: модель даних (Model), вигляд (View) та керування (Controller) [2].

Модель (Model) відповідає за керування бізнес логікою, обробкою та зберіганням даних. Приймає інформацію від Controller та передає дані на View [2].

Керування (Controller) відповідає за зняття даних з вводу та перетворення їх на команди для вигляду чи моделі [2].

Вигляд (View) це певне представлення інформації, отримане від моделі чи контролера. Також може передавати дані на контролер при взаємодії з інтерфейсом користувача.

Розглянемо цей архітектурний шаблон в контексті Unreal Engine.

ACharacter або клас гравця – це візуальне відображення гравця в світі, тому він виступає як View, однак він також може містити елементи обробки інформації, тому також можна сказати, що він є і Model. В деяких випадках він може виступати як View, Model та Controller, але таке нагромадження відповідальностей може спричинити проблеми у подальшій розробці, тому так краще не робити. Також як View виступають UserWidget, який використовується для створення інтерфейсу та AnimInstance, який відповідає за програвання анімацій.

Компоненти – це окремі класи, які відповідають за певний функціонал, наприклад MovementComponent, який є вбудований в Unreal Engine чи кастомні компоненти створені користувачем. Вони виступають як Model.

Controller – відповідає за зняття вводу від користувача і подальшої обробки. Як зрозуміло з назви він виступає як Controller з MVC. Є два види контролерів Controller та AIController. Перший призначений для гравця, другий для неігрових персонажів. Також як Controller може виступає GameMode, який керує усією грою в цілому та визначає її правила.

Як можна замітити, це не є жорстким слідуванням правил, а скорішещо деяка варіація на тему реалізації MVC. Не варто також забувати, що багато чого залежить і від самого розробника: Unreal Engine – це всього лиш інструмент, який не ставить програміста в жорсткі рамки.

Як вже було сказано ACharacter виступає як View, однак окрім нього є ще декілька класів для наслідування, які можна використати для представлення неігрових персонажів чи персонажів гравців. Розглянемо детальніше:

– Actor – базовий клас для усіх об’єктів, які розміщуються у світі. Може мати графічне представлення, колізії, звуки тощо, але не може мати логіки переміщення чи контролю за допомогою контролерів.

– Pawn – наслідується від Actor, має усі можливісті свого батьківського класу, але тепер його вже можна переміщувати за допомогою контролерів.

– Character – наслідується від Pawn та має за замовчуванням додаткові компоненти: CapsuleComponent, MovementComponent, SkeletalMeshComponent.

У таблиці 2.1 наведено порівняння між класами.

Таблиця 2.1 порівняння між класами

Клас	Наслідує	Керування через Controller	Рух/фізика	Додаткові можливості
Actor	-	Ні	Обмежений/Вручну	Рендер, колізії, логіка
Pawn	Actor	Так	Через Movement	Рух, зв’язок з контролером
Character	Pawn	Так	Розширений	Capsule, Movement, Skeletal Mesh

User Widget – клас для наслідування при створені елементів інтерфейсу. Релізація відбуваєтсья у спеціальному вікні у якому можна виставляти заготовлені елементи чи створені розробником. На рисунку 2.2 зображено вікно редактору для створення UI.

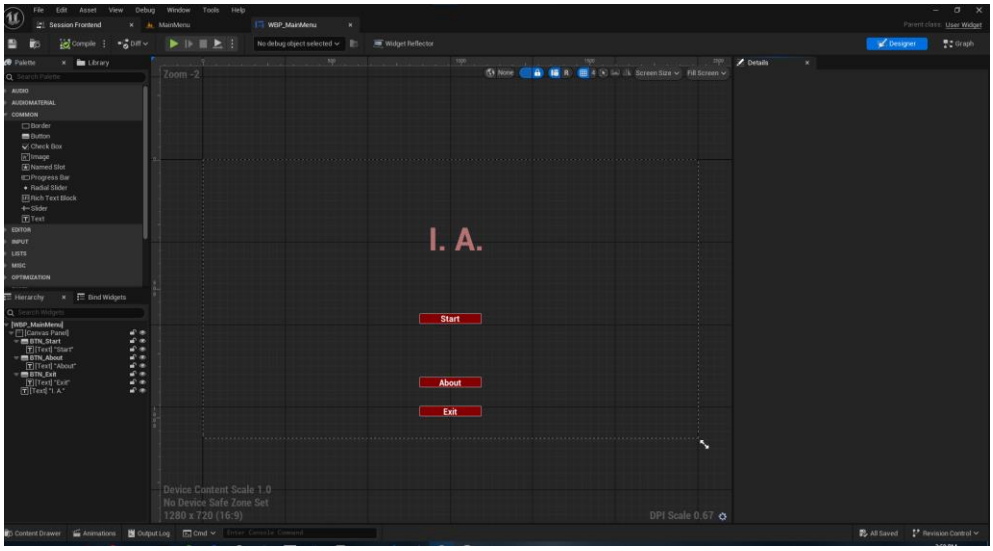

Рисунок 2.2 – Вікно редактору для створення UI

Також у вкладці «Graph» можна налаштовувати логіку, для опрацювання натискання на кнопку наприклад, за допомогою Blueprint системи. На рисунку 2.3 зображено редактор «Graph».

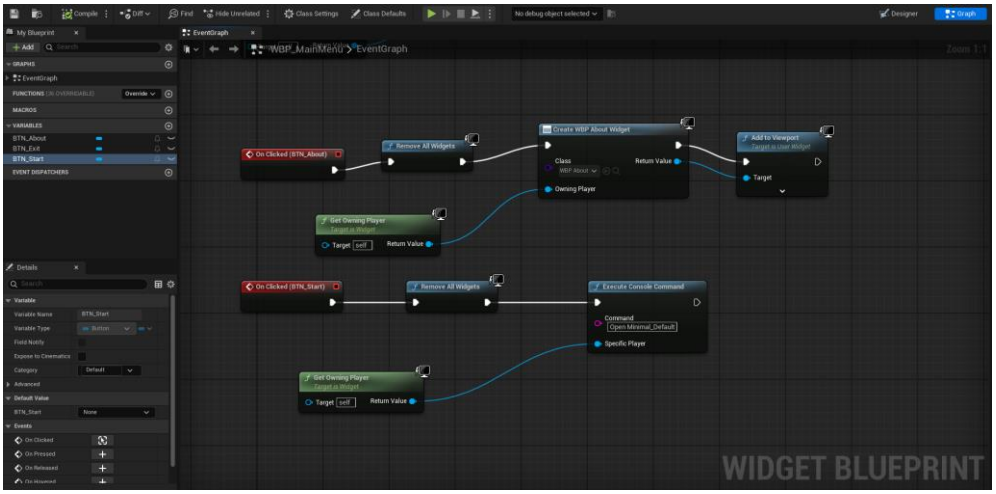

Рисунок 2.3 – Редактор «Graph»

Компоненти є важливою частиної розробки гри на Unreal Engine, вони дозволяють написати систему один раз і далі перевикористовувати де необхідно. Для створення кастомного компонента, необхідно при створенні класу вибрати Actor Component для наслідування. На рисунку 2.4 зображено клас для наслідування при створені кастомного компонента.

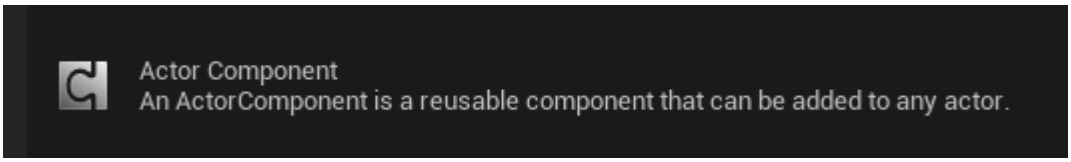

Рисунок 2.4 - Клас для наслідування при створені кастомного компонента

Контролери – це спеціальні класи, які відповідають за контроль персонажів: Controller для гравця та AIController – для неігрових персонажів. Для створення власного контролера необхідно вибрати як клас для наслідування відповідний клас. У ньому можна описувати додаткову логіку для керування певним актором. Разом з контролером гравця, використовується система «Enhanced Input» – це плагін, який поставляється разом з рушієм. Даний плагін дозволяє централізувати система керування за допомогою Input Mapping Context та Input Action.

Input Action – це певна подія: рух, стрибок, атака тощо. Їх можна додатково налаштовувати за допомогою тригерів та модифікаторів, які можуть міняти вхідні сигнали.

Input Mapping Context – це певний набір Input Action, який визначає загальне керування. Тут також можна добавляти різні модифікатори та тригери, щоб не робити цього на самих діях, що дозволює контролювати усе з одного місця. Тут встановлюються клавіші, які відповідають за той чи інший Input Action.

На рисунку 2.5 зображено Input Mapping Context.

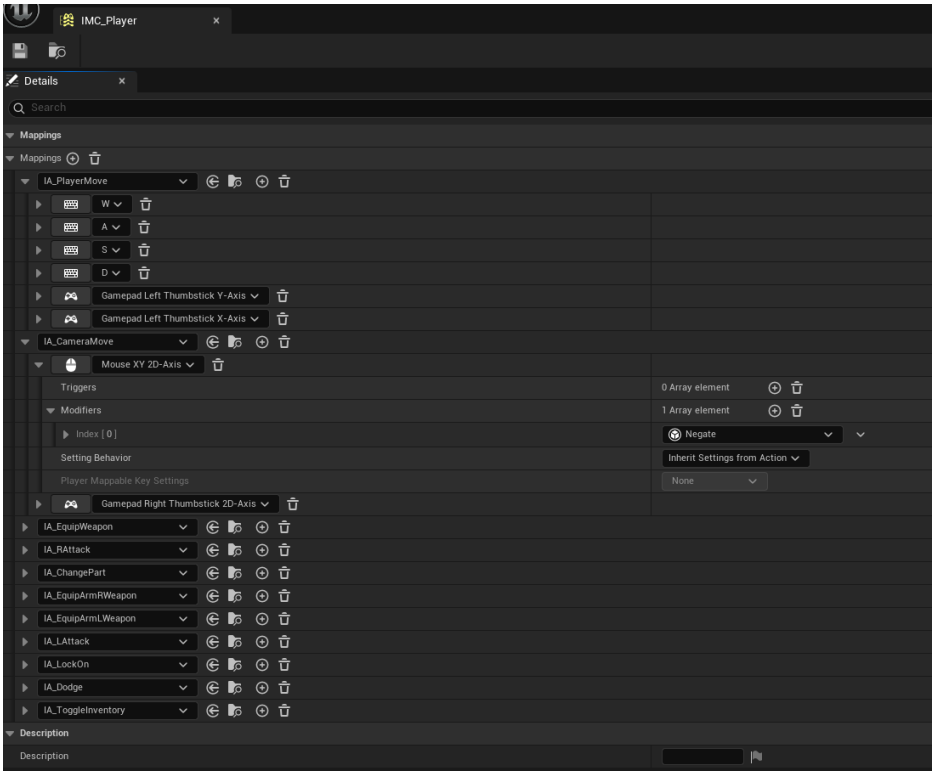

Рисунок 2.5 - Input Mapping Context

Весь код компонентів, контролерів тощо – можна знайти у додатку А.

Варіанти використання програми:

– запустити програму;
– відкрити сторінку «About»;
– вийти з програми;
– розпочати гру;
– переміщувати персонажа;
– ухилятися;
– взаємодіяти з неігровими персонажами;
– зациклюватися на ворозі;
– відроджуватися.

Графічне представлення варіантів використання програми наведено у додатку «Варіанти використання програми».

2.2 Розробка системи класів

Для зручності систему класів поділено на групи: персонажі, компоненти, предмети, загальні.

Персонажі.

Основним класом для усіх наступних персонажів є «MyBaseCharacter», який наслідується від стандартного класу «Character». Тут визначаються основні поля та методи, які будуть необхідні для наступних класів. Отже, тут є методи для роботи з тегами: GrantTag, CheckTag, RemoveTag – дані методи надають тег, перевіряють чи є тег та видаляють тег з персонажа відповідно, приймають референс FGameplayTag, CheckTag повертає булеве значення. Наявні методи для отримання основних компонентів: GetStatComponent, GetCombatComponent – які повертають вказівники на компоненти IIStatComponent та IICombatComponent відповідно. Методи для роботи опрацювання пошкоджених ворогів: AddDamagedActors – приймає вказівник AActor, HasDamagedActor – приймає вказівник AActor та повертає булеве значення, ResetDamagedActor. Методи для визначення точки відродження та самого відродження: SetActorSpawnLocation – приймає константний референс FVector, GetActorSpawnLocation – повертає константий референс FVector, Respawn. Опрацювання попадання по персонажові: OnCapsuleOverlap – приймає OverlappedComp типу вказівник UPrimitiveComponent, OtherActor типу вказівник AActor, OtherComp типу вказівник UPrimitiveComponent, OtherBodyIndex типу int32, bFromSweep - булеве значення, SweepResult – константний референс FHitResult, HandleHit – приймає вказівник AMyBaseCharacter. Поля: m_StatComponent типу вказівник UMyStatComponent, m_MontagesMap типу TMap<FName, UAnimMontage*>, m_DeathMontage типу вказівник UAnimMontage, m_CharacterTags типу FGameplayTagContainer – контейнер для зберігання тегів, m_DamagedActors типу TSet<AActor*>, m_bIsDead, m_bIsInvincible – булеві значення,

m_CombatComponent типу вказівник UCombatComponent, m_SpawnLocation типу FVector, m_GM – вказівник AMyGameModeBase.

«MyPlayerCharacter» – основний клас для персонажу гравця, наслідується від «MyBaseCharacter». Містить методи для роботи з SkeletalMesh: InitializeModularMesh – прйимає FName, ChangeSkeletalMesh та GetSKMeshPtr – приймає константний референс FName, повертає вказівник на вказівник USkeletalMeshComponent. Методи для роботи з колайдерами SkeletalMesh: SetActiveCollider – приймає константний референс FName, GetActiveCollider – повертає константний референс FName, GetCollidersMap – повертає TMap<FName, UCapsuleComponent*>, UpdateActiveColliderPosition, EnableCollider, DisableCollider. Методи для екіпірування: ChangeItem, SetSkeletalItem. Також містить інші методи для взаємодії з аніматором, компонентами тощо. Поля: m_CameraBoom типу вказівник USpringArmComponent, m_Camera – вказівник UCameraComponent, m_LockOnComponent – вказівник ULockOnComponent, m_ModularSKMeshMap – типу TMap<FName, USkeletalMeshComponent*>, m_AnimInstance типу вказівник UPlayerAnimInstance, m_ActiveCollider – Fname, m_Inventory – вказівник UInventoryComponent.

«Enemy» – основний клас для ворожих неігрових персонажів, наслідується від «MyBaseCharacter». Містить методи для патрулювання, переслідування: OnSeePawn – приймає вказівник типу APawn, StopChasing, ChoosePatrolTarget – повертає вказівник AActor*, Attack, StopAttack, RotateToPlayer. Методи для припинення роботи та відродження: TurnOffAI, Respawn.

Компоненти.

«ICombatComponent» – інтерфейс для «CombatComponent». Має методи для екіпірування та маніпуляції зі зброєю: EquipWeapon, ReattachFromTo, SwapWeapon, HasWeaponIn. Для роботи з монтажами зброї: GetActiveMontage, GetMontage, GetLRMontage, ChangeMontage.

«CombatComponent» – компонент, який наслідує від інтерфейсу «ICombatComponent». Перевизначає методи з інтерфейсу. Та має методи для роботи з колайдерами зброї: EnableHitbox, DeactivateHitbox. Та для встановлення зброї для заміни: SetWeaponClass.

«ILockOnComponent» – інтерфейс для компонента зациклення на неігровому персонажі. Має наступні методи: StartLockOn, EndLockOn, ToggleLockOn, GetTarget. StartLockOn – приймає значення з плаваючою крапкою для радіусу в якому потрібно знайти ціль для зациклення. ToggleLockOn також приймає значення з плаваючою крапкою для радіусу і перемикає стани між зацикленим та незацикленим. GetTarget повертає вказівник типу AActor.

«LockOnComponent» – наслідується від «ILockOnComponent». Перевизначає його методи.

«IStatComponent» – інтерфейс для компонента, який відповідає за параметри персонажа: здоров'я, витривалість тощо. AddStat – метод для додавання параметру, приймає FName& та три значення з плаваючою крапкою для початкового значення, мінімального та максимального значень. SetStatValue – приймає FName& для назви параметру та float для значення, встановлює поточне значення параметра. SetStatMinValue, SetStatMaxValue – методи для встановлення мінімального та мксимального значень, приймають FName& для назви параметру та float для значення. GetStatMaxValue, GetStatValue – методи для отримання максимального та поточного значень, приймають FName& для назви параметру.

«MyStatComponent» – наслідує від «IStatComponent», перевизначає його методи. Має додатковий метод GetPercentage, який приймає FName& для назви параметру та повертає float значення відсотка і ReduceHealth, який міняє значення стрічки здоров'я.

«InventoryComponents» – компонент для зберігання предметів. Має методя GetAllItems, який повертає референс на масив (TArray) структури FFItemData. Має атрибут TArray<FItemData>, який зберігає усі предмети.

Предмети.

«FItemData» – структура, яка містить FName для назви предмета, Utexture2D* для зображення предмета та TSubclassOf<AMyBaseWeapon> – вказівник на любий клас похідний від AMyBaseWeapon.

«MyBaseWeapon» – основний клас для зброї. GetAttachToSocketName – повертає FName сокета зброї, до якого потрібно прикріпити її. GetWeaponAttackMontage – повертає UAnimMontage* на анімацію. GetTags – повертає FGameplayTagContainer& – контейнер тегів. SetAttachTo – приймає FName&, замінює поле m_AttachTo. Також містить поля m_WeaponAttackMontage* – анімація атаки зброї та m_WeaponTags – теги зброї.

«MyWeapon» – наслідується від «MyBaseWeapon». GetResortSocket() – повертає FName на назву сокета, на який зброя переміщається в неекіпірованому стані. ActivateHitbox – активовує колайдер зброї. DeactivateHitbox – вимикає колайдер зброї. ResetDamagedActors – очищає список акторів, яким нанесено шкоду. Поля: m_WeaponMesh* – вказівник на статичний меш, m_Collider – вказівник на колайдер, m_ResortSocket – назва сокету «відпочинку», m_DamagedActors – масив акторів, яким нанесено шкоду.

«MySkeletalModule» - наслідується від «MyBaseWeapon». GetSkeletalMesh – повертає вказівник на USkeletalMesh. GetBeginSocket – повертає FName& назва сокета початку колайдера. GetEndSocket – повертає FName& назву сокета кінця колайдера. Поля: m_SkeletalMesh, m_StartSocket, m_EndSocket.

Загальні.

«MyGameModeBase» – «режим» гри. Методи: AddRnrmyToArray – добавляє на початку гри усіх ворогів на рівні в масив, приймає вказівник AMyBaseCharacter, RespawnAllEnemies – відроджує всіх ворогів, AddPlayerToArray – добавляє всіх гравців на початку гри в масив, приймає вказівник AMyBaseCharacter, RespawnAllPlayers – відроджує всіх гравців. Має

поля: m_EnemiesArray типу TArray<AMyBaseCharacter*> та m_PlayersArray типу TArray<AMyBaseCharacter*>.

«MyBaseAnimInstnace» – кастомний базовий клас для роботи з анімаціями персонажів. Методи: NativeInitializeAnimation та NativeUpdateAnimation, який приймає float значення. Поля: m_CurrentPawn – вказівник APawn на поточного власника, m_CharacterVelocity – Fvector зберішає швидкість персонажа, m_CharacterDirection – float, напрямок персонажа, m_PlayerMovementComponent – вказівник UCharacterMovementComponent, на компонент переміщення персонажа.

«PlayerAnimInstance» – наслідується від «MyBaseAnimInstance», призначений для роботи з анімаціями гравця. Методи: PlayDodge, який приймає const float кута, SetSkeletalUpDown – приймає два булеві значення для того чи права чи ліва частина та чи піднята чи опущена, HandleUpdatetarget – приймає вказівник AActor на зацикленого ворога, SetIsDead – приймає булева значення. Поля: m_Player – вказівник AMyPlayerCharacter на клас власника гравця, m_ForwardDodge, m_BackwardsDodge, m_LeftDodge, m_RightDodge – вказівник UAnimMontage для відповідних анімацій ухиляння в певному напрямку, m_bIsLockOn, m_bIsDead, m_bIsSkeletalLUp, m_bIsSkeletalRUp – bool значення для контролю анімацій.

«MyPlayerController» – кастомний контролер для гравця. Методи: ResetDodging, SetIsDodging – ймають параметрів, призначенні для роботи з полем m_IsDodging, SetIsAttacking – приймає bool значення, Move, Look, PlayRAttack, PlayLAttack, SwitchRWeapon, SwitchLWeapon, LockOn, Dodge – приймають FInputActionValue&, PlayAttack – приймає булеве значення, SwitchWeapon – приймає два референса FName, вказівник UAnimMontage та булеве значення. Поля: m_PlayerInputContext типу вказівник UInputMappingContext, m_MoveAction, m_CameraAction, m_WeaponRAttack, m_WeaponLAttack, m_WeaponRSwitch, m_WeaponLSwitch, m_LockOnAction, m_DodgeAction, m_ToggleInventory – типу вказівник UInputAction, m_VectorDirection типу FVector2D, m_bIsDodging, m_bIsAttacking – булеві

значення, m_ControlledPlayer – типу вказівник AMyPlayerCharacter, m_CurrentAnimInstance – типу вказівник UAnimInstance.

«Checkpoint» – містить методи: GetLocation – повертає FVector, OnOverlapBegin – приймає OverlappedComp типу вказівник UPrimitiveComponent, OtherActor типу вказівник AActor, OtherComp типу вказівник UPrimitiveComponent, OtherBodyIndex типу int32, bFromSweep - булеве значення, SweepResult – константний референс FHitResult. Поля: m_CheckpointLocation – Fvector, m_SphereCollider – вказівник USphereComponent, m_Mesh – вказівник UStaticMeshComponent.

Графічне представлення класів зображено у додатку «UML-діаграма класів».

2.3 Розробка методів

У даному розділі наведено розробку та реалізацію методів класів: MyBaseCharacter, MyPlayerCharacter. Описано різницю між поліморфними методами.

MyPlayerCharacter.

На рисунку 2.6 зображено реалізацію конструктора класу MyBaseCharacter.

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

    m_StatComponent = CreateDefaultSubobject<UMyStatComponent>("Stat Component");

    GetCapsuleComponent()->OnComponentBeginOverlap.AddDynamic(this,
        &AMyBaseCharacter::OnCapsuleOverlap);

    GetCapsuleComponent()->SetCollisionResponseToChannel(ECollisionChannel::ECC_Camera,
        ECollisionResponse::ECR_Ignore);

    m_bIsDead = false;
}
```

Рисунок 2.6 – Реалізація конструктора класу MyBaseCharacter

PrimaryActortick.bCanEverTick – визначає чи буде кожного кадрого викликатися метод Tick. Це корисно якщо потрібно постійно обробляти певні дані, наприклад позицію актора.

m_StatComponent = CreateDeafaultSubobject<UMyStatComponent>(“Stat Component”) – створює компонент UMyStatComponent через шаблонний метод та називає його Stat Component. Це буде відображатися в Outliner та blueprint`і актора, якщо забезпечити певну видимість. CreateDeafaultSubobject<>() повертає вказівник на створений об’єкт. m_StatComponent – це вказівник на клас UMyStatComponent, як вже було описано в минулому розділі. На рисунку 2.7 зображено створений об’єкт та його назва в blueprint`і гравця.

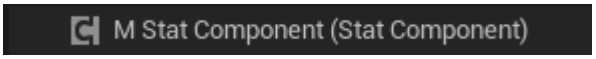

Рисунок 2.7 – Створений об’єкт та його назва в blueprint`і гравця

M Stat Component – назва атрибуту в класі, Stat Component – назва вказана в CreateDeafaultSubobject<>().

GetCapsuleComponent()->OnComponentBeginOverlap.AddDynamic(this, &AMyBaseCharacter::OnCapsuleOverlap) – отримує вказівник на колайдер з батьківського класу ACharacter та підписується на подію, коли щось перекривається з компонентом. Передає клас в якому є метод для виклику при події та адресу самого методу.

GetCapsuleComponent() -> SetCollisionResponseToChannel (ECollisionChannel::ECC_Camera, ECollisionResponse::ECR_Ignore) – визначає рекцію на перекривання з певними каналами. В даному випадку реакція на перекриття з каналом камери буде ігноруватися.

m_bIsDead = false – явно встановлюємо початкове значення на false, для того щоб бути впевненими у правильності початкових значень при створенні об’єкту.

Оскільки в конструкторі згадується метод OnCapsuleOverlap, доцільно наступним розглянути його. На рисунку 2.8 зображено реалізацію методу OnCapsuleOverlap класу MyBaseCharacter.

```
void AMyBaseCharacter::OnCapsuleOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp,
    int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    if (m_bIsDead || m_bIsInvincible) return;
    if (OtherActor && OtherActor != this && OtherComp)
    {
        if (OtherComp->ComponentHasTag(FName("WeaponCollider")))
        {
            AMyBaseCharacter* Damager;

            AActor* Attacker = OtherComp->GetOwner();

            if (!IsValid(Damager = Cast<AMyBaseCharacter>(Attacker)))
            {
                Attacker = Attacker->GetOwner();
                Damager = Cast<AMyBaseCharacter>(Attacker);
            }

            if (!Damager || Damager == this) return;

            UMyStatComponent* AttackerStat = Cast<UMyStatComponent>(Damager->GetStatComponent());

            if (AttackerStat && m_StatComponent)
            {
                if (Damager->HasDamagedActor(this)) return;

                float DamageAmount = AttackerStat->GetStatValue("Damage");

                m_StatComponent->AddStatCurrentValue("Health", DamageAmount);
            }
            if (this->m_StatComponent->GetStatValue("Health") <= 0.f)
            {
                m_bIsDead = true;
            }

            HandleHit(Damager);
        }
    }
}
```

Рисунок 2.8 - Реалізацію методу OnCapsuleOverlap класу MyBaseCharacter

UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult – усі параметри отримують свої значення від OnComponentBeginOverlap. OverlappedComponent – це компонент на акторі, з яким відбулось перекриття, OtherActor – інший актор, який має OtherComp з яким відбулось перекриття.

Першочергово перевіряємо чи поточний актор є «живим» та чи він є вразливим.

Наступним перевіряємо чи OtherActor не є nullptr, OtherActor не є собоє ж та чи OtherComp не є nullptr. Наступна перевірка на присутність у іншого

компонента тегу «WeaponCollider», що дозволяє визначати чи потрібно отримати шкоду.

Далі створюємо посилання на AMyBaseCharacter та Attacker, з останнім пробуємо отримати власника іншого компонента. Перевіряємо чи можемо скастувати до AMyBaseCharacter, якщо не можемо то пробуємо взяти ще раз власника. Це робиться для того, щоб якщо перекриття відбулося з іншим актором, який прив’язаний до персонажа, то ми отримаємо саме персонажа на якому розміщений компонент з параметрами.

Знову перевіряємо не nullptr, тому що з вказівниками так потрібно працювати, в гіршому випадку при спробі дереференсу nullptr програма крашниться. Отримуємо вказівник на компонент з параметрами.

Після наступних перевірок наявності об'єктів, перевіряємо чи під час поточної атаки вже було нанесено шкоду даному актору. Якщо шкода не було нанесена витягуємо відповідний параметр з компоненту та додаємо його значеня, яке є від'ємним, до поточного здоров'я. Перевіряємо чи воно не дорівнює нуля та викликаємо метод для обробки попадання, який в базовому класі є пустим.

Після виклики конструктора – викликається метод BeginPlay(). В ньому доцільно виконувати логіку, для якої вже необхідно мати створенні інші об’єкти. На рисунку 2.9 зображено реалізацію методу BeginPlay класу AMyBaseCharacter.

```
void AMyBaseCharacter::BeginPlay()
{
    Super::BeginPlay();

    m_SpawnLocation = this->GetActorLocation();

    m_GM = GetWorld()->GetAuthGameMode<AMyGameModeBase>();
}
```

Рисунок 2.9 – Реалізація методу BeginPlay класу AMyBaseCharacter

Саме в BeginPlay() знімаємо початкове значення появи актора та посилання на GameMode, бо можемо бути впевненні, що на цьому моменті вже існує світ, контролери тощо.

На рисунку 2.10 зображено реалізацію метода Respawn класу MyBaseCharacter.

```
void AMyBaseCharacter::Respawn()
{
    m_bIsDead = false;

    m_StatComponent->AddStatCurrentValue("Health",
        m_StatComponent->GetStatMaxValue("Health"));

    SetActorLocation(m_SpawnLocation);
}
```

Рисунок 2.10 – Реалізація метода Respawn класу MyBaseCharacter

Даний метод викликається після «смерті» персонажа і встановлює значення стану m_bIsDead на false, відновлює здоров’я, беручи його максимальне значення з компонента параметрів та перміщує актора на його позиція появи.

MyPlayerCharacter.

На рисунку 2.11 зображено реалізацію методу BeginPlay класу MyPlayerCharacter.

```
void AMyPlayerCharacter::BeginPlay()
{
    Super::BeginPlay();

    UPlayerAnimInstance* AnimInstance =
        Cast<UPlayerAnimInstance>(this->GetMesh()->GetAnimInstance());
    AnimInstance->SetIsDead(false);

    m_GM->AddPlayersToArray(this);
}
```

Рисунок 2.11 – Реалізацію методу BeginPlay класу MyPlayerCharacter

Оскільки BeginPlay – віртуальний метод, спочатку викликаємо батьківський метод через Super::. Отримуємо для вказівника AnimInstance посилання на аніматор та встановлюємо стан IsDead на false.

Оскільки ми отримали посилання на GameMode в батьківському класі, то тут викликаємо на ньому метод для додавання поточного гравця до списку. Після того як здоров'я гравця досягне нуля – GameMode викличе на ньому метод Respawn. На рисунку 2.12 зображено реалізацію метода Respawn класу MyPlayerCharacter.

```
void AMyPlayerCharacter::Respawn()
{
    Super::Respawn();

    m_StatComponent->ReduceHealth();

    m_bWasHit = false;

    GetCurrentAnimInstance()->SetIsDead(false);
    GetPlayerControlelr()->SetIsAttacking(false);
    GetPlayerControlelr()->ResetIsDodging();

}
```

Рисунок 2.12 – Реалізація метода Respawn класу MyPlayerCharacter

Respawn в MyBaseCharacter є віртуальним, тому спочатку викликається батьківський метод, потім реалізується логіка поточного класу. m_StatComponent->ReduceHealth() відповідає за зв'язок з UI та, не зважаючи на назву, оновлює стрічку здоров'я. m_bWasHit встановлюється значення false, щоб забезпечити можливість переміщення. Далі встановлюється значення IsDead аніматора для коректного опрацювання анімацій та скидаються атрибути IsAttacking, IsDodging, з тією ж метою що і m_bWasHit.

На рисунку 2.13 зображено реалізацію методу HandleHit класу MyPlayerCharacter.

```cpp
void AMyPlayerCharacter::HandleHit(AMyBaseCharacter* Attacker)
{
    if(m_StatComponent)
        m_StatComponent->ReduceHealth();

    if (m_bIsDead)
    {
        GetCurrentAnimInstance()->SetIsDead(true);
        GetCurrentAnimInstance()->OnMontageBlendingOut.AddDynamic(this,
            &AMyPlayerCharacter::OnDeathMontageBlendOut);

        GetCurrentAnimInstance()->Montage_Play(m_DeathMontage);

        return;
    }

    m_bWasHit = true;

    if(m_HitMontage)
        this->GetMesh()->GetAnimInstance()->Montage_Play(m_HitMontage);

    if (GetPlayerControlelr())
    {
        AMyPlayerController* ThisController = GetPlayerControlelr();
        ThisController->ResetIsDodging();
        ThisController->SetIsAttacking(false);
    }
}
```

Рисунок 2.13 – Реалізація методу HandleHit класу MyPlayerCharacter

Даний метод відповідає за опрацювання попадання по персонажу. Як завжди, при роботі з вказівниками, перевіряємо не nullptr. Далі через m_StatComponent оновлюємо стрічку здоров'я. Перевіряємо чи персонаж «мертвий» – встановлюємо для аніматора відповідне значення. Підписуємо на «затухання» монтажу метод для роботи з UI та програємо анімацію. Якщо ж ерсонаж «живий» встановлюємо значення m_bWasHit на true, програємо анімацію та скидаємо інші значення пов'язанні з контролем персонажа.

2.4 Проектування і опис інтерфейсу користувача

Глобально інтерфейс користувача можна розділити на категорії: головне меню та ігровий процес.

Головне меню.

На рисунку 2.14 зображено головне меню гри.

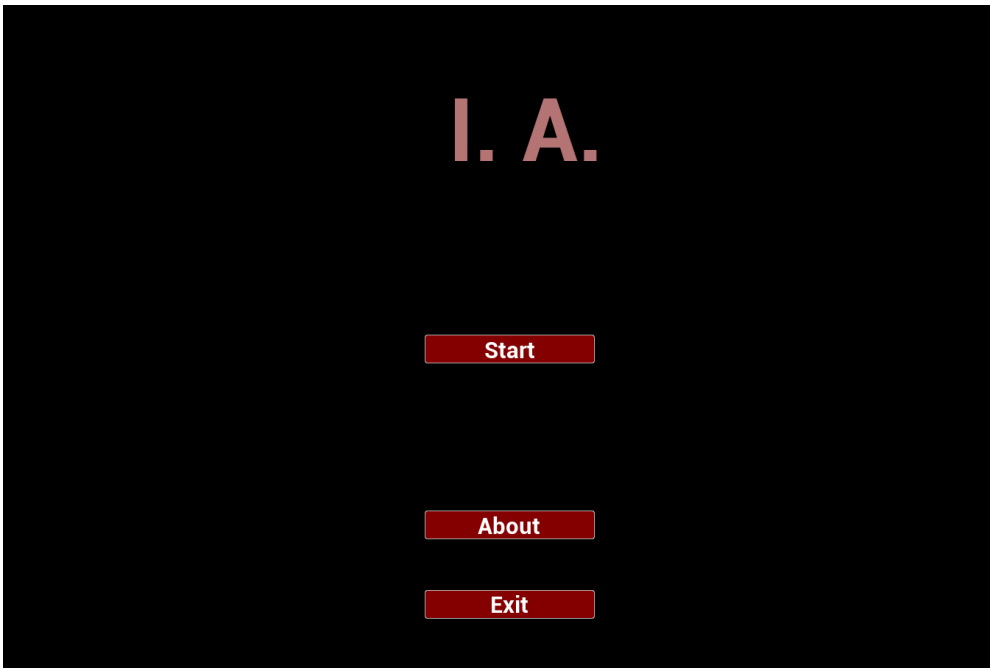

Рисунок 2.14 – головне меню гри

Згідно малюнка 2.14, головне меню складається з трьох кнопок та одного надпису. Розглянемо їх детальніше.

Кнопка «Exit» – відповідає за вихід з гри. На події OnClicked кнопки, прив’язано логіки виходи з гри за допомогою blueprint системи. На рисунку 2.15 зображено логіку виходу з гри кнопки «Exit», реалізованої на blueprint.

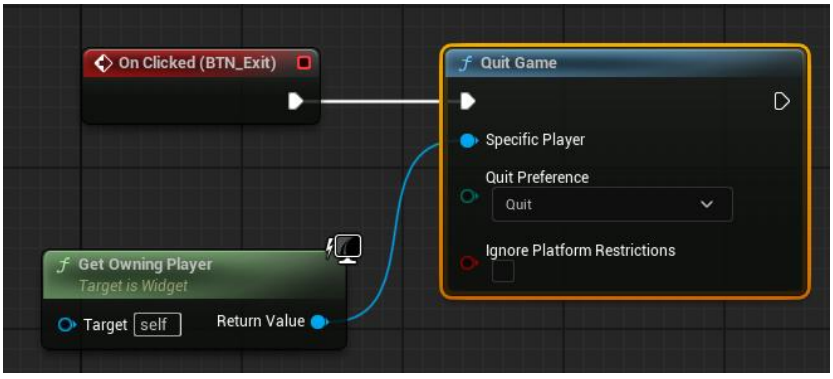

Рисунок 2.15 - Логіка виходу з гри кнопки «Exit», реалізованої на blueprint

Дана логіка викликає QuitGame – вбудований в рушій метод – на поточному персонажі, що і призводить до виходу з гри.

Кнопка «About» відповідає за перехід на сторінку «Про автора». На рисунку 2.16 зображено сторінку «Про автора».

Рисунок 2.16 – Сторінка «Про автора»

На рисунку 2.17 зображено логіку кнопки «About».

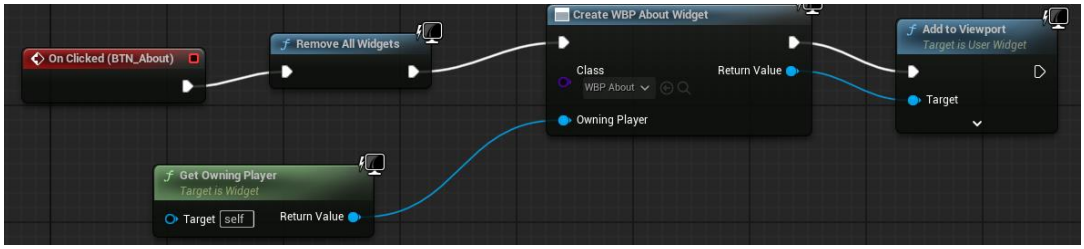

Рисунок 2.17 – Логіка кнопки «About»

Remove All Widgets – очищає екран від усіх інших елементів.

Create WBP About Widget – створює новий елемент UI, в даному випадку сторінку «Про автора».

Add to Viewport – виводить на екран елемент UI.

Така ж логіка і у кнопки «Back» на сторінці «Про автора», але цього разу у Create Widget вибирається «Main Menu».

Кнопка «Start» відповідає за перехід до ігрового процесу. На рисунку 2.18 зображено логіку кнопки «Start».

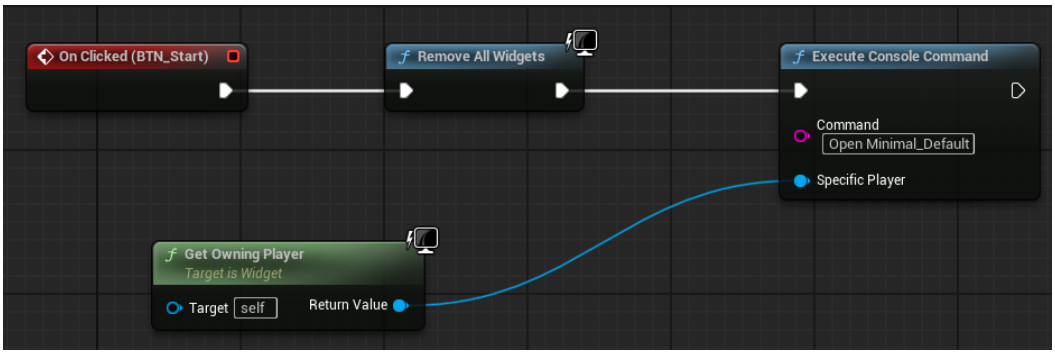

Рисунок 2.18 – Лоігка кнопки «Start»

Кнопка знову ж таки очищає всі віджети та запускає команду на відкриття заданого рівня.

Ігровий процес.

На рисунку 2.19 зображено стрічку здоров’я гравця.

Рисунок 2.19 – Стрічка здоров’я гравця

Дана стрічка оновлюється за допомогою методу ReduceHealth в StatComponent та за допомогою логіки на самій стрічці. На рисунку 2.20 зображено логіку оновлення стрічки.

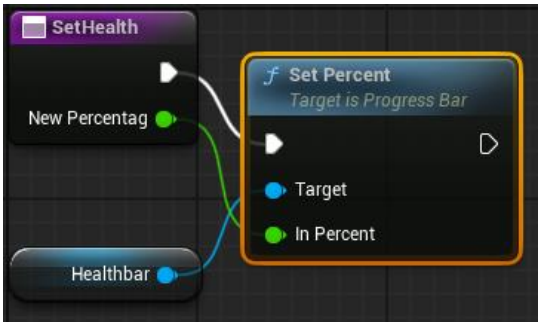

Рисунок 2.20 – Логіка оновлення стрічки

На рисунку 2.21 зображено логіку методу ReduceHealth.

```
void UMyStatComponent::ReduceHealth()
{
    OnHealthUpdateDelegate.Broadcast(GetPercentage("Health"));
}
```

Рисунок 2.21 – Логіка методу ReduceHealth

Логіка полягає в тому, що метод повідомляє через делегат про те, що метод викликаний і передає новий відсоток від загального здоров'я. В Event graph самого персонажа на цей івент прикраплюється виклик та передача методу стрічки. На рсиунку 2.22 зображено логіку івента оновлення стрічки.

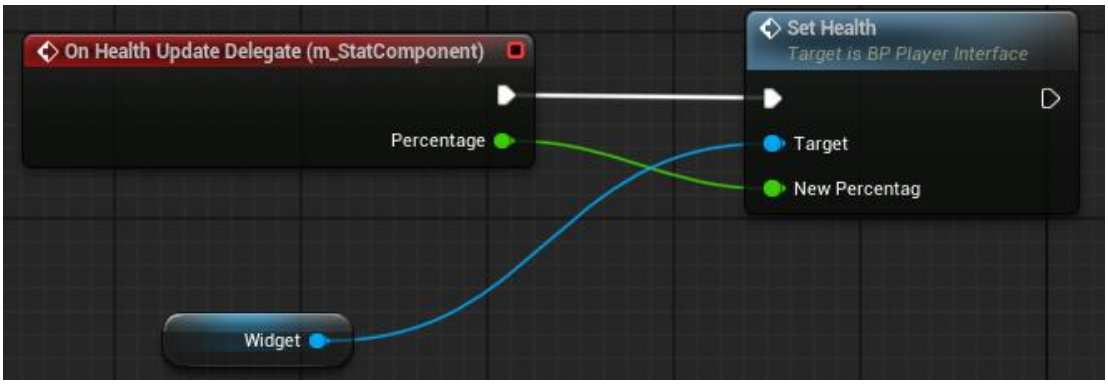

Рисунок 2.22 – Логіка івенту оновлення стрічки

Інвентар складається з декількох елементів: самого відображення інвентарю та кнопки предмету.

Інвентар являє собою коробку з заднім фоном, який має логіку динамічного заповнення себе контентом – кнопками предметів.

Кнопки предметів – кнопка з назвою прадмета та іконка, з логікою отримання даних і опрацюванням натискання.

На рисунку 2.23 зображено інвентар з кнопками предметів під час ігрового процесу.

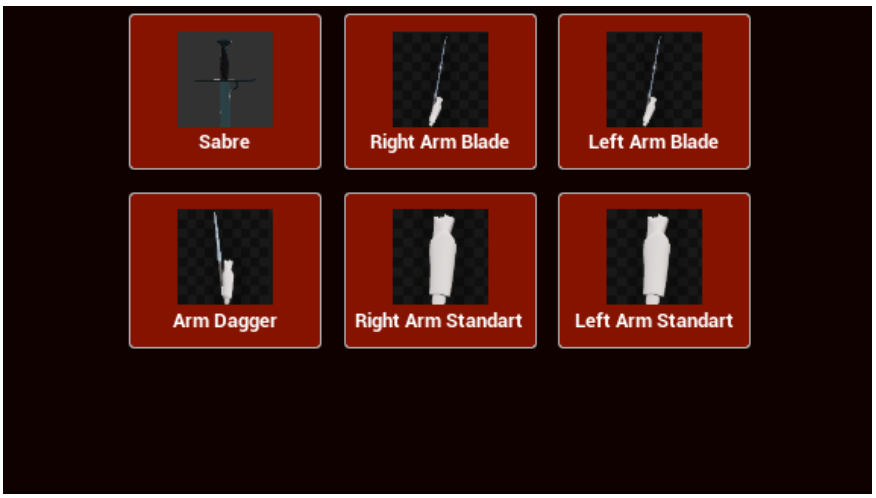

Рисунок 2.23 – Інвентар з кнопками предметів під час ігрового процесу

На рисунку 2.24 зображено логіку отримання назви кнопки предмету.

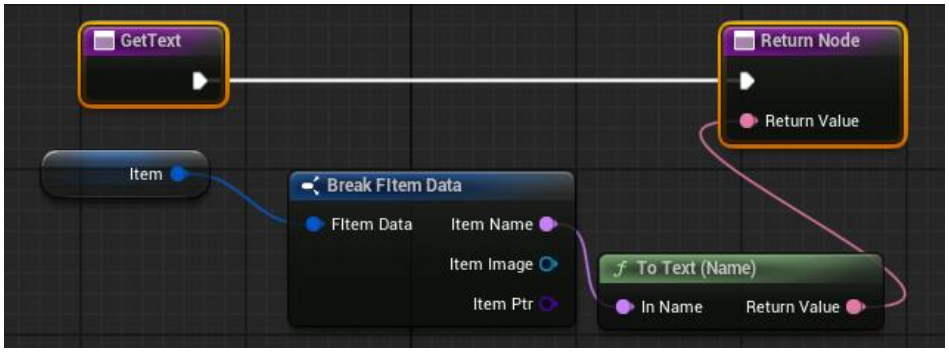

Рисунок 2.24 – Логіка отримання назви кнопки предмету

На рисунку 2.25 зображено лоігку отримання іконки предмету.

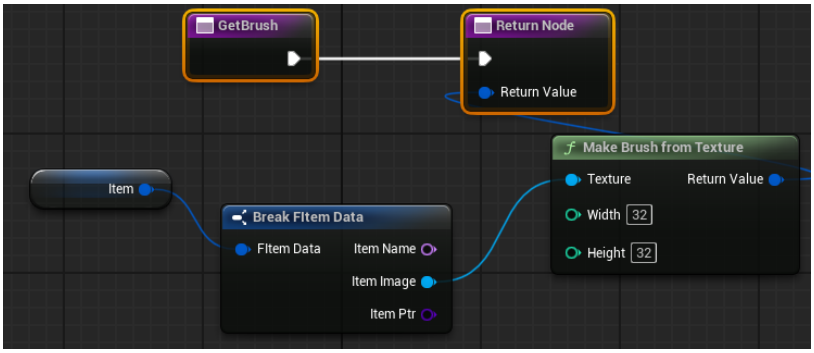

Рисунок 2.25 – Логіка отримання іконки предмету

На рисунку 2.26 зображено логіку заповнення інвентарю кнопками предметів.

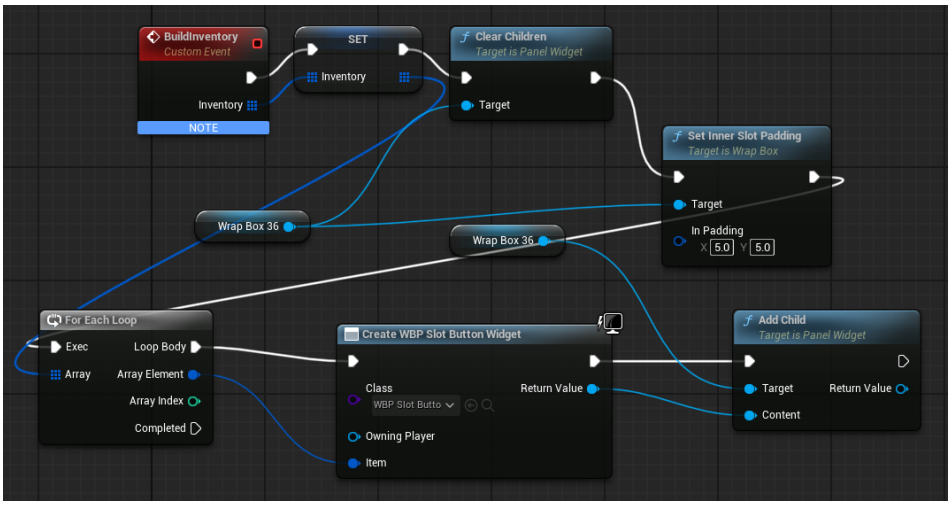

Рисунок 2.26 – Логіка заповнення інвентарю кнопками предметів

Спочатку передається інвентар предметів, далі встановлюються відступи між ними та проходяться по кожному предмету і створюється відповідний віджет з заданням назви та іконки, в кінці кнопка встановлюється як «дитина» до інвентарю.

2.5 Опис файлової структури програми

Після збірки повноцінної збірки, файли гри мають наступну струкутуру:

1. Engine – зберігає ресурси рушія, містить компоненти Unreal Engine, які є ключовими для роботи.

1.1 Engine/Plugins – містить опціональні плагіни та модулі, які є використовуються у проекті.

1.2 Engine/Binaries – містить dll-бібліотеки, які використовують у проекті.

1.3 Engine/Config – конфігураційні файли рушія, які визначають поведінку проекту.

1.4 Engine/Content – містить файли пов’язані з рендером проекту та графічного інтерфейсу.

1.5 Engine/Extras – містить додаткові утиліти та інструменти.

2. GameName – папка з ресурсами проекту.

2.1 GameName/Binaries – містить основний виконуваний застосунок та dll бібліотеки.

2.2 GameName/Content – містить увесь контент: 3D моделі, анімації тощо.

На рисунку 2.27 зображено файлову структуру гри після повноцінної збірки проекту.

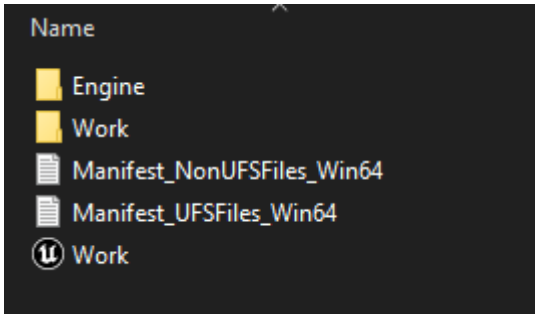

Рисунок 2.27 – Файлова структура гри після повноцінної збірки проекту

На рисунку 2.28 зображено вміст папки Engine.

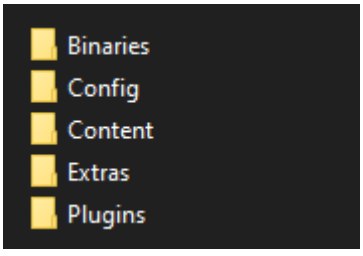

Рисунок 2.28 – Вміст папки Engine

На рисунку 2.29 зображено вміст папки Work/Binaries/Win64.

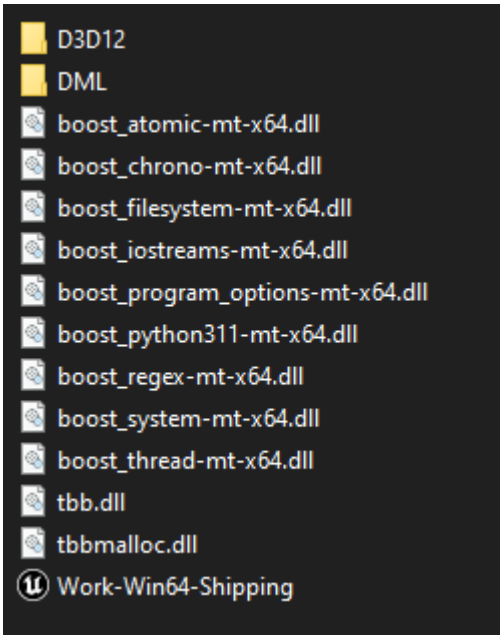

Рисунок 2.29 – Вміст папки Work/Binaries/Win64.

Графічне представлення класів зображено у додатку «Діаграма файлової структури».

2.6 Тестування програми і результати її виконання

Для тестування використовуються наступні методи:

– unit-тестування за допомогою плагіна Unreal Engine – Test Automation;

– тестування «чорною скринькою» для елементів, які представляють лише візульну репрезентацію певних подій.

Unit-тестування.

Тестування відбувається в наступні етапи:

1. Підключити плагін в редакторі.
2. Підключити модулі UnrealEd, AutomationController в Name.Build.cs
3. Створити новий .cpp файл для тестів, наприклад якщо тестується функціонал базового персонажу, то доцільно назвати його MyBaseCharacterTests.cpp.

4. Підключити необхідні заголовочні файли: Misc/AutomationTest.h, Tests/AutomationEditorCommon.h – відповідають за самі тести та підключити заголовочні файли класів, які тестуються.

5. Написати тести.

На рисунку 2.30 наведено тест для перевірки нанесення шкоди базового класу MyBaseCharacter.

```
IMPLEMENT_SIMPLE_AUTOMATION_TEST(AMyBaseCharacter_OverlapDamageTest, "MyGame.Unit.MyBaseCharacter.OverlapDamage",
    EAutomationTestFlags::EditorContext | EAutomationTestFlags::ProductFilter)

bool AMyBaseCharacter_OverlapDamageTest::RunTest(const FString& Parameters)
{
    UWorld* World = FAutomationEditorCommonUtils::CreateNewMap();
    TestNotNull(TEXT("World created"), World);

    AMyBaseCharacter* Target = World->SpawnActor<AMyBaseCharacter>();
    TestNotNull(TEXT("Target created"), Target);
    AMyBaseCharacter* Attacker = World->SpawnActor<AMyBaseCharacter>();
    TestNotNull(TEXT("Attacker created"), Attacker);

    UMyStatComponent* AttStat = Cast<UMyStatComponent>(Attacker->GetStatComponent());
    AttStat->AddStat("Damage", -25, -100, 0);

    TestTrue(TEXT("No damage stat"), Attacker->GetStatComponent()->GetStatValue("Damage") == -25);

    Target->GetStatComponent()->AddStat("Health", 100, 0, 100);

    TestTrue(TEXT("No health stat"), Target->GetStatComponent()->GetStatValue("Health") == 100);

    Target->SetIsInvincible(false);
    Target->SetIsDead(false);

    auto* DummyComp = NewObject<UCapsuleComponent>(Attacker);
    TestNotNull(TEXT("DummyComp created"), DummyComp);
    DummyComp->ComponentTags.Add(FName("WeaponCollider"));
    DummyComp->RegisterComponentWithWorld(World);
    Attacker->AddInstanceComponent(DummyComp);
    DummyComp->AttachToComponent(Attacker->GetRootComponent(), FAttachmentTransformRules::KeepWorldTransform);

    TestTrue("Owner is Attacker", DummyComp->GetOwner() == Attacker);

    Target->OnCapsuleOverlap(nullptr, Attacker, DummyComp, 0, true, FHitResult());

    float H = Target->GetStatComponent()->GetStatValue("Health");

    TestTrue(TEXT("Health reduced after overlap"), H < Target->GetStatComponent()->GetStatMaxValue("Health"));

    return true;
}
```

Рисунок 2.30 – Тест для перевірки нанесення шкоди базового класу MyBaseCharacter

IMPLEMENT_SIMPLE_AUTOMATION_TEST – макрос для оголошення створення тесту.

AMyBaseCharacter_OverlapDamageTest – назва тесту.

MyGame.Unit.MyBaseCharacter.OverlapDamage – як буде відображатися, в яких каталогах та підкаталогах, тест у рушії.

EAutomationTestFlags::EditorContext|EAutomationTestFlags::ProductFilter – вказують, що тест запускається у редакторі та включений в пошук.

bool AMyBaseCharacter_OverlapDamageTest::RunTest(const FString& Parameters) – оголошення методу запуску тесту, RunTest – головна функція запуску.

UWorld* World = FAutomationEditorCommonUtils::CreateNewMap() – створення тестового світу, оскільки деякі механіки потребують його.

TestNotNull(TEXT("World created"), World) – перевірка чи світ дійсно створився, якщоне створився – фейлить тест.

AMyBaseCharacter* Target = World->SpawnActor<AMyBaseCharacter>() – створюється тестовий персонаж та перевіряється на Null як і світ.

UMyStatComponent* AttStat = Cast<UMyStatComponent>(Attacker->GetStatComponent()) – створюється компонент з параметрами.

AttStat->AddStat("Damage", -25, -100, 0) – задаються значення.

TestTrue(TEXT("No damage stat"), Attacker->GetStatComponent()->GetStatValue("Damage") == -25) – перевірка на наявність значення шкоди, якщо немає – фейлить тест.

Далі створюється колайдер, реєструється в світі, прикріплюється до актора та викликається метод OnCapsuleOverlap – який і є суттю даного тесту. В кінці перевіряємо чи зменшилося здоров’я в актора.

Для перевірки результату тесту виконуються наступні дії:

1. В рушії відкривається Tools -> Test Automation.
2. У відкритому вікні знаходимо свої створені тести за допомогою колеса прокрутки або пошуку.
3. Вибираємо результати яких тести хочемо перевірити.
4. Натискаємо кнопку Start/Stop Tests.
5. Чекаємо результату.

На рисунку 2.31 зображено вікно Test Automation.

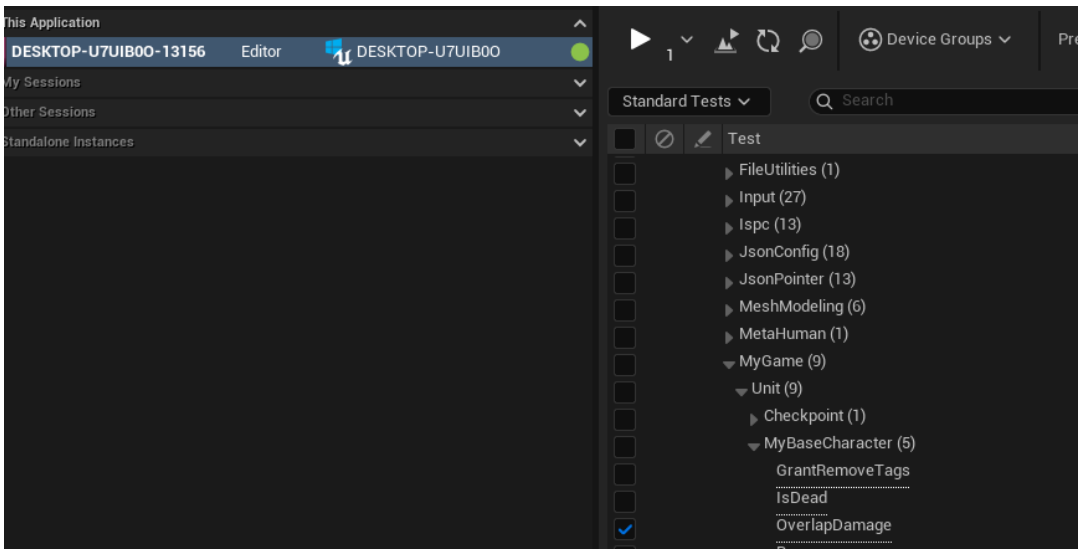

Рисунок 2.31 – Вікно Test Automation

Після виконання тесту отримуємо вікно з результатами тестування зображене на рисунку 2.32. У ньому буде повідомлене чи тести пройшли успішно чи ні, також який саме «підтест» провалився.

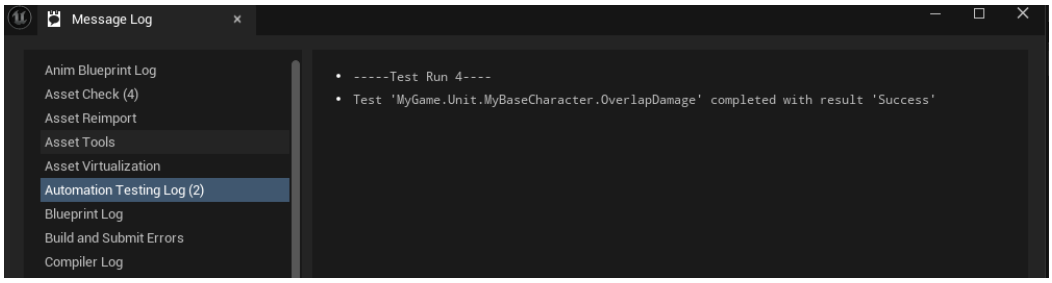

Рисунок 2.32 - Вікно з результатами тестування

Також маємо кольорове представлення пройдених та не пройдених тестів як зображено на рисунку 2.33.

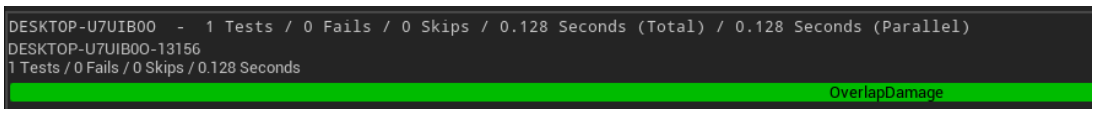

Рисунок 2.33 - Кольорове представлення пройдених та не пройдених тестів

Даним тестуванням було перевірено:

- встановлення гравцю позиції відродження;
- коректність встановлення «мертвого» стану персонажам;
- коректність відродження персонажів;
- нанесення шкоди персонажу;
- роботу з тегами персонажів, дарування, перевірка, вилучення;
- додавання та забирання пошкоджених персонажів в масив;
- зміна модульних частин гравця;
- коректність роботи колайдерів на модульних частинах гравця;
- ініціалізація скелетних мешів гравця.

На рисунку 2.34 зображено результат виконання тестів.

Рисунок 2.34 – Результат виконання тестів

Тестування «чорною скринькою».

Тестування «чорною скринькою» застосовуються для швидкої перевірки елементів, які не потребують складного тестування, наприклад первірка реакції кнопок на натиск та чи коректно вони пересилають користувача.

Даним тестуванням було перевірено:

- роботу виходу з гри з головного меню;
- перехід на сторінку «Про автора» з головного меню;
- перехід зі сторінки «Про автора» до головного меню;
- запуск гри кнопкою «Start»;
- відображення UI гравця;
- відображення інвентаря гравця;
- зміна модульних частин гарвця за допомогою інвентарю;
- робота анімацій персонажів.

Після завершення всіх проведених тестів можна зробити висновок, що всі системи працюють стабільно, демонструючи очікувану функціональність та ефективність, і не призводять до збоїв або помітних несправностей у роботі гри. У такому випадку можна приступати до фінального етапу створення інсталятора та інструкцій з використання програмного комплексу.

3 СПЕЦІАЛЬНИЙ РОЗДІЛ

3.1 Інструкція з інсталяції програмного забезпечення

Програма розповсюджується у вигляді виконавчого застосунку – інсталятора. Для її запуску необхідно:

1. Завантажити інсталятор.
2. Запустити його та виконати дії, які він вимагає.
3. Натиснути кнопку для встановлення.
4. Дочекатися завершення інсталяції.
5. Запустити гру.

3.2 Інструкція з використання тестових наборів

Для тестування за допомогою Unit-тестів:

1. Встановити рушій.
2. Встановити плагін Test Automation.
3. Скомпілювати проект.
4. Вибрати тести у вікні Test Automation.
5. Дочекатися результату.

Для тестування «чорною скринькою»:

1. Встановити гру.
2. Зайти в гру.
3. Перевірити кнопку «About».
4. Перевірити кнопку «Back» на сторінці «Про автора».
5. Натиснути кнопку «Start».
6. Перевірити переміщення персонажа клавішами: W, A, S та D.
7. Перевірити роботу інвентарю клавішею «I».

8. Перевірити чи змінюються модульні частини на персонажі натиснувши на кнопку предмету в інвентарі.

9. Закрити інвентар повторно натиснувши клавішу «I»

10. Перевірити чи працюють ухиляння клавішею «L Shift».

11. Перевірити зациклення на персонажі клавішею «Middle Mouse Button».

12. Перевірити чи працюють напрямкові ухиляння під час зациклення.

13. Перевірити нанесення, ліва та права клавіші мишки, та отримання шкоди від ворогів та по ним.

14. Перевірити чи працює система відродження опустивши здоров’я до нуля.

15. Перевірити клавішу «Exit» в інвентарі.

3.3 Інструкція з експлуатації програмного комплексу

Для коректної роботи гри необхідно щоб система задовільняла мінімальні параметри:

- CPU: Intel Core i3 – 4160;
- RAM: 500 MB;
- GPU: GeForce GTX 1050Ti.
- HDD/SSD: 2GB.

Також, мають бути встановленні наступні компоненти:

- DirectX 11;
- Microsoft Visual C++ 2015-2022 Redistributable.

4 ЕКОНОМІЧНИЙ РОЗДІЛ

Метою економічної частини кваліфікаційної роботи бакалавра є здійснення економічних розрахунків, спрямованих на визначення економічної ефективності проєкту «Розробка комп'ютерної гри "Impartial Automaton" з використанням рушія Unreal Engine та мови C++».

4.1 Визначення стадій технологічного процесу та загальної тривалості проведення НДР

Для визначення загальної тривалості проведення НДР зведено таблицю 4.1, в яку занесено дані витрат часу по окремих операціях технологічного процесу.

Таблиця 4.1 - Середній час виконання НДР та стадії (операції) технологічного процесу

№ п/п	Назва операції (стадії)	Виконавець	Середній час виконання операції, год.
1	2	3	4
1	Формулювання задачі	керівник	24
2	Аналіз та проектування	керівник	48
3	Розробка основних механік	програміст	36
4	Розробка 3D моделей та анімацій	Графічний дизайнер, аніматор	120
5	Розробка контенту	Сценарист, геймдизайнер	72
6	Реалізація контенту	програміст	120

Продовження таблиці 4.1

1	2	3	4
7	Тестування	Тестувальник	72
8	Реліз	Керівник	8
Разом			500

4.2 Визначення витрат на оплату праці та відрахувань на соціальні заходи

Оплата праці - грошовий вираз вартості і ціни робочої сили, який виступає у формі будь-якого заробітку, виплаченого керівником підприємства найманому працівникові за виконану роботу. Заробітна плата працівника залежить від кінцевих результатів його роботи, регулюється податками і максимальними розмірами не обмежується.

Основна заробітна плата розраховується за формулою 4.1:

$$З_{осн} = Т_с * К_г \quad (4.1)$$

де, $Т_с$ – тарифна ставка, грн.;

$К_г$ – кількість відпрацьованих годин.

Рекомендовані тарифні ставки: керівник проєкту – 150 грн./год., програміст – 250 грн./год., графічний дизайнер/аніматор – 150 грн./год., сценарист – 150 грн./год., геймдизайнер – 150 грн./год., тестувальник – 130 грн./год.

Основна заробітна плата становить:

– Керівника: $З_{осн1} = 150 * 80 = 12\ 000$ грн.;

– Програміста: $З_{осн2} = 250 * 156 = 39\ 000$ грн.;

– Графічного дизайнер/аніматор: $З_{осн3} = 150 * 120 = 18\ 000$ грн.;

– Сценарист та геймдизайнер: $З_{осн4} = (150 * 72)*2 = 21\ 600$ грн.;

– Тестувальник: $З_{осн5} = 130 * 72 = 9\ 360$ грн.

Сумарна основна заробітна плата становить:

$$З_{осн} = 12\ 000 + 39\ 000 + 18\ 000 + 21\ 600 + 9360 = 99\ 960 \text{ грн.}$$

Додаткова заробітна плата становить 10-15 % від суми основної заробітної плати та обчисоюється за формулою 4.2:

$$З_{дод} = З_{осн} * К_{допл} , \quad (4.2)$$

де, $К_{допл}$ – коефіцієнт додаткових виплат працівникам 0,1-0,15.

Отже, додаткова заробітна плата становить:

1. Керівника: $З_{дод1}$ =12 000 * 0,1 = 1 200 грн.;
2. Програміста: $З_{дод2}$ = 39 000 * 0,1 = 3 900 грн.;
3. Графічного дизайнер/аніматор: $З_{дод3}$ =18 000 * 0,1 = 1 800 грн.;
4. Сценарист та геймдизайнер: $З_{дод4}$ = (10 800 * 0,1) * 2 = 2160 грн. кожен;
5. Тестувальник: $З_{дод5}$ = 9 360 * 0,1 = 936 грн.

Сумарна додаткова заробітна плата становить:

$З_{дод}$ = 1 200 + 3 900 + 1 800 + 2160 + 936 = 9 996 грн.

Звідси загальні витрати на оплату праці визначаються за формулою 4.3:

$$В_{о.п.} = З_{осн} + З_{дод} \quad (4.3)$$

$$В_{о.п.} = 99\,960 + 9\,996 = 109\,956 \text{ грн.}$$

Необхідно визначити відрахування на соціальні заходи:

– фонд страхування на випадок безробіття – 1,6 %;

– фонд по тимчасовій втраті працездатності – 1,4 %;

– пенсійний фонд – 33,2 %;

– внески на страхування від нещасного випадку на виробництві та професійного захворювання - 1,4%.

Загальна сума зазначених відрахувань становить 37,6 %.

Отже, сума нарахувань на заробітну плату буде становити згідно формули 4.4:

$$В_{с.з.} = ФОП * 0,376, \quad (4.4)$$

Де ФОП – фонд оплати праці, грн.

$$В_{с.з.} = 109\ 956 * 0{,}376 = 41\ 343{,}46 \text{ грн.}$$

Проведені розрахунки витрат на оплату праці зведено у таблиці 4.2.

Таблиця 4.2 – Зведені розрахунки витрат на оплату праці

№ п/п	Категорія працівника	Основна заробітна плата, грн			Додаткова заробітна плата, грн.	Нарах ув. На ФОП, грн.	Всього витрати на оплату праці, грн.
		Тарифна ставка, грн.	К-сть відпрацьов. год.	Фактично нарах. з/пл., грн.			
1	Керівник	150	80	12 000	1 200	-	-
2	Програміст	250	156	39 000	3 900	-	-
3	Графічний дизайнер/аніматор	150	120	18 000	1 800	-	-
4	Сценарист та геймдизайнер	130	72	21 600	2160	-	-
5	Тестувальник	130	72	9 360	936	-	-
Разом				99 960	9 960	41 343,46	151 263, 46

Загальні витрати на оплату праці становлять 151 263, 46 грн.

4.3 Розрахунок матеріальних витрат

Матеріальні витрати визначаються за наступною формулою (4.5):

$$М_{в.і.} = q_i * p_i, \quad (4.5)$$

де, q_i – кількість витраченого матеріалу і-го виду;

p_i – ціна матеріалу і – го виду.

Звідси, загальні матеріальні витрати можна визначити за формулою 4.6:

$$З_{м.в.} = \sum М_{в.і.}, \quad (4.6)$$

Проведені розрахунки занесемо у таблицю 4.3.

Таблиця 4.3 – Зведені розрахунки матеріальних витрат

№ п/п	Найменування матеріальних ресурсів	Од. виміру	Факт. витрачено матеріалів	Ціна 1-ці, грн.	Загальна сума витрат, грн
1	2	3	4	5	6
1	Процессор Ryzen 7 3700X	шт.	1	4 500	4 500
2	Відеокарта GeForceRTX 4060	шт.	1	12 000	12 000
3	Оперативна пам’ять 16 GB	шт.	4	2 038	8 152
4	Материнська плата Gigabyte B450M	шт.	1	2 867	2 867
5	БЖ GIGABYTE GP-P850GM 850W	шт.	1	3 360	3 360
6	Вінчестер 500 GB	шт.	1	650	950
7	SSD 240 GB SATA III	шт.	1	994	994
8	SSD M.2 512 GB	шт.	1	2 200	2 200
9	Корпус	шт.	1	3 500	3 500
10	Монітор 23.8″	шт.	1	6 500	6 500

Продовження таблиці 4.3.

1	2	3	4	5	6
11	Монітор 21.5″	шт.	1	3 000	3 000
12	Клавіатура	шт.	1	2 500	2 500
13	Комп’ютерна миша	шт.	1	1 500	1 500
14	Графічний планшет	шт.	1	1 500	1 500
15	Графічний редактор	коп.	1	1 420	1 420
16	Ігровий рушій	коп.	1	-	-
Разом					54 613

Загальна сума матеріальних витрат становить 54 613 грн.

4.4 Розрахунок витрати на електроенергію

Витрати на електроенергію 1-ці обладнання визначається за формулою:

$$З_е = W * T * S, \quad (4.7)$$

де, W – необхідна потужність, кВт;

T – кількість годин роботи обладнання;

S – вартість кіловат-години електроенергії.

Час роботи ПК над даним проєктом становить 500 годин, споживана потужність 0,7 кВт.год., вартість 1 кВт електроенергії – 7 грн.

$$З_е = 0,5 * 500 * 7 = 1\ 750 \text{ грн.}$$

4.5 Визначення транспортних затрат

Транспортні витрати слід прогнозувати у розмірі 8-10 % від загальної суми матеріальних затрат. Транспортні витрати розраховуються за формулою 4.8.

$$Т_в = З_{м.в.} * 0,08 \dots 0,1, \quad (4.8)$$

Отже, транспортні витрати будуть становити:

$$Т_в = 54\ 613 * 0,09 = 4\ 915,17 \text{ грн.}$$

4.6 Розрахунок суми амортизаційних відрахувань

Комп’ютери та оргтехніка належать до четвертої групи основних фондів. Мінімально допустимі строки їх використання 2 роки. Для визначення амортизаційних відрахувань застосовуємо формулу 4.9:

$$A = \frac{Б_в * Н_а}{150\%} * Т_А, \quad (4.9)$$

де, А – амортизаційні відрахування за звітній період, грн;

$Б_в$ – балансова вартість групи основних фондів на початок звітного періоду, грн.;

$Н_а$ – норма амортизації, 0,04%.

Для написання програми та її тестування використовувався ПК та оргтехніка із сумарною вартістю 54 613, тому:

$$А = \frac{54\ 613 * 0,04}{150} * 500 = 7282 \text{ грн}$$

4.7 Обчислення накладних витрат

Накладні витрати - це витрати, не пов'язані безпосередньо з технологічним процесом виготовлення продукції, а утворюються під впливом певних умов роботи по організації, управлінню та обслуговуванню виробництва. В залежності від організаційно-правової форми діяльності господарюючого суб’єкта, накладні витрати можуть становити 20 – 60 % від суми основної та додаткової заробітної плати працівників, обчислюються за формулою 4.10, де НВ – накладні витрати.

$$Н_в = В_{о.п.} * 0,2...0,6, \quad (4.10)$$

$$Н_в = 109\ 956 * 0,4 = 43\ 982,4 \text{ грн.}$$

4.8 Складання кошторису витрат та визначення собівартості НДР

Кошторис витрат являє собою зведений план усіх витрат підприємства на майбутній період виробничо-фінансової діяльності.

Результат проведених вище розрахунків зведено у таблицю 4.4.

Таблиця 4.4 – Кошторис витрат на НДР

Зміст витрат	Сума, грн	В % до загальної суми
Витрати на оплату праці	109 956	41,67%
Відрахування на соціальні заходи	41 343,46	15,67%
Матеріальні витрати	54 613	20,7%
Витрати на електроенергію	1 750	0,66%
Транспортні витрати	4 915,17	1,86 %
Амортизаційні відрахування	7282	2,76 %
Накладні витрати	43 982,4	16,67%
Собівартість	263 842,03	100%

Собівартість ($С_в$) НДР розрахуємо за формулою 4.11:

$$С_в = В_{0.л.} + В_{с.з.} + З_{м.в.} + З_е + Т_в + А + Н_в \qquad (4.11)$$

Отже, собівартість становить $С_в$ = 263 842,03 грн.

4.9 Розрахунок ціни НДР

Ціну НДР можна визначити за формулою 4.12:

$$Ц = С_в * (1 + Р_{рен.}) * (1 + ПДВ), \qquad (4.12)$$

Де: $С_в$ – собівартість виконання НДР;

$Р_{рен.}$ – рівень рентабельності;

ПДВ – ставка податку на додану вартість, 20%.

Ц = 263 842,03 * (1 + 0,3) * (1 + 0,2) = 411 593,57 грн.

4.10 Визначення економічної ефективності і терміну окупності капітальних вкладень

Ефективність виробництва – це узагальнене і повне відображення кінцевих результатів використання робочої сили, засобів та предметів праці на підприємстві за певний проміжок часу.

Прибуток розраховується за формулою 4.13:

$$П = Ц - С_в, \quad (4.13)$$

П = 411 593,57 - 263 842,03 = 147 751,54 грн.

Економічна ефективність ($Е_р$) полягає у відношенні результату виробництва до затрачених ресурсів і розраховується за формулою 4.11, де П – прибуток; $С_в$ – собівартість.

$$Е_р = \frac{П}{С_в}, \quad (4.14)$$

$Е_р$ = 147 751,54 / 263 842,03 = 0,56.

Поряд із економічною ефективністю розраховують (формула 4.15) термін окупності капітальних вкладень $Т_р$:

$$Т_р = 1/Е_р \quad (4.15)$$

Допустимим вважається термін окупності до 5 років. В даному випадку:

$Т_р$ = 1/0,56 = 1,8

Всі дані внесемо в зведену таблицю 4.5 техніко-економічних показників.

Таблиця 4.5 - Техніко-економічні показники

№ п/п	Показник	Одиниця виміру	Значення
1	2	3	4
1	Собівартість, грн.	грн.	246 688,89
2	Плановий прибуток, грн.	грн.	147 751,54
3	Ціна, грн.	грн.	411 593,57
4	Економічна ефективність	-	0,56
5	Термін окупності, рік	рік	1,8

Загальна вартість розробки комп’ютерної гри «Impartial Automaton» становить 411 593,57.

Зважаючи на високі показники економічної ефективності - 0,56, кошти, вкладені в проведення проектних робіт окупляться за 1,8 року.

5 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

5.1 Розрахунок системи штучного освітлення для приміщення, де здійснюється розробка комп'ютерної гри «Impartial Automaton»

При розробці комп’ютерної гри «Impartial Automaton» досить важливим є вимоги до освітлення приміщень,оскільки відомо, що тривала робота за комп’ютером та з документами при недостатньому рівні освітленості може призвести до значного перенапруження зору. Необхідно дотримуватися нступних вимог:

– природне освітлення має забезпечувати коефіцієнт природної освітленості (КПО) не нижче ніж 1,5%. Для регулювання рівня освітлення природним світлом бажано застосовувати жалюзі;

– робоче місце, обладнане ПК повинно бути розташоване так, щоб уникнути попадання в очі прямого сонячного світла;

– штучне освітлення приміщення має бути обладнане системою загального рівномірного освітлення;

– застосування світильників без розсіювачів та екрануючих сіток забороняється;

– рівень освітленості на робочому столі в зоні розташування документів має бути в межах 300–500 лк[2].

Призначене приміщення для розробки комп’ютерної гри має розміри: 4х4 метри та висоту 3,5 метри. Необхідно розрахувати систему штучного світлення.

Вихідні дані:

– площа S = 4 * 4 = 16 м^2;

– висота стелі H = 3,5 м;

– норма освітленості E = 400 лк (середнє значення);

– Тип світильників – LED-панелі, світловий потік = 4000 лм.

Для розрахунку загального світлового потоку скористаємося формулою 5.1.

$$Ф = E * S * коеф.висоти, \quad (5.1)$$

де: Ф – світловий потік;

E – освітленність (E = 400 лк);

S – площа приміщення (S = 16 м^2);

коеф.висоти (при 3,5 м = 2)[3].

$$Ф = 400 * 16 * 2 = 12\ 800 \text{ лм}$$

Для обрахунку кількості світильників використовуємо формулу 5.2.

$$N = \frac{E*S*K_з*z}{Ф_{одн}*n}, \quad (5.2)$$

де: E – освітленність (E = 400 лк);

S – площа приміщення (S = 16 м^2);

$K_з$ – запас освітленості ($K_з$ = 1,5);

$Ф_{одн}$ – світлоивй потік одного світильника ($Ф_{одн}$ = 4000 лм);

n – коефіцієнт використання світлового потоку (n = 0,6).

$$N = \frac{400 * 16 * 1{,}5 * 1{,}1}{4000 * 0{,}6} = 4{,}4$$

Отже, для економічного варіанту, можна використати 5 світильників, кожен з яких буде забезпечувати по 4000 лм.

Розміщувати світильники необхідно наступним чином:

– відстань між світильниками 1 м;

– відстань світильників від стін 1 м;

– вікно, розташоване праворуч від персонального комп'ютера, буде забезпечувати додаткове освітлення.

На рисунку 5.1 зображено план розміщення світильників у приміщення розробки комп'ютерної гри.

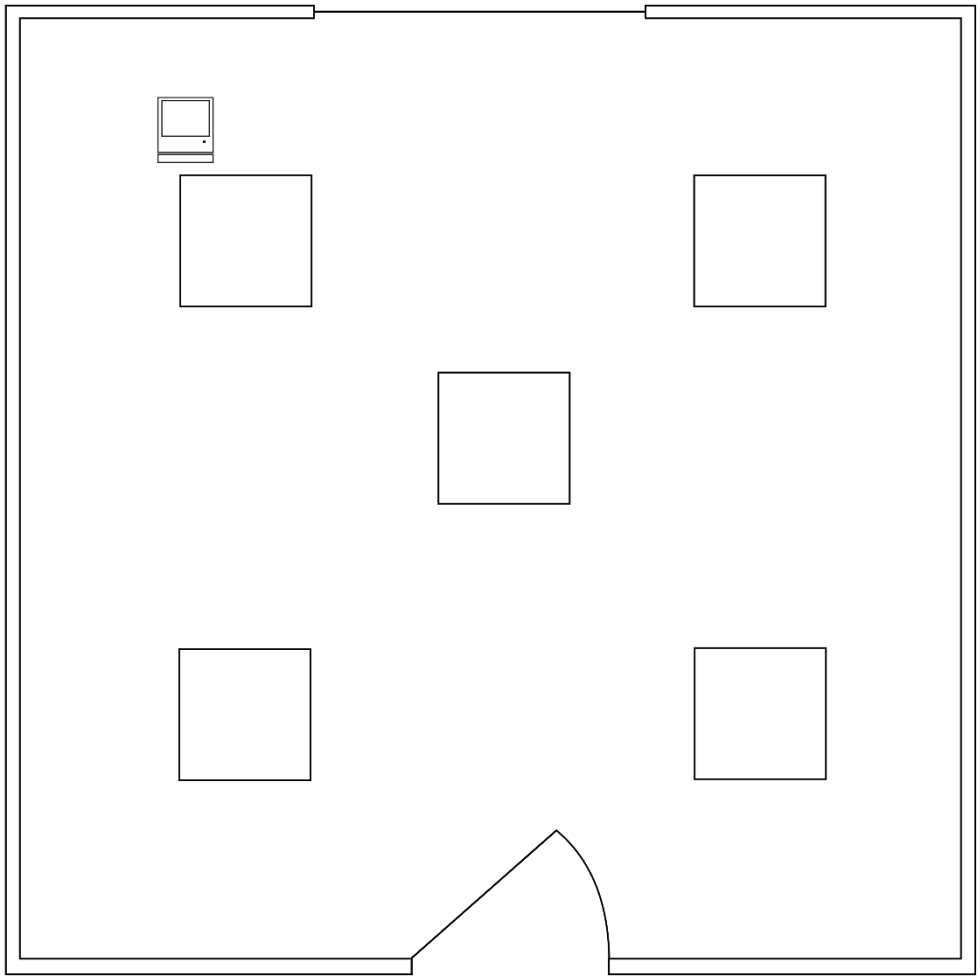

Рисунок 5.1 - План розміщення світильників у приміщення розробки комп'ютерної гри

5.2 Небезпечні і шкідливі виробничі фактори, джерелом яких є комп’ютер. Засоби захисту

При роботі з комп’ютером можна виділити наступні групи небезпечних та шкідливих факторів: фізичні, ергономічні та псиемоційні.

Фізичні фактори.

– Електромагнітне випромінювання (ЕМП). Персональний комп’ютер (особливо монітор) є джерелом електростатичного поля, низько-, наднизько- та високочастотного ЕМП (2 Гц–400 кГц), а також оптичного (УФ, ІЧ, видимого) і навіть рентгенівського випромінювання. Поглинання цих полів і випромінювань викликає напругу ЦНС: головні болі, депресію, безсоння,

стрес. Низькочастотне ЕМП може сприяти шкірним захворюванням, хворобам серця та органів травленняКрім того, бічні та задні стінки монітора і системного блоку є джерелами такої радіації, що особливо небезпечно при тривалій роботі [4]. Для захисту використовують екрани-фільтри: вони зменшують статичні поля та УФ/рентген-складові, знижуючи навантаження на організм [5].

– Шум і вібрації. Робота вентиляторів охолодження, принтерів та іншої офісної техніки створює рівні шуму. Хоча звичайний шум ПК невисокий, постійне фонове гудіння може викликати втомленість, підвищувати нервову напругу та знижувати концентрацію, а при тривалому впливі – ризик порушень слуху.

– Неналежне освітлення. Недостатня чи надмірна (висококонтрастна) освітленість, а також відблиски на екрані призводять до перенапруження зору, сльозотечі, головного болю. Відблиски є однією з найпоширеніших причин стомлюваності очей [6].

– Температурно-вологісні умови. Системний блок і монітор виділяють тепло. Якщо приміщення перегріте або холодне (поза оптимальним діапазоном 18–24 °C), це знижує працездатність, викликає загальну втому та дискомфорт (також висока вологість може стимулювати ріст бактерій у запиленому середовищі). Треба дотримуватися норм мікроклімату та забезпечувати вентиляцію.

– Пил та статична електрика. Монітор накопичує пил через статичні поля. Пилові частинки і бактерії в повітрі можуть спричиняти алергії, подразнення слизових, загострення астми чи дерматит. Це підтверджено спостереженнями, що запиленість довкола ЕОМ часто викликає шкірні й респіраторні реакції. Антистатичні покриття екрану і регулярне провітрювання допомагають знизити пилові концентрації.

– Електробезпека. Нестабільне живлення чи несправна проводка можуть становити ризик ураження електричним струмом і замикань.

Перевищення напруги або грубі пошкодження кабелів можуть спричинити перебої в роботі обладнання і навіть пожежу.

Ергономічні та психоемоційні фактори.

– Зорове навантаження. Тривала фокусування на дисплеї викликає сухість, роздвоєння, «затуманення» зору і біль в очах. 40–70% постійних користувачів комп'ютерів мають симптоми так званого комп'ютерного зорового синдрому (болі, печіння, відчуття піску в очах). Постійна робота на близькій відстані і висока яскравість екрана посилюють ці ефекти. Зрештою це може сприяти зниженню гостроти зору або поглибленню наявних аномалій (міопії тощо) [7].

– Статичні пози, малорухомість. Оператор ПК довго сидить у одній позі, задіюючи одні й ті ж м'язи шиї, плечей та спини. Це призводить до м'язового напруження, спазмів і порушення постави. У результаті часто розвиваються болі в шиї, попереку, шийному відділі хребта; можливі сколіотичні зміни й остеохондроз. Доведено, що тривале сидіння спричиняє застій крові в малому тазі, радикуліт і може зменшувати кровообіг у кінцівках [8].

– Монотонність та психологічне навантаження. Одноманітна робота перед монітором без змін діяльності веде до емоційного вигоряння. Фіксоване зорове та інтелектуальне навантаження без змін стимулює стрес, втому та апатію. Наприклад, відсутність рухливості та соціальних відволікань часто спричиняє дратівливість, втрату мотивації та синдром хронічної втоми. Систематичний стрес може викликати порушення сну, зниження імунітету та психоемоційні розлади.

Засоби захисту.

– Технічні засоби: Використання захисних екранів-фільтрів на моніторах (особливо ЕПТ) знижує вплив ШХЧ та статичних полів. Антиблікові покриття монітора та меблів запобігають виникненню відблисків. Забезпечують рівномірне й достатнє освітлення робочого місця (зазвичай комбіноване: природне + штучне без мерехтіння) та кондиціювання повітря

для оптимальної температури/вологості. Вентилятори й шумозахисні екрани можуть зменшити шумовий вплив. Ергономічні крісла й столи з регулюванням висоти дозволяють підтримувати правильну позу, а регульовані підставки – встановити монітор під оптимальним кутом і на відстані приблизно 50 см.

– Організаційні заходи: Дотримання режиму роботи й відпочинку. Оптимальна тривалість безперервної роботи за ПК не перевищує 30–45 хв; після цього обов'язкова перерва на 5–10 хв для фіззарядки та відпочинку очей. Рекомендують міняти види діяльності (час від часу відволікатися на нетехнічну роботу чи загальнорозвиваючі вправи), планувати регулярні перерви на прогулянку чи розминку. Такі заходи знижують статичне навантаження і психоемоційний стрес.

– Індивідуальні засоби: Носіння окулярів за рекомендацією окуліста (зокрема з антибліковим або синьо-блокувальним покриттям) підтримує зоровий комфорт при тривалій роботі. Виконання гімнастичних вправ для очей (наприклад, періодичний фокус на об'єктах на різній відстані чи закриття очей на декілька хвилин) допомагає зняти напругу. Регулярні вправи для шиї, плечей, рук і спини у перервах сприяють зниженню статичних м'язових навантажень. До індивідуальних засобів також належать періодичні медогляди, настанови з гігієни праці та зручний одяг, що не обмежує рухів.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи розроблено комп'ютерну гру «Impartial Automaton» з використанням ігрвого рушія Unreal Engine та мови програмування C++.

У першому розділі проведено аналітичний огляд існуючих рішень на ринку, виявлено їх слабкі та сильні сторони, створено технічне завдання.

У другому розділі розроблено загальну структуру гри та варінти її використання, розроблено систему класів з полями та методами, розроблено методи, спроектовано та опиано інтерфейс користувача, описано файлову систему комп'ютерної гри, описано та проведено тестування, відображено результати тестування.

За результатами виконання другого розділу розроблено UML – діаграму варіантів використання програми, UML – діаграму послідовності дій, UML – діаграму класів програми, UML – діаграму файлової структури програми. І представлено на окремих плакатах графічної частини дипломного проекту.

У третьому розділі надано інструкцію з комп'ютерної гри, порядок тестування та експлуатації програмного комплексу.

У економічній частині пояснювальної записку дипломного проєкту розраховано собівартість розробки програмного продукту.

Останній розділ описує питання охорони та техніки безпеки, яке було отримано від консультанта – викладача з охорони праці.

Дипломний проєкт містить увесь обсяг інформації та документації, необхідний для проектування та впровадження комп'ютерної гри «Impartial Automaton».

ПЕРЕЛІК ПОСИЛАНЬ

1. Методичні вказівки до виконання дипломного проекту з спеціальності 5.05010101 «Обслуговування програмних систем і комплексів» напрямок «Програмування» [Електронний ресурс] – Режим доступу до ресурсу: https://eguru1.tk.te.ua/pluginfile.php/76350/mod_resource/content/1/Metod_vkazi v_DP.pdf – Дата доступу: 13.06.2025р.

2. «Модель-вид-контролер» сторінка у «Wikipedia» [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Модель-вид-контролер – дата звертання: 14.06.2025р.

3. «Управління інспекційної діяльності у Тернопільській області Південно-Західного міжрегіонального управління Державної служби з питань праці» «Робота в офісі: основні санітарно-гігієнічні вимоги» [Електронний ресурс] – Режим доступу: https://te.dsp.gov.ua/robota-v-ofisi-osnovni-sanitarno-gigiyenichni-vymogy/ – дата звертання: 15.06.2025р.

4. «Artled» «Як розрахувати кількість світильників або потужність освітлення для різних типів приміщень» [Електронний ресурс] – Режим доступу: https://artled.com.ua/rozrakhunok-svitla/?srsltid=AfmBOoqeh_Mius92fnPXR_cdVYpKlleWbUx8zFRGnxoqNLG7CY7jnc3j – дата звертання: 15.06.2025р.

5. «Лугинська громада» «комп’ютер і здоров’я людини» [Електронний ресурс] – Режим доступу: https://lugynska-gromada.gov.ua/news/1604391805/ – дата звертання: 15.06.2025р.

6. «На Урок» «Монітори порівняльний аналіз та основні характеристики» [Електронний ресурс] – Режим доступу: https://naurok.com.ua/monitori-porivnyalniy-analiz-ta-osnovni-harakteristiki-230018.html – дата звертання: 15.06.2025р.

7. «Вперед» «Робота за комп`ютером є одним зі шкідливих виробничих факторів» [Електронний ресурс] – Режим доступу: https://vpered.od.ua/rajon/robota-za-kompyuteromy-e-odnim-zi-shkidlivix-virobnichix-faktoriv/ – дата звертання: 15.06.2025р.

8. «The Epoch Time» «Як роботу за комп’ютером зробити безпечнішою для здоров’я» [Електронний ресурс] – Режим доступу: https://www.epochtimes.com.ua/zdorovyi-sposib-zhyttya/yak-robotu-za-kompyuterom-zrobiti-bezpechnishoyu-dlya-zdorovya-116487 – дата звертання: 15.06.2025р.

9. «Mukachevo.net» «Чому постійна робота за комп'ютером завдає шкоди здоров'ю і як із цим боротися?» [Електронний ресурс] – Режим доступу: https://mukachevo.net/news/chomu-postiyna-robota-za-kompiuterom-zavdaye-shkody-zdoroviu-i-iak-iz-tsym-borotysia_6270788.html – дата звертання: 15.06.2025р.

ДОДАТКИ

Додаток А. Текст програми

Вихідний код файлів MyBaseCharacter.h та MyBaseCharacter.cpp.

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "NativeGameplayTags.h"
#include "ICombatComponent.h"
#include "MyBaseCharacter.generated.h"

class IIStatComponent;

UCLASS()
class WORK_API AMyBaseCharacter : public ACharacter
{
	GENERATED_BODY()
public:
	AMyBaseCharacter();
	virtual void GrantTag(FGameplayTag& TagToGrant);
	virtual bool CheckTag(const FGameplayTag& TagToCheck);
	virtual void RemoveTag(FGameplayTag& TagToRemove);
	virtual IIStatComponent* GetStatComponent() const;
	virtual void AddDamagedActors(AActor* Actor);
	virtual bool HasDamagedActor(AActor* Actor) const;
	virtual void ResetDamagedActors();
	UFUNCTION(BlueprintCallable)
	virtual void SwitchCollider(FName Socket, bool Enable = true);
```

```
        IICombatComponent* GetCombatComponent();

        UFUNCTION()
        virtual void OnCapsuleOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
            UPrimitiveComponent* OtherComp, int32 OtherBodyIndex,
            bool bFromSweep, const FHitResult& SweepResult);
        virtual void SetIsDead(bool IsDead);
        UFUNCTION(BlueprintCallable)
        virtual bool GetIsDead();
        UFUNCTION(BlueprintCallable)
        virtual void SetActorSpawnLocation(const FVector& Location);
        UFUNCTION(BlueprintCallable)
        virtual const FVector& GetActorSpawnLocation() const;
        UFUNCTION(BlueprintCallable)
        virtual void Respawn();
        virtual void SetIsInvincible(bool IsInvincible = true);
        UFUNCTION(BlueprintCallable)
        virtual void SetDamage(float Damage);
    protected:
        virtual void BeginPlay() override;
        virtual void HandleHit(AMyBaseCharacter* Attacker);
    public:
        virtual void Tick(float DeltaTime) override;
        virtual void SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent) override;
    protected:
        UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
```

```
		class UMyStatComponent* m_StatComponent;


		UPROPERTY(EditAnywhere, Category = "Animation | Combat")
		TMap<FName, UAnimMontage*> m_MontagesMap;
		UPROPERTY(EditAnywhere, Category = "Animation | Death")
		UAnimMontage* m_DeathMontage;
		UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "Tags")
		FGameplayTagContainer m_CharacterTags;
		UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
		TSet<AActor*> m_DamagedActors;
		UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
		bool m_bIsDead = false;
		UPROPERTY(BlueprintReadWrite, meta = (AllowPrivateAccess = true))
		bool m_bIsInvincible = false;
		UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
		class UCombatComponent* m_CombatComponent;
		UPROPERTY()
		FVector m_SpawnLocation;
		UPROPERTY(VisibleAnywhere)
		class AMyGameModeBase* m_GM;
};
	#include "MyBaseCharacter.h"
	#include "MyStatComponent.h"
	#include "Kismet/KismetMathLibrary.h"
	#include "CombatComponent.h"
	#include "Components/CapsuleComponent.h"
	#include "MyGameModeBase.h"
```

```
AMyBaseCharacter::AMyBaseCharacter()
{
	PrimaryActorTick.bCanEverTick = true;
	m_StatComponent = CreateDefaultSubobject<UMyStatComponent>("Stat Component");
	GetCapsuleComponent()->OnComponentBeginOverlap.AddDynamic(this,
		&AMyBaseCharacter::OnCapsuleOverlap);
	GetCapsuleComponent()->SetCollisionResponseToChannel(ECollisionChannel::ECC_Camera,
		ECollisionResponse::ECR_Ignore);
	m_bIsDead = false;
}
void AMyBaseCharacter::GrantTag(FGameplayTag& TagToGrant)
{
	m_CharacterTags.AddTag(TagToGrant);
}
bool AMyBaseCharacter::CheckTag(const FGameplayTag& TagToCheck)
{
	return m_CharacterTags.HasTag(TagToCheck);
}
void AMyBaseCharacter::RemoveTag(FGameplayTag& TagToRemove)
{
	m_CharacterTags.RemoveTag(TagToRemove);
}
IIStatComponent* AMyBaseCharacter::GetStatComponent() const
{
	return m_StatComponent;
```

```
}

void AMyBaseCharacter::AddDamagedActors(AActor* Actor)
{
	m_DamagedActors.Add(Actor);
}
bool AMyBaseCharacter::HasDamagedActor(AActor* Actor) const
{
	return m_DamagedActors.Contains(Actor);
}
void AMyBaseCharacter::ResetDamagedActors()
{
	m_DamagedActors.Empty();
}
void AMyBaseCharacter::HandleHit(AMyBaseCharacter*
Attacker){}
void AMyBaseCharacter::SetIsDead(bool IsDead)
{
	m_bIsDead = IsDead;
}
bool AMyBaseCharacter::GetIsDead()
{
	return m_bIsDead;
}
void AMyBaseCharacter::SetActorSpawnLocation(const FVector&
Location)
{
	m_SpawnLocation = Location;
}
const FVector& AMyBaseCharacter::GetActorSpawnLocation() const
```

```
{
	return m_SpawnLocation;}
void AMyBaseCharacter::Respawn()
{
	m_bIsDead = false;

	m_StatComponent->AddStatCurrentValue("Health",
		m_StatComponent->GetStatMaxValue("Health"));

	SetActorLocation(m_SpawnLocation);
}
void AMyBaseCharacter::SetIsInvincible(bool IsInvincible)
{
	m_bIsInvincible = IsInvincible;
}
void AMyBaseCharacter::SetDamage(float Damage)
{
}
void AMyBaseCharacter::OnCapsuleOverlap(UPrimitiveComponent*
OverlappedComponent, AActor* OtherActor, UPrimitiveComponent*
OtherComp,
	int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult)
{
	if (m_bIsDead || m_bIsInvincible) return;
	if (OtherActor && OtherActor != this && OtherComp)
	{
		if (OtherComp->ComponentHasTag(FName("WeaponCollider")))
		{
```

```
AMyBaseCharacter* Damager;

AActor* Attacker = OtherComp->GetOwner();

if (!IsValid(Damager = Cast<AMyBaseCharacter>(Attacker)))
{
    Attacker = Attacker->GetOwner();
    Damager = Cast<AMyBaseCharacter>(Attacker);
}
if (!Damager || Damager == this) return;

UMyStatComponent* AttackerStat = Cast<UMyStatComponent>(Damager->GetStatComponent());

if (AttackerStat && m_StatComponent)
{
    if (Damager->HasDamagedActor(this)) return;
    float DamageAmount = AttackerStat->GetStatValue("Damage");
    m_StatComponent->AddStatCurrentValue("Health", DamageAmount);
}
if (this->m_StatComponent->GetStatValue("Health") <= 0.f)
{
    m_bIsDead = true;
}
```

```
                    HandleHit(Damager);
                }}}
    void AMyBaseCharacter::BeginPlay()
    {
        Super::BeginPlay();
        m_SpawnLocation = this->GetActorLocation();
        m_GM = GetWorld()->GetAuthGameMode<AMyGameModeBase>();
    }
    void AMyBaseCharacter::Tick(float DeltaTime)
    {
        Super::Tick(DeltaTime);

    }
    void AMyBaseCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
    {
        Super::SetupPlayerInputComponent(PlayerInputComponent);

    }
    void AMyBaseCharacter::SwitchCollider(FName Socket, bool Enable)
    {
        (Enable) ? m_CombatComponent->EnableHitbox(Socket) : m_CombatComponent->DeactivateHitbox(Socket);
    }
    IICombatComponent* AMyBaseCharacter::GetCombatComponent()
    {
        return m_CombatComponent;}
```

Вихідний код файлів MyBaseCharacter.h та MyBaseCharacter.cpp.

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "NativeGameplayTags.h"
#include "ICombatComponent.h"
#include "MyBaseCharacter.generated.h"

class IIStatComponent;

UCLASS()
class WORK_API AMyBaseCharacter : public ACharacter
{
	GENERATED_BODY()

public:
	AMyBaseCharacter();

	virtual void GrantTag(FGameplayTag& TagToGrant);

	virtual bool CheckTag(const FGameplayTag& TagToCheck);

	virtual void RemoveTag(FGameplayTag& TagToRemove);

	virtual IIStatComponent* GetStatComponent() const;
```

```
virtual void AddDamagedActors(AActor* Actor);

virtual bool HasDamagedActor(AActor* Actor) const;

virtual void ResetDamagedActors();

UFUNCTION(BlueprintCallable)
virtual void SwitchCollider(FName Socket, bool Enable = true);

IICombatComponent* GetCombatComponent();

UFUNCTION()
virtual void OnCapsuleOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
    UPrimitiveComponent* OtherComp, int32 OtherBodyIndex,
    bool bFromSweep, const FHitResult& SweepResult);

virtual void SetIsDead(bool IsDead);

UFUNCTION(BlueprintCallable)
virtual bool GetIsDead();

UFUNCTION(BlueprintCallable)
virtual void SetActorSpawnLocation(const FVector& Location);

UFUNCTION(BlueprintCallable)
virtual const FVector& GetActorSpawnLocation() const;

UFUNCTION(BlueprintCallable)
```

```
        virtual void Respawn();

        virtual void SetIsInvincible(bool IsInvincible = true);

        UFUNCTION(BlueprintCallable)
        virtual void SetDamage(float Damage);

    protected:
        virtual void BeginPlay() override;

        virtual void HandleHit(AMyBaseCharacter* Attacker);


    public:
        virtual void Tick(float DeltaTime) override;

        virtual void SetupPlayerInputComponent(class
UInputComponent* PlayerInputComponent) override;

    protected:
        UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
        class UMyStatComponent* m_StatComponent;

        UPROPERTY(EditAnywhere, Category = "Animation |
Combat")
        TMap<FName, UAnimMontage*> m_MontagesMap;

        UPROPERTY(EditAnywhere, Category = "Animation | Death")
        UAnimMontage* m_DeathMontage;
```

```
        UPROPERTY(VisibleAnywhere, BlueprintReadWrite,
Category = "Tags")
        FGameplayTagContainer m_CharacterTags;

        UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
        TSet<AActor*> m_DamagedActors;

        UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
        bool m_bIsDead = false;

        UPROPERTY(BlueprintReadWrite, meta =
(AllowPrivateAccess = true))
        bool m_bIsInvincible = false;

        UPROPERTY(VisibleAnywhere, BlueprintReadOnly)
        class UCombatComponent* m_CombatComponent;

        UPROPERTY()
        FVector m_SpawnLocation;

        UPROPERTY(VisibleAnywhere)
        class AMyGameModeBase* m_GM;
    };
    #include "MyBaseCharacter.h"
    #include "MyStatComponent.h"
    #include "Kismet/KismetMathLibrary.h"
    #include "CombatComponent.h"
    #include "Components/CapsuleComponent.h"
    #include "MyGameModeBase.h"
```

```
AMyBaseCharacter::AMyBaseCharacter()
{
	PrimaryActorTick.bCanEverTick = true;
	m_StatComponent =
CreateDefaultSubobject<UMyStatComponent>("Stat Component");
	GetCapsuleComponent()-
>OnComponentBeginOverlap.AddDynamic(this,
		&AMyBaseCharacter::OnCapsuleOverlap);
	GetCapsuleComponent()-
>SetCollisionResponseToChannel(ECollisionChannel::ECC_Camera,
		ECollisionResponse::ECR_Ignore);
	m_bIsDead = false;
}
void AMyBaseCharacter::GrantTag(FGameplayTag& TagToGrant)
{
	m_CharacterTags.AddTag(TagToGrant);
}
bool AMyBaseCharacter::CheckTag(const FGameplayTag&
TagToCheck)
{
	return m_CharacterTags.HasTag(TagToCheck);
}
void AMyBaseCharacter::RemoveTag(FGameplayTag&
TagToRemove)
{
	m_CharacterTags.RemoveTag(TagToRemove);
}
IIStatComponent* AMyBaseCharacter::GetStatComponent() const
{
	return m_StatComponent;
```

```
}
void AMyBaseCharacter::AddDamagedActors(AActor* Actor)
{
	m_DamagedActors.Add(Actor);
}
bool AMyBaseCharacter::HasDamagedActor(AActor* Actor) const
{
	return m_DamagedActors.Contains(Actor);
}
void AMyBaseCharacter::ResetDamagedActors()
{
	m_DamagedActors.Empty();
}
void AMyBaseCharacter::HandleHit(AMyBaseCharacter* Attacker){}

void AMyBaseCharacter::SetIsDead(bool IsDead)
{
	m_bIsDead = IsDead;
}

bool AMyBaseCharacter::GetIsDead()
{
	return m_bIsDead;
}

void AMyBaseCharacter::SetActorSpawnLocation(const FVector& Location)
{
	m_SpawnLocation = Location;
```

```
}

const FVector& AMyBaseCharacter::GetActorSpawnLocation() const
{
	return m_SpawnLocation;
}
void AMyBaseCharacter::Respawn()
{
	m_bIsDead = false;
	m_StatComponent->AddStatCurrentValue("Health",
		m_StatComponent->GetStatMaxValue("Health"));
	SetActorLocation(m_SpawnLocation);
}
void AMyBaseCharacter::SetIsInvincible(bool IsInvincible)
{
	m_bIsInvincible = IsInvincible;
}
void AMyBaseCharacter::SetDamage(float Damage){}

void AMyBaseCharacter::OnCapsuleOverlap(UPrimitiveComponent*
OverlappedComponent, AActor* OtherActor, UPrimitiveComponent*
OtherComp,
	int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult)
{
	if (m_bIsDead || m_bIsInvincible) return;
	if (OtherActor && OtherActor != this && OtherComp)
	{
		if (OtherComp-
>ComponentHasTag(FName("WeaponCollider")))
```

```
{
    AMyBaseCharacter* Damager;
    AActor* Attacker = OtherComp->GetOwner();
    if (!IsValid(Damager = Cast<AMyBaseCharacter>(Attacker)))
    {
        Attacker = Attacker->GetOwner();
        Damager = Cast<AMyBaseCharacter>(Attacker);
    }
    if (!Damager || Damager == this) return;
    UMyStatComponent* AttackerStat = Cast<UMyStatComponent>(Damager->GetStatComponent());
    if (AttackerStat && m_StatComponent)
    {
        if (Damager->HasDamagedActor(this)) return;

        float DamageAmount = AttackerStat->GetStatValue("Damage");

        m_StatComponent->AddStatCurrentValue("Health", DamageAmount);
    }
    if (this->m_StatComponent->GetStatValue("Health") <= 0.f)
    {
        m_bIsDead = true;
    }
    HandleHit(Damager);}}}
```

```
void AMyBaseCharacter::BeginPlay()
{
	Super::BeginPlay();
	m_SpawnLocation = this->GetActorLocation();
	m_GM = GetWorld()->GetAuthGameMode<AMyGameModeBase>();
}
void AMyBaseCharacter::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);
}
void AMyBaseCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
	Super::SetupPlayerInputComponent(PlayerInputComponent);
}
void AMyBaseCharacter::SwitchCollider(FName Socket, bool Enable)
{
	(Enable) ? m_CombatComponent->EnableHitbox(Socket) : m_CombatComponent->DeactivateHitbox(Socket);
}
IICombatComponent* AMyBaseCharacter::GetCombatComponent()
{
	return m_CombatComponent;}
```

Вихідний код файлів Enemy.h та Enemy.cpp.

```
#pragma once

#include "CoreMinimal.h"
```

```
#include "MyBaseCharacter.h"
#include "GameFramework/CharacterMovementComponent.h"

#include "Enemy.generated.h"


UCLASS()
class WORK_API AEnemy : public AMyBaseCharacter
{
	GENERATED_BODY()

public:
	AEnemy();

	virtual void Tick(float DeltaTime) override;

	UFUNCTION(BlueprintCallable)
	void RotateToPlayer();

	virtual void Respawn() override;

protected:

	virtual void BeginPlay() override;

	UFUNCTION()
	void OnSeePawn(APawn* Pawn);

	UFUNCTION(BlueprintCallable)
	void StopChasing();
```

```
UFUNCTION(BlueprintCallable)
APawn* GetTargetPawn() const;

float GetDistance();

UFUNCTION(BlueprintCallable)
AActor* ChoosePatrolTarget();

UFUNCTION(BlueprintCallable)
void Attack();

UFUNCTION(BlueprintCallable)
void StopAttack();

virtual void HandleHit(AMyBaseCharacter* Attacker) override;

UFUNCTION(BlueprintCallable)
const FName& GetRandomMontageSection() const;

UFUNCTION(BlueprintCallable)
void PlayMontage(UAnimMontage* Montage, const FName& Section);

UFUNCTION(BlueprintCallable)
UAnimMontage* GetMontage() const;

void TurnOffAI();

UFUNCTION(BlueprintCallable)
```

```
        UBlackboardComponent* GetBlackBoard();

    protected:

        UPROPERTY()
        class AAIController* m_Controller;

        UPROPERTY(VisibleAnywhere, BlueprintReadWrite,
meta=(AllowPrivateAccess = true))
        class APawn* m_TargetActor;

        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        class UPawnSensingComponent* m_Senses;

        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        class UBehaviorTree* m_Tree;

        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        TArray<AActor*> m_PatrolPoints;

        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        AActor* m_ActivePatrolTarget;

        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        UAnimMontage* m_HitMontage;
```

```
        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        UAnimMontage* m_AttackMontage;

        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        TArray<FName> m_MontageSections;

        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        FName m_DefaultMontageSectionName;

        UPROPERTY(EditAnywhere, meta=(AllowPrivateAccess =
true))
        float m_ChaseableDistance;

        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        float m_AttackDistance;

        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        class UBlackboardComponent* m_Blackboard;

        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        float m_RotationSpeed;
```

```
        UPROPERTY(EditAnywhere, meta = (AllowPrivateAccess =
true))
        TSubclassOf<AMyWeapon> m_Weapon;
    };
    #include "Enemy.h"
    #include "BehaviorTree/BehaviorTree.h"
    #include "BehaviorTree/BlackboardComponent.h"
    #include "Perception/PawnSensingComponent.h"
    #include "CombatComponent.h"
    #include "Kismet/KismetMathLibrary.h"
    #include "Components/CapsuleComponent.h"
    #include "MyGameModeBase.h"
    #include "AIController.h"


    AEnemy::AEnemy()
    {
        PrimaryActorTick.bCanEverTick = true;
        GetMesh()-
>SetCollisionObjectType(ECollisionChannel::ECC_WorldDynamic);
        GetMesh()-
>SetCollisionResponseToChannel(ECollisionChannel::ECC_Visibility,
ECollisionResponse::ECR_Block);
        GetMesh()-
>SetCollisionResponseToChannel(ECollisionChannel::ECC_Camera,
ECollisionResponse::ECR_Ignore);
        GetMesh()->SetGenerateOverlapEvents(true);
        m_Senses                                              =
CreateDefaultSubobject<UPawnSensingComponent>(TEXT("Pawn
Sensing"));
```

```
        m_Senses->SetPeripheralVisionAngle(70.f);
        m_Senses->SightRadius = 500.f;
        m_Senses->SensingInterval = 0.5f;
        m_CombatComponent =
CreateDefaultSubobject<UCombatComponent>("Combat");
    }

    void AEnemy::Tick(float DeltaTime)
    {
        Super::Tick(DeltaTime);
        if (m_bIsDead) return;
        if (m_TargetActor && GetDistance() > m_ChaseableDistance)
        {
            StopChasing();
        }
        if (m_TargetActor && GetDistance() <= m_AttackDistance)
        {
            Attack();
        }
        else
        {
            StopAttack();
        }
    }

    void AEnemy::RotateToPlayer()
    {
        if (!m_TargetActor) return;
        FVector TargetLoc = m_TargetActor->GetActorLocation();
        FRotator CurrentRotation = GetActorRotation();
```

```
            FRotator TargetRotation =
UKismetMathLibrary::FindLookAtRotation(GetActorLocation(),
TargetLoc);
            FRotator NewRotation = FMath::RInterpTo(CurrentRotation,
TargetRotation, GetWorld()->GetDeltaSeconds(), m_RotationSpeed);
            SetActorRotation(NewRotation);
        }

        void AEnemy::Respawn()
        {
            Super::Respawn();
            GetCapsuleComponent()-
>SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
            m_Blackboard->SetValueAsBool(TEXT("IsDead"), false);
            m_Controller->BrainComponent->StartLogic();
            auto MoveComp = GetCharacterMovement();
            MoveComp->Activate();
            MoveComp->SetMovementMode(MOVE_Walking);
        }

        void AEnemy::BeginPlay()
        {
            Super::BeginPlay();
            m_GM->AddEnemyToArray(this);
            m_Controller = Cast<AAIController>(GetController());
            m_Controller->RunBehaviorTree(m_Tree);
            if (m_Senses)
            {
                m_Senses->OnSeePawn.AddDynamic(this,
&AEnemy::OnSeePawn);
```

```
        }
        m_Blackboard = m_Controller->GetBlackboardComponent();
        m_ActivePatrolTarget = ChoosePatrolTarget();
        if(m_CombatComponent)
            m_CombatComponent->EquipWeapon();
        else
            UE_LOG(LogTemp, Warning, TEXT("None Combat"));
        if (m_bIsDead)
            TurnOffAI();
    }

    void AEnemy::OnSeePawn(APawn* Pawn)
    {
        if (Pawn)
        {
            m_TargetActor = Pawn;
            m_Controller->GetBlackboardComponent()-
>SetValueAsBool("HasSeenActor", true);
            this->GetCharacterMovement()->MaxWalkSpeed =
400.f;
        }
    }

    void AEnemy::StopChasing()
    {
        if (AAIController* AIController =
Cast<AAIController>(GetController()))
        {
            AIController->StopMovement();
        }
```

```
		this->GetCharacterMovement()->MaxWalkSpeed = 200.f;
		m_Blackboard->SetValueAsBool("HasSeenActor", false);
		m_TargetActor = nullptr;
	}

	UFUNCTION(BlueprintCallable)
	APawn* AEnemy::GetTargetPawn() const
	{
		return m_TargetActor;
	}

	float AEnemy::GetDistance()
	{
		return FVector::Dist(this->GetActorLocation(), m_TargetActor-
>GetActorLocation());
	}

	AActor* AEnemy::ChoosePatrolTarget()
	{
		TArray<AActor*> ValidTargets;

		for (AActor* Target : m_PatrolPoints)
		{
			if (Target != m_ActivePatrolTarget)
			{
				ValidTargets.AddUnique(Target);
			}
		}
		const int32 NumPatrolTargets = ValidTargets.Num();
		if (NumPatrolTargets > 0)
```

```
            {
                const int32 TargetSelection = FMath::RandRange(0,
NumPatrolTargets - 1);
                m_ActivePatrolTarget = ValidTargets[TargetSelection];
                return m_ActivePatrolTarget;
            }
            return nullptr;
        }

        void AEnemy::Attack()
        {
            m_Blackboard->SetValueAsBool("CanAttack", true);
        }

        void AEnemy::StopAttack()
        {
            m_Blackboard->SetValueAsBool("CanAttack", false);
        }

        void AEnemy::HandleHit(AMyBaseCharacter* Attacker)
        {
            if (m_bIsDead)
            {
                this->GetMesh()->GetAnimInstance()-
>Montage_Play(m_DeathMontage);
                TurnOffAI();
                return;
            }
            if (!m_TargetActor)
            {
```

```
            m_TargetActor = Attacker;
            m_Blackboard->SetValueAsBool("HasSeenActor", true);
        }
        m_Blackboard->SetValueAsBool("WasHit", true);
        this->GetMesh()->GetAnimInstance()-
>Montage_Play(m_HitMontage);

    }
    const FName& AEnemy::GetRandomMontageSection() const
    {
        if (m_AttackMontage && m_MontageSections.Num() != 0)
        {
            const int32 SectionSelection = FMath::RandRange(0,
m_MontageSections.Num() - 1);
            return m_MontageSections[SectionSelection];
        }
        return m_DefaultMontageSectionName;
    }
    void AEnemy::PlayMontage(UAnimMontage* Montage, const
FName& Section)
    {
        this->GetMesh()->GetAnimInstance()-
>Montage_Play(Montage);
        this->GetMesh()->GetAnimInstance()-
>Montage_JumpToSection(Section);
    }
    UAnimMontage* AEnemy::GetMontage() const
    {
        return m_AttackMontage;
    }
```

```
void AEnemy::TurnOffAI()
{
	GetCapsuleComponent()-
>SetCollisionEnabled(ECollisionEnabled::NoCollision);
	m_Blackboard->SetValueAsBool(TEXT("IsDead"), true);
	m_Controller->BrainComponent->StopLogic(TEXT("Dead"));
	m_Controller->StopMovement();
	auto MoveComp = GetCharacterMovement();
	MoveComp->DisableMovement();
	MoveComp->StopMovementImmediately();
}
UBlackboardComponent* AEnemy::GetBlackBoard()
{
	return m_Blackboard;
}
```

Вихідний код файлів Checkpoint.h та Checkpoint.cpp.

```
#pragma once

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

UCLASS()
class WORK_API ACheckpoint : public AActor
{
	GENERATED_BODY()

public:
	ACheckpoint();
```

```
        virtual const FVector GetLocation() const;
    protected:
        virtual void BeginPlay() override;

        UFUNCTION()
        void OnOverlapBegin(UPrimitiveComponent*
OverlappedComp,
            AActor* OtherActor,
            UPrimitiveComponent* OtherComp,
            int32 OtherBodyIndex,
            bool bFromSweep,
            const FHitResult& SweepResult);


    public:
        virtual void Tick(float DeltaTime) override;

    protected:

        UPROPERTY(BlueprintReadWrite, VisibleAnywhere)
        FVector m_CheckpointLocation;

        UPROPERTY(VisibleAnywhere)
        class USphereComponent* m_SphereCollider;

        UPROPERTY(EditAnywhere)
        class UStaticMeshComponent* m_Mesh;
};
```

```
#include "Checkpoint.h"
#include "Components/SphereComponent.h"
#include "MyPlayerCharacter.h"
#include "Components/StaticMeshComponent.h"
#include "IStatComponent.h"

ACheckpoint::ACheckpoint()
{
	PrimaryActorTick.bCanEverTick = true;
	m_SphereCollider =
CreateDefaultSubobject<USphereComponent>("Collider");
	m_SphereCollider->InitSphereRadius(200.f);
	m_SphereCollider-
>SetCollisionProfileName(TEXT("Trigger"));
	RootComponent = m_SphereCollider;
	m_SphereCollider-
>OnComponentBeginOverlap.AddDynamic(this,
&ACheckpoint::OnOverlapBegin);
	m_Mesh =
CreateDefaultSubobject<UStaticMeshComponent>("Mesh");
	m_Mesh->SetupAttachment(GetRootComponent());
}
void ACheckpoint::BeginPlay()
{
	Super::BeginPlay();
	m_CheckpointLocation = this->GetActorLocation();
}

void ACheckpoint::OnOverlapBegin(UPrimitiveComponent*
OverlappedComp, AActor* OtherActor,
```

```
        UPrimitiveComponent* OtherComp, int32 OtherBodyIndex,
bool bFromSweep,
        const FHitResult& SweepResult)
    {
        if (!OtherActor) return;
        if (AMyPlayerCharacter* Player =
Cast<AMyPlayerCharacter>(OtherActor))
        {
            Player-
>SetActorSpawnLocation(m_CheckpointLocation);
            Player->GetStatComponent()-
>AddStatCurrentValue("Health",
                Player->GetStatComponent()-
>GetStatMaxValue("Health"));}}

    const FVector ACheckpoint::GetLocation() const
    {return m_CheckpointLocation;}
    void ACheckpoint::Tick(float DeltaTime)
    {Super::Tick(DeltaTime);}
```

Вихідний код файлів CombatComponent.h та CombatComponent.cpp.

```
    #pragma once

    #include "CoreMinimal.h"
    #include "Components/ActorComponent.h"
    #include "ICombatComponent.h"
    #include "CombatComponent.generated.h"
```

```
UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class WORK_API UCombatComponent : public UActorComponent, public IICombatComponent
{
    GENERATED_BODY()
public:
    UCombatComponent();
    void EquipWeapon() override;
    UAnimMontage* GetActiveMontage() const override;
    UAnimMontage* GetMontage(const FName& MontageName) const override;
    UAnimMontage* GetLRMontage(bool Right = true) const override;
    bool HasWeaponIn(const FName& SocketName) override;
    void ReattachFromTo(const FName& SocketName, const FName& TargetSocket) override;
    void ChangeMontage(UAnimMontage* ChangeTo, bool Right = true) override;
    void SwapWeapon(const FName& SocketName, const FName& TargetSocket, UAnimMontage* ChangeTo) override;
    void EnableHitbox(const FName& Socket = "Hand_R_Socket");
    void DeactivateHitbox(const FName& Socket = "Hand_R_Socket");
    UFUNCTION(BlueprintCallable)
    void SetWeaponClass(TSubclassOf<AMyBaseWeapon> WeaponClass);
protected:
    virtual void BeginPlay() override;
public:
```

```
        virtual void TickComponent(float DeltaTime, ELevelTick
TickType, FActorComponentTickFunction* ThisTickFunction) override;
    protected:
        UPROPERTY(VisibleAnywhere)
        class AMyBaseCharacter* m_ComponentOwner;
        UPROPERTY(EditAnywhere, Category = "Combat",
meta=(AllowPrivateAccess = true))
        TSubclassOf<class AMyWeapon> m_WeaponClass;
        UPROPERTY(EditAnywhere, Category = "Combat", meta =
(AllowPrivateAccess = true))
        TMap<FName, AMyWeapon*> m_SocketsMap;
        UPROPERTY(VisibleAnywhere)
        UAnimMontage* m_EquipedRWeaponMontage;
        UPROPERTY(VisibleAnywhere)
        UAnimMontage* m_EquipedLWeaponMontage;};

    #include "CombatComponent.h"
    #include "MyWeapon.h"
    #include "MyPlayerCharacter.h"
    #include "GameFramework/Character.h"

    UCombatComponent::UCombatComponent()
    {
        PrimaryComponentTick.bCanEverTick = false;
    }

    void UCombatComponent::EquipWeapon()
    {
        if (!m_WeaponClass) return;
```

```
if (!m_ComponentOwner) return;

FName TargetSocket = m_WeaponClass->GetDefaultObject<AMyWeapon>()->GetAttachToSocketName();

if (!m_SocketsMap.Contains(TargetSocket)) return;

AMyWeapon** TargetWeapon = m_SocketsMap.Find(TargetSocket);

if (TargetWeapon && *TargetWeapon)
{
    (*TargetWeapon)->Destroy();
    *TargetWeapon = nullptr;
}

FActorSpawnParameters SpawnParams;
SpawnParams.Owner = m_ComponentOwner;
SpawnParams.Instigator = m_ComponentOwner->GetInstigator();

*TargetWeapon = GetWorld()->SpawnActor<AMyWeapon>(m_WeaponClass, SpawnParams);

if (TargetWeapon)
{
    USkeletalMeshComponent* Mesh = m_ComponentOwner->GetMesh();

    FGameplayTag Tag = Tag_Part_SkeletalRUp;
```

```
                FName ToSocket;

                if (!m_ComponentOwner->CheckTag(Tag))
                {
                    ToSocket = (*TargetWeapon)->GetAttachToSocketName();

                    if (UAnimMontage* Montage = (*TargetWeapon)->GetWeaponAttackMontage())
                        m_EquipedRWeaponMontage = Montage;
                }
                else
                {
                    ToSocket = (*TargetWeapon)->GetResortSocket();
                }
                (*TargetWeapon)->AttachToComponent(
                    Mesh,
        FAttachmentTransformRules::SnapToTargetIncludingScale,
                    ToSocket
                );
            }
            for (auto tag : (*(TargetWeapon))->GetTags())
            {
                m_ComponentOwner->GrantTag(tag);
            }
            TargetWeapon = nullptr;
        }
        UAnimMontage* UCombatComponent::GetActiveMontage() const
```

```
{
    return m_EquipedRWeaponMontage;
}
UAnimMontage* UCombatComponent::GetMontage(const FName& MontageName) const
{
    AMyWeapon* const* FoundWeapon = m_SocketsMap.Find(MontageName);

    if (FoundWeapon && *FoundWeapon)
    {
        return (*FoundWeapon)->GetWeaponAttackMontage();
    }
    return nullptr;
}
UAnimMontage* UCombatComponent::GetLRMontage(bool Right) const
{
    return (Right) ? m_EquipedRWeaponMontage : m_EquipedLWeaponMontage;
}
bool UCombatComponent::HasWeaponIn(const FName& SocketName)
{
    if (m_SocketsMap.Contains(SocketName))
    {
        return ((*(m_SocketsMap.Find(SocketName)))) ? true : false;
    }
    return false;
```

```
}
void UCombatComponent::ReattachFromTo(const FName& SocketName, const FName& TargetSocket)
{
	AMyWeapon** TargetWeapon = m_SocketsMap.Find(SocketName);
	if (!(TargetWeapon && *TargetWeapon)) return;
	USkeletalMeshComponent* Mesh = m_ComponentOwner->GetMesh();
	(*TargetWeapon)->AttachToComponent(
		Mesh,
	FAttachmentTransformRules::SnapToTargetIncludingScale,
		TargetSocket
	);
}
void UCombatComponent::ChangeMontage(UAnimMontage* ChangeTo, bool Right)
{
	(Right) ? m_EquipedRWeaponMontage = ChangeTo : m_EquipedLWeaponMontage = ChangeTo;
}
void UCombatComponent::SwapWeapon(const FName& SocketName, const FName& TargetSocket, UAnimMontage* ChangeTo)
{
	ReattachFromTo(SocketName, TargetSocket);

	ChangeMontage(ChangeTo);
}
void UCombatComponent::EnableHitbox(const FName& Socket)
{
```

```
        (*(m_SocketsMap.Find(Socket)))->ActivateHitbox();
    }
    void UCombatComponent::DeactivateHitbox(const FName& Socket)
    {
        (*(m_SocketsMap.Find(Socket)))->DeactivateHitbox();
    }
    void
UCombatComponent::SetWeaponClass(TSubclassOf<AMyBaseWeapon>
WeaponClass)
    {
        m_WeaponClass = WeaponClass;
        EquipWeapon();
    }
    void UCombatComponent::BeginPlay()
    {
        Super::BeginPlay();
        m_ComponentOwner                                    =
Cast<AMyBaseCharacter>(GetOwner());
    }
    void UCombatComponent::TickComponent(float DeltaTime,
ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
    {Super::TickComponent(DeltaTime, TickType, ThisTickFunction);}
```

Вихідний код файлів LockOnComponent.h та LockOnComponent.cpp.

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

    #pragma once

    #include "CoreMinimal.h"
```

```
#include "Components/ActorComponent.h"
#include "ILockOnComponent.h"
#include "LockOnComponent.generated.h"

UDELEGATE(BlueprintCallable)
DECLARE_DYNAMIC_MULTICAST_SPARSE_DELEGATE_OneParam(
    FOnUpdateTargetSignature,
    ULockOnComponent, OnUpdateTargetDelegate,
    AActor*, TargetActorRef
);


UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class WORK_API ULockOnComponent : public UActorComponent, public IILockOnComponent
{
    GENERATED_BODY()

public:
    ULockOnComponent();

    void StartLockOn(float Radius) override;

    void EndLockOn() override;

    void ToggleLockOn(float Radius) override;

    AActor* GetTarget() override;
```

```
protected:
    virtual void BeginPlay() override;

public:
    virtual void TickComponent(float DeltaTime, ELevelTick
TickType, FActorComponentTickFunction* ThisTickFunction) override;

public:
    UPROPERTY(VisibleAnywhere)
    AActor* m_CurrentTargetActor;

    UPROPERTY(BlueprintAssignable, Category = "LockOn")
    FOnUpdateTargetSignature OnUpdateTargetDelegate;

protected:
    UPROPERTY(VisibleAnywhere)
    ACharacter* m_ComponentOwner;

    UPROPERTY(VisibleAnywhere)
    APlayerController* m_PlayerController;

    UPROPERTY(VisibleAnywhere)
    class UCharacterMovementComponent*
m_MovementComponent;

    UPROPERTY(VisibleAnywhere)
    class USpringArmComponent* m_SpringArm;

    UPROPERTY(EditAnywhere)
    double m_BreakLockOnDistance = 1000.0;
```

```
};
	#include "LockOnComponent.h"
	#include "GameFramework/Character.h"
	#include "GameFramework/CharacterMovementComponent.h"
	#include "Kismet/KismetMathLibrary.h"
	#include "MyBaseCharacter.h"
	#include "GameFramework/SpringArmComponent.h"

	ULockOnComponent::ULockOnComponent()
	{
		PrimaryComponentTick.bCanEverTick = true;
	}

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

		m_ComponentOwner = GetOwner<ACharacter>();
		m_PlayerController = m_ComponentOwner->GetController<APlayerController>();
		m_MovementComponent = m_ComponentOwner->GetCharacterMovement();

		m_SpringArm = m_ComponentOwner->FindComponentByClass<USpringArmComponent>();
	}

	void ULockOnComponent::StartLockOn(float Radius)
	{
		TArray<FHitResult> OutResults;
```

```
		FVector CurrentLocation = m_ComponentOwner->GetActorLocation();
		FCollisionShape Sphere = FCollisionShape::MakeSphere(Radius);
		FCollisionQueryParams IgnoreParams(FName(TEXT("Ignore Objects")), false, m_ComponentOwner);

		bool bHasFoundTargets = GetWorld()->SweepMultiByChannel(
			OutResults,
			CurrentLocation,
			CurrentLocation,
			FQuat::Identity,
			ECollisionChannel::ECC_GameTraceChannel1,
			Sphere,
			IgnoreParams
		);

		if (!bHasFoundTargets) return;

		AMyBaseCharacter* ClosestEnemy = nullptr;
		float MinDistance = FLT_MAX;

		for (const FHitResult& Hit : OutResults)
		{
			AActor* HitActor = Hit.GetActor();
			if (!HitActor) continue;

			AMyBaseCharacter* Enemy = Cast<AMyBaseCharacter>(HitActor);
			if (!Enemy) continue;
```

```
                if (Enemy->GetIsDead()) continue;

                float Distance = FVector::Dist(CurrentLocation, Enemy->GetActorLocation());
                if (Distance < MinDistance)
                {
                    MinDistance = Distance;
                    ClosestEnemy = Enemy;
                }
            }

            if (!ClosestEnemy) return;

            m_CurrentTargetActor = ClosestEnemy;
            m_PlayerController->SetIgnoreLookInput(true);
            m_MovementComponent->bOrientRotationToMovement = false;
            m_MovementComponent->bUseControllerDesiredRotation = true;
            m_SpringArm->TargetOffset = FVector(0.0f, 0.0f, 100.0f);
            OnUpdateTargetDelegate.Broadcast(m_CurrentTargetActor);
        }

        void ULockOnComponent::EndLockOn()
        {
            m_CurrentTargetActor = nullptr;
            m_MovementComponent->bOrientRotationToMovement = true;
```

```
		m_MovementComponent->bUseControllerDesiredRotation = false;

		m_SpringArm->TargetOffset = FVector::ZeroVector;

		m_PlayerController->ResetIgnoreLookInput();

		OnUpdateTargetDelegate.Broadcast(m_CurrentTargetActor);
	}

	void ULockOnComponent::ToggleLockOn(float Radius)
	{
		if (m_CurrentTargetActor)
		{
			EndLockOn();
		}
		else
		{
			StartLockOn(Radius);
		}
	}

	AActor* ULockOnComponent::GetTarget()
	{
		return m_CurrentTargetActor;
	}


	void ULockOnComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
```

```
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    if (!m_CurrentTargetActor) return;

    if (Cast<AMyBaseCharacter>(m_CurrentTargetActor)->GetIsDead())
    {
        EndLockOn();
        return;
    }

    FVector CurrentLocation = m_ComponentOwner->GetActorLocation();

    FVector TargetLocation = m_CurrentTargetActor->GetActorLocation();

    double DistanceToTarget = FVector::Distance(CurrentLocation, TargetLocation);

    if (DistanceToTarget >= m_BreakLockOnDistance)
    {
        EndLockOn();
        return;
    }

    TargetLocation.Z -= 125;
```

```
        FRotator NewRotation =
UKismetMathLibrary::FindLookAtRotation(
            CurrentLocation, TargetLocation
        );

        NewRotation.Pitch = FMath::Clamp(NewRotation.Pitch, -30,
75);

        m_PlayerController->SetControlRotation(NewRotation);
}
```