МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

КАЛЬНИЦЬКА АДЕЛІНА АСКОЛЬДІВНА

Допускається до захисту:
завідувач кафедри
комп'ютерних наук та
інформаційних технологій,
доцент
_____ Нескородєва Т. В.
«___» червня 2021 р.

РОЗРОБКА КОМП'ЮТЕРНОЇ ГРИ З РЕАЛІЗАЦІЄЮ МЕХАНІЗМУ АДАПТАЦІЇ

Спеціальність 122 Комп'ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Штовба С. Д., професор кафедри
інформаційних технологій
д.т.н., професор

Оцінка:____/____/ _____
(Бали/ за шкалою ECTS/ за національною шкалою)
Голова ЕК: _____
(Підпис)

Вінниця 2021

АНОТАЦІЯ

Кальницька А.А. Розробка комп'ютерної гри з реалізацією механізму адаптації. Спеціальність 122 «Комп'ютерні науки», спеціалізація «Сучасні інформаційні технології та програмування». Донецький національний університет імені Василя Стуса, Вінниця, 2021.

У кваліфікаційній (бакалаврській) роботі проведено аналіз існуючих методів до розробки комп'ютерної гри в жанрі шутер. Побудована логіка ігрового штучного інтелекту на основі дереві поведінки. Розроблена комп'ютерна гра в жанрі шутер від третьої особи з реалізованим механізмом адаптації.

Ключові слова: комп'ютерна гра, шутер, дерева поведінки.

Рис. 48. Бібліограф.: 28 найм.

ABSTRACT

Kalnytska A. Development of a computer game with the implementation of the adaptation mechanism. Specialty 122 "Computer Science", specialization "Modern information technologies and programming". Vasyl' Stus Donetsk National University, Vinnytsia, 2021.

This bachelor's thesis analyzes the existing methods of development a computer game in the shooter game genre. The logic of game artificial intelligence is build on the basic of the behavior tree. Was developed a third-person shooter computer game with an implemented mechanism of adaptation.

Keywords: computer game, shooter game, behavior tree.

Fig. 48. Bibliographer.: 28 items.

ЗМІСТ

ВСТУП

Актуальність теми. Ігри супроводжують людство з давніх часів. Тільки існували в різних формах. В наш час набули популярність відеоігри. Ігрова індустрія – одна з швидкозростаючих та прибуткових. Але розробка ігор це не тільки заробіток, це, насамперед, розвага, яка впливає на соціум. Комп'ютерні ігри стали дуже важливим елементом сучасної культури.

Від маленьких простих платформерів в ігрових автоматах відеоігри перетворились на якісні високобюджетні комп'ютерні ігри. В 2020 році близько 35 процентів людей привернула людей до індустрії ігор. Якщо людина не приймає участі в самої гри, але вона може спостерігати за грою інших гравців, тобто бути учасником кіберспорту.

З кожним роком кількість користувачів тільки зростає. Тому є попит на створення і випуск нових ігор, які б відповідали потребам суспільства.

Метою дослідження бакалаврської роботи є розробка комп'ютерної гри з реалізацією механізму адаптації.

Задачами бакалаврської роботи є:

–огляд існуючих реалізацій подібних програм;

–проаналізувати предметну область комп'ютерних ігор, їх різновиди;

–огляд та аналіз існуючих адаптаційних алгоритмів;

–вивчити існуючі сучасні інструменти та технології розробки комп'ютерних ігор, які використовуються при їх розробці;

–розробити комп'ютерну гру в жанрі «шутер від третьої особи», де головним завданням гравця буде знищити супротивників за певну кількість часу.

Об'єктом дослідження бакалаврської роботи є реалізація комп'ютерної гри за допомогою ігрового движка.

Предметом дослідження є методи реалізації механізму адаптації та комп'ютерна гра в жанрі шутер від третьої особи. Це гра, де гравцю потрібно відстрілювати супротивників за певний час.

Методи дослідження. Аналіз та порівняння інформації з інтернет-мережі та літературних джерел.

Практичне значення одержаних результатів. Отримані результати можуть бути використані для вдосконалення подібних комп’ютерних програм.

Результат роботи:

- проведено аналіз існуючих методів до розробки комп’ютерної гри в жанрі шутер;
- побудована логіка ігрового штучного інтелекту на основі дереві поведінки;
- розроблена комп’ютерна гра в жанрі шутер від третьої особи з реалізованим механізмом адаптації.

Бакалаврська робота складається зі вступу, трьох розділів, висновків, списку використаних посилань із 28 найменувань, 48 рисунків. Загальний обсяг роботи становить 48 сторінок.

РОЗДІЛ 1
ОГЛЯД СТАНУ ПИТАННЯ ТА ДЕТАЛІЗАЦІЯ ЗАДАЧ РОЗРОБКИ

1.1 Аналіз комп'ютерних ігор як об'єкту дослідження

Відеогра – це гра, для використання якої використовувались аудіовізуальні засоби для створення віртуального середовища, в якому гравець може грати.

Відеоігри найпопулярніший спосіб розваги та відпочинку. Це втеча від реального світу, хоча б на деякий час, та поринути в захоплюючий «віртуальний Всесвіт». [2] Відеоігри вже перестали бути дрібничками, які такими вважались на початку свого зародження, а перетворились на окремий вид мистецтва, з величезною кількістю шанувальників.

Ігри супроводжують людство на протязі усього його існування. Спочатку були спортивні, потім словесні, пізніше настільні [6]

З розвитком електроніки з'явились комп'ютерна ігри [7].

Перша відеогра була створена в 1958 році, фізиком Уільямом Хігінботем. Він хотів створити інтерактивний спосіб взаємодії відвідувачів Наукової лабораторії Брукхейвена. Гра «Теніс для двох» стала тоді проривом в лабораторії. Велика кількість відвідувачів хотіли пограти в гру, яка тоді виглядала, як тонкі сині лінії на маленькому екрані осцилографу. Але це було щось незвичне на той час і викликало неабияку хвилю інтересу до нової розробки.

Комп'ютерна гра – це комп'ютерна програма, яка написана на мові програмування, з використанням ігрового двигуна. [1]

На сьогоднішній час комп'ютерна ігри поділяються на різні категорії, або за жанрами, залежно від того, як гравці взаємодіють з грою: кількість гравців, способи взаємодії гравців, тип платформи.

Жанр відеоігор – це категорія ігор, зі схожими характеристиками ігрового процесу. Жанр відеогри залежить від того, як гравець взаємодіє в ігровому світі.

Тип платформи: на ігровому автоматі, браузерна гра, мобільні ігри, для ігрових консолей та персональних комп'ютерів [4].

Кількість гравців: одиночна гра (singleplayer), спільна гра на одному пристрої (splitscreen), для великої кількості гравців (multiplayer), онлайн гра для декількох пристроїв (MMO).

Графічне зображення: двомірна графіка, тривимірна графіка, віртуальна реальність, доповнена реальність, від першої особи, від третьої особи.

На теперішній час немає чіткого та однозначного розподілу комп'ютерних ігор до якогось одного жанру. Одна гра може поєднувати в собі характерні ознаки декількох різних жанрів. Тому, щоб мати якесь конкретне уявлення про вид гри, жанри ігор також поділяються на піджанри [7].

Класифікація ігор за жанрами та піджанрами:

а) action - це ігри, в яких гравець знаходиться в центрі дій з фізичними випробуваннями, які потрібно подолати.

Через те, що в ці ігри можна швидко почати грати, вони досі залишаються найпопулярнішими серед гравців.

- платформер – персонаж взаємодіє з платформами, тобто бігає, стрибає, падає, увесь час ігрового процесу.

Наприклад, Super Mario Bros. та Donkey Kong.

- шутер – використання зброї проти супротивника.

Шутери поділяються на такі піджанри:

1) Шутер від першої особи (First-person Shooter, FPS) – гравець знаходиться, нібито на місці персонажа. Наприклад, Doom, Counter-Strike та Call of Duty.

2) Шутер від третьої особи (Third-person Shooter, TPS) – гравець бачить персонажа з боку (зазвичай з-за спини). Наприклад, Fortnite, Max Payne та Pubg, *Grand Theft Auto 5*.

3) Шутер з видом зверху – гравець бачить ігровий процес над головою персонажа. Наприклад, Diablo та Fallout.

- файтінг – гра-поєдинок, де потрібно битися, зазвичай, рукопашним боєм. В таких іграх є вибір персонажа з певним унікальним типом бою. Наприклад, Mortal Combat та Super Smash Bros.

- бійки – персонаж б'ється з потоком супротивників. Наприклад, Double Dragon та Teenage Mutant Ninja Turtles 2.

- стелс-ігри – гравці уникають виявлення своєї присутності супротивником та атакують раптово. Наприклад, Hitman 2: Blood Money, Batman: Arkham Asylum та Assassins Creed 2.

б) action-travel – квести з перешкодами, які треба подолати за допомогою отриманого предмета. Це також ігри, ігровий жанр яких важко виявити.

- пригодницький бойовик – гравець розв'язує головоломки, але боїв небагато. Наприклад, The Legend of Zelda.

- виживання в кошмарі – гра сприяє виклику хвилювання та напруження, через зображення жахливих сцен з кров'ю. Наприклад, Resident Evil Village, Little Nightmares 2 та The Last of Us.

в) квест (броділки) – персонаж взаємодіє з іншими персонажами для вирішення головоломок з підказками. Наприклад, Minecraft та Machinarium.

г) рольові– ігри, в яким переважно фантазійне оточення.

- масові рольові онлайн-ігри, розраховані на багато користувачів (MMORPG) - взаємодія великої кількості гравців в одній грі, з постійно існуючим ігровим світом. Наприклад, World of Warcraft та The Elder Scrolls.

д) симуляційні – імітація реального світу (або вигаданого), для імітації реальної ситуації.

- містопобудування – побудова та управління містом. Наприклад, Tropico 1-5, Cities: Skylines.

- моделювання життя – управляння віртуальним життям персонажа. Наприклад, Sims 1-4, Youtubers Life та Spore.

- транспортного засобу – переважно створюються для навчання управлінню транспортом для реального життя. Наприклад, Farming Simulator.

е) стратегії – основані на традиційних настільних іграх.

- стратегія в реальному часі (RTS) – збір та підтримка ресурсів, для розвитку бойових одиниць. Наприклад, Age of Empires.

1.2 Особливості комп'ютерних ігор в жанрі шутер

1.2.1 Класифікація шутерів

Шутер – гравці використовують зброю для знищення супротивників. В таких іграх потрібна швидка реакція.

1) Шутер від першої особи (First-person Shooter, FPS) – гравець знаходиться, нібито на місці персонажа. Наприклад, Doom, Counter-Strike та Call of Duty.

Історія появи перших FPS починається напочатку 1970 років. Першою спробою створити FPS можна вважати гру, в якій декілька гравців могли переміщуватись по тривимірному лабіринту на одну клітинку за раз, та стріляти в інших гравців (у вигляді шарів). Ця розробка була виповнена в дослідницькому центрі NASA Ames Research Center на комп'ютері Maze War в 1973 році, розробником Стівом Коллі [8].

SpySim («Космічний симулятор») з'явився на комп'ютері PLATO в 1974 році. В грі гравцю потрібно керувати космічним кораблем .

Взагалі історія будь-якої відеогри починається з аркадних автоматів. Тоді, на початку 1980-х років це був, напевно, чи не єдиний спосіб пограти в відеогру.

Першою аркадною грою FPS можна вважати Battlezone 1980 року. Гравці опиняються на смертельних гусеницях штурмового танку, який переміщується в будь-якому напрямку, по полю з геометричними фігурами та ворогами.

Після розробки Джоном Кармаком концепції raycasting Catacomb 3D, змушуючи комп'ютер зображувати те, що гравець може бачити, а не все, що знаходиться навкруги нього, тривимірні ігри стали на нову ступінь розвитку. Бо до розробки Джона Кармака, тривимірна графіка використовувалась переважно в симуляторах польоту (авіасимулятор).

Кармак розробив простий тривимірний ігровий двигун, в якому для відображення супротивників використовувались анімаційні двумірні спрайти – графічні об'єкті в комп'ютерній графіці.

В 1992 році з’являється Wolfenstein 3D. В грі гравці біли переміщені в голову шпіона Уільяма Бласковича, який вбивав в тюрьмі нацистів. Розробники Джон Ромеро и Джон Кармак, з дизайнером Том Холл таким чином створили новий жанр, тобто прообраз сучасного FPS.

Після успіху Wolfenstein 3D , було вирішено розробляти гру, яка буде ще масштабніше та страшніше. На ігровому движку Джона Кармака, з використанням Hovertank 3D (для рендерінга) та Catacomb 3D (для віображення текстур на поверхні), з ідеєю вбивати демонів, з’явилась гра DOOM.

Перший шутер від першої особи з використанням тривимірної графіки DOOM з’явився в 1993 році.

Шутер Gun Buster, який з’явився в один рік з DOOM, був першим FPS, який керувався за допомогою джойстика для руху з зброєю для прицілювання та стрільби.

В 1996 році з’являється Quake. Гра, в якій ігрове середовище було повністю тривимірним з концепцією «стрибків на ракетах», при якої гравці переміщуються в повітря при вибухівки зброї.

В 1998 році компанія Epic випускає гру Unreal, яка була побудована на ігровому двигуні Unreal. В подальшому ігровий движок перетворився в надійнішу програму для розробки ігор.

В 1999 році з’являється гра Counter-Strike. В ній терористи-супротивники борються один з одним. Те, що відрізняло цю гру від інших, була відсутність відродження після вбивства. Тобто, гравець вже не міг повернутись в гру, поки не закінчиться поточний раунд. Тому через таку появу змагання, гра стала дуже популярною серед гравців.

Після DOOM, створеного на ігровому двигуні Джона Кармака, інші розробники створили свої тривимірні ігрові двигуни, які вони б могли використовувати повторно.

В 2000 році вперше була випущена гра FPS (шутер від першої особи) TimeSplitter для консолі PlayStation 2. Цей шутер мав редактор рівнів, введення якого в склад гри, дало їй більше можливостей для проходження.

Гра Far Cry, яка з’явилась в 2004 році, на ігровому двигуні CryEngine, була з високою якістю реалістичної графіки.

2) Шутер від третьої особи (Third-person Shooter, TPS).

Перші ігри жанру TPS з’явились на початку 1980-х років. Гру, де потрібно стріляти та персонаж присутні на екрані, можна вважати FPS. Але вигляд ззаду персонажа був непоширений до появи тривимірної графіки.

В іграх TPS гравець спостерігає за діями свого персонажа [9].

Є декілька типів розміщення камер для створення виду сцени TPS.

1) Зафіксовані камери, положення яких гравець ніяк не може змінити. В ранніх іграх прийом розміщення фіксованих камер для створення напруженого стану сцени.

2) Слідкування камери за персонажем, в той момент коли він рухається.

3) Інтерактивні камери слідкують за персонажем, але дають змогу керувати положенням камери.

В сучасних іграх використовуються переважно інтерактивні камери.

В жанрі шутера від першої особи (FPS) гравцю дається зброя та дається завдання вбити супротивників. В грі переважна кількість ігрового часу віддається боям з використанням різних типів зброї.

В жанрі шутера від третьої особи (TPS) також, як і в FPS багато часу віддається відстрілюванню супротивників. Але прицілювання може бути ускладнене тому, що персонаж гравця або предмети в грі можуть заважати огляду локації гри.

1.2.2 Ігрова механіка в шутері

Ігрова механіка – це набір правил та способів, за допомогою яких реалізується віртуальна взаємодія гравця та гри. Ігрові механіки формують реалізацію ігрового процесу в грі [11].

Базові елементи ігрової механіки [12].

- Механіка
- Повторення
- Зміна

В шутерах є велика кількість ігрових механік. Це біг, стрибки, стрільба, зміна зброї, смерть.

Тобто, що буде відбуватися з об’єктами гри, в залежності від дій над ними.

Головний персонаж має механіки: ходити, стрибати, бігати, використання зброї, предметів.

Супротивник має механіки: ходити, атакувати персонажа.

- Movement – переміщення в пространстві
- Jumping – стрибки
- Rotation – повертання
- Targeting – прицілювання
- Shooting – постріл в ціль.
- Temporary – зміна часу
- Act – зміна стану ігрового середовища
- Collect – збір ресурсів
- Reduction – зменшення розміру життя, кількості очок
- Elimination – знищення (смерть) об’єкта

1.2.3. Проблеми та недоліки в шутерах

Шутери, так само як інші відеоігри, не уникають проблем з швидкодії ігрового процесу уникаючи збої та підвисання. Наприклад, якщо шутер з клієнт-серверною взаємодією, тоді виникають недоліки через поганий зв’язок в мережі і це може вплинути на відгук гри.

Взагалі, шутер потребує великих апаратних можливостей пристрою. Будь то персональний комп’ютер або мобільний пристрій. Від цього буде залежати швидкодія гри та якість графіки.

1.3 Специфікація задач розробки

Розробка гри складається з декількох основних етапів.

1. Підготовка
2. Preproduction
3. Production
4. Реліз

На етапі підготовки формується ідея гри. Тобто, її опис:

- тип платформи, для якої робиться гра
- жанр
- ігровий простір (оточення в грі), де буде проходити процес гри
- опис правил гри
- зображення моделей об’єктів
- основні механіки
- цілі гравця

На етапі preproduction створюється консепт (документацію) гри. Тобто це зображення того, як буде виглядати гра після розробки. На цьому етапі описується суть гри, ключові елементи та дії, які потребуються від гравця.

Етап production – це процес розробки гри.

Реліз – це фінальний етап, на якому гра проходить тестування, оптимізацію під різні типи платформ.

З технічної точки зору гра складається з таких частин:

1. Логіка гри
2. Графіка
3. Інтерфейс
4. Карти гри
5. Фізика ігрового світу
6. Звук
7. Збереження та обробка даних (якщо гра клієнт-серверна)

Засоби розробки та рішення при створенні гри:

1. Дизайн гри
2. Вибір платформи
3. Вибір ігрового двигуна
4. Ассети гри (ресурси, об’єкти)

Вибір ігрового двигуна важливий етап, тому що це програмне забезпечення, яке містить багато готових рішень потрібних в процесі розробки гри, такі як:

редактор карт, систему штучного інтелекту. Вибір ігрового двигуна залежить від мови програмування, на якій буде писатись код.

Висновки до розділу 1

Історія відеоігор не перевищує 65 років. Але за цей час ігрова індустрія зробила великий стрибок в своєму розвитку. Це вже не просто розвага, а окремий вид мистецтва з величезною аудиторією шанувальників. З’явились інструменти розробки, які допомагають оптимізувати процес виробництва комп’ютерних ігор та дали змогу створювати ігри не тільки великим компаніям, але також і незалежним розробникам.

РОЗДІЛ 2
АЛГОРИТМ ФУНКЦІЙ ГРИ З РЕАЛІЗАЦІЄЮ МЕХАНІЗМА АДАПТАЦІЇ

2.1 Алгоритми реалізації ігрової механіки

Ігрова механіка – це правила, які визначають поведінку гравця в грі. В грі не може бути тільки одна механіка. Це, як правило, комбінація з декількох ігрових механік, які визначають поведінку гравця, рівень складності [14].

В шутері це механіки переміщення, прицілу зброї, відновлення здоров'я.

Логіка механіки переміщення персонажа гри (рис.2.1):

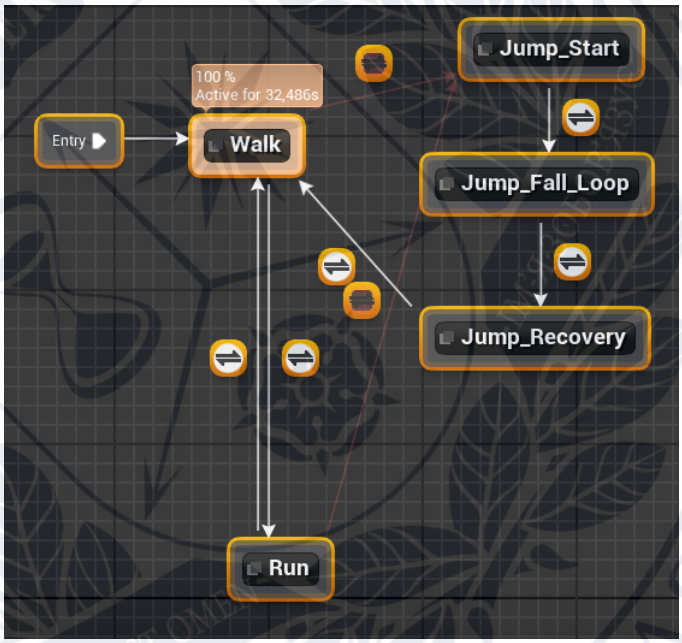

Рисунок 2.1 – Логіка руху персонажа в Blueprint

Приклад коду, який відповідає за прискорення руху в грі шутер (рис.2.2):

```
float USPlayCharacterMovementComponent::GetMaxSpeed() const
{
    const float MaxSpeed = Super::GetMaxSpeed();
    const ASPlayCharacter* Player = Cast<ASPlayCharacter>(GetPawnOwner());
    return Player && Player-> IsRunning() ? MaxSpeed * RunModifier :
MaxSpeed;
}
```

Рисунок 2.2 – Код механіки бігу персонажа

2.2 Стратегії та алгоритми прийняття рішень в ході гри

В відеоіграх існує елемент прийняття рішень. Прийняття рішень пов’язано зі змінами, які трапляються в ігровому світі. Наприклад, в шутері це прийняття рішення стосовно того, потрібно робити перезарядку зброї або ще ні. Якщо кількість ресурсів обмежена, тоді потрібно вирішити, які дії потрібно зробити, щоб отримати виграш, та не загинути. Це стосується як кількості заряду зброї, так і кількості часу до кінця гри, або приближення супротивника.

В шутері важливо приймати рішення, які залежать від дій супротивника.

За алгоритми прийняття рішень в відеоіграх відповідає ігровий штучний інтелект [15]. Штучний інтелект в відеогрі використовується для створення моделі поведінки для неігрових персонажів (NPC – non-player character). Супротивники зі штучним інтелектом потрібна для ускладнення рівня гри, моделювання різної поведінки руху та ігрових ситуацій, які залежать від дій гравця. В відеоіграх використовуються такі методи, як пошук шляху та дерева рішень, щоб реалізувати дії неігрового персонажа. Ігровий штучний інтелект відрізняється від «реального», бо його підхід в відеоіграх реалізується для ігрового процесу в віртуальному середовищі.

Створення штучного інтелекту для персонажів або для інших створінь в проекті гри здійснюється за допомогою декількох систем, які працюють разом.

Дерево поведінки – це дерево ієрархічних вузлів, які керують процесом прийняття рішень об’єкта.

Дерево рішень складається з листя та гілок:

- листя – це команди, які керують об’єктом;

- гілки – це різні типи вузлів, які керують рухом штучного інтелекту по деревам, щоб достигнути послідовності команд, які найкраще підійдуть до певної ситуації.

Спосіб реалізації поведінки в дереві виконується створенням умовних позначень як самостійні вузли (декоратори), які прикріплюються до іншого вузла[15]. Цей спосіб відрізняється від класичного, за яким вузли листя розташовуються на кінцях гілок. Тобто, умова поведінки не поділяється на більш

маленькі гілки, ліва частина яких повертає true або false, повертаючись, в залежності від умови, на рівень вище по дереву поведінки.

Логіка вмикається залежно від подій, через декоратори (вузли), які базуються на подіях та через доступ переміщення по дереву далі.

В грі дерево поведінки будується для супротивників.

Дерево поведінки складається з одного класу, але потребує також два допоміжні класи [16].

Перший клас – це клас саме дерева поведінки Behavior Tree. Дерево містить вузли та послідовність виконання вузлів дерева. Кожен вузел містить в собі візуальний клас, який місить код та внутрішні функції.

Кожне дерево поведінки об’єднується з класом Blackboard, який містить змінні для використання дерева поведінки. Ці змінні розміщуються в вузли, в якості аргументів та використовуються для виконання логіки в грі.

Наприклад, для відображення логіки стрільби, використовується дерево поведінки (рис.2.3):

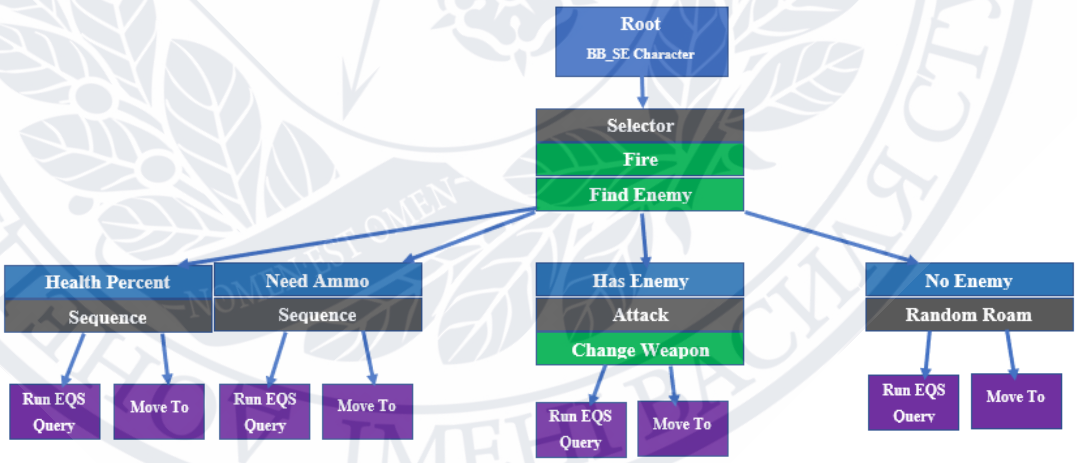

Рисунок 2.3 – Дерево поведінки персонажа для виконання пострілу

Вузли декоратора це умовні вираження, які обчислюють змінні та повертають значення true або false. Вузел декоратора розміщується на вузлі задачі або діє як умова для вузлів, які розміщуються нижче. Тобто, від значення

true або false визначається, чи може продовжуватись переміщення вниз по піддереву задач.

2.3 Адаптаційні алгоритми на основі дерева поведінки

В 1990-х роках починають використовувати інструменти ігрового штучного інтелекту, які мають назву кінцеві автомати (finite-state machine) [15]. Потім з'являються дерева поведінки (Behavior tree). Вони використовувались як інструмент для створення моделі поведінки неігрового персонажу. Головна різниця між кінцевим автоматом та деревом поведінки в тому, що в дереві поведінки головним є задача, а не поточний стан. Також, через більшу зрозумілість дерева поведінки менш схильні до помилок в реалізації. Тому частіше їх використовують в розробці відеоігор.

Дерево поведінки – орієнтований ациклічний граф [16].

Поведінка в грі будується через реалізацію дій неігрового персонажа та проектування структури дерева (рис.2.4):

1. Кінцеві вузли дерева – це дії,
2. Внутрішні вузли – це прийняття рішення неігрового персонажа.

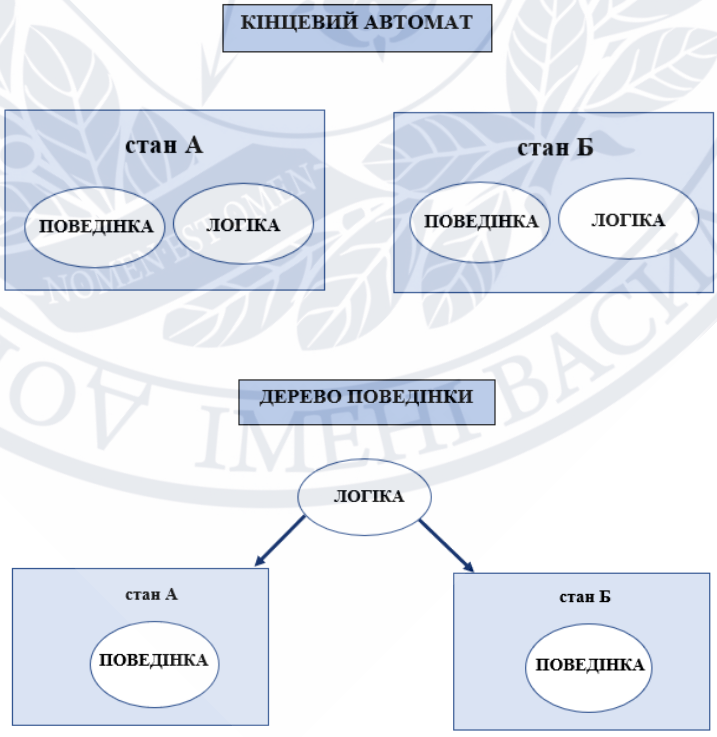

Рисунок 2.4 – Різниця між двома методами ігрового штучного інтелекту

При підході побудування моделі поведінки кінцевого автомата, кожному агенту (персонажу) зіставляється багато станів, які змінюються під впливом зовнішніх факторів [17]. І в кожному стані визначається послідовність дій по досягненню цілі, яка закінчується переміщенням агенту в інший стан.

Майже кожна гра містить ігровий штучний інтелект. Він керує поведінкою ботів. Це супротивники або напарники в грі, якими «керує» комп'ютер. Це прописані інструкції для ботів NPC. В цих інструкціях для ботів неігрових персонажів описана поведінка, як реагувати на навколишній ігровий світ.

Наприклад, такий алгоритм поведінки:

Бот (супротивник) патрулює територію, постійно переміщуючись від точки А в точку Б. Якщо бот помічає гравця, тоді виконується алгоритм поведінки. Бот буде атакувати, або спробує сховатись, або буде йти за гравцем, тобто переміщуватись до координат місцезнаходження гравця.

Інструкції поведінки залежно від ситуації будують дерево поведінки. Тобто, мережу, яка об'єднує задачі-логічні вузли (if…then). Цими задачами є алгоритми реагування на подію в ігровому процесі: алгоритми переміщення, алгоритми атаки.

Математична модель дерева поведінки. Дерево поведінки це три кортежі [20] (2.1.):

$$Ti = \{f_i,\ r_i, \Delta t\} \tag{2.1}$$

де $i \in \mathbb{N}$ - індекс дерева, $f_i : \mathbb{R}^n \to \mathbb{R}^n$ - права частина рівняння, Δt – час кроку, $r_i : \mathbb{R}^n \to \{R, S, F\}$ – повернення стану яке дорівнює Робота (R), Успіх (S), Невдача (F). Нехай область Робота (Ri), Успіх (Si) та Невдача (Fi) відповідає розподілу стану, яке визначається (2.2):

$$\begin{aligned} R\text{i} &= \{\, x :\ \ r_i(x) = R\} \\ S\text{i} &= \{\, x :\ \ r_i(x) = S\} \\ F\text{i} &= \{\, x :\ \ r_i(x) = F\}. \end{aligned} \tag{2.2}$$

Нехай $x_k = x(t_k)$ це стан часу t_k , тоді виконання дерева рішення Ti – різниця рівнянь (2.3):

$$x_{k+1} = f_i(x_k),$$

$$t_{k+1} = t_k + \Delta t. \quad (2.3)$$

Повернення стану r_i використовуємо при рекурсівному об'єднанні дерева рішень.

Дерево поведінки реалізує дерево рішень наступним чином (рис.2.5):

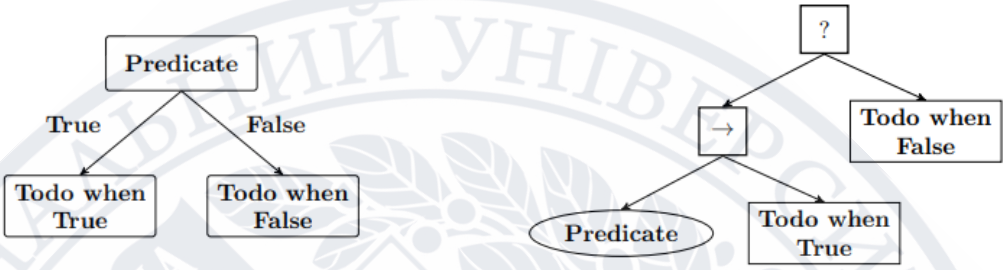

Рисунок 2.5 – Генерація дерева поведінки з дерева рішень

Базовими блока дерева рішень є твердження «Якщо…тоді…інакше» (зліва (рис.2.5)), та побудовані в дереві поведінки (справа ((рис.2.5)) [18].

Приклад дерева поведінки для атаки супротивника, яке представлене набіром вузлів, які розташовані в дереві (рис.2.6):

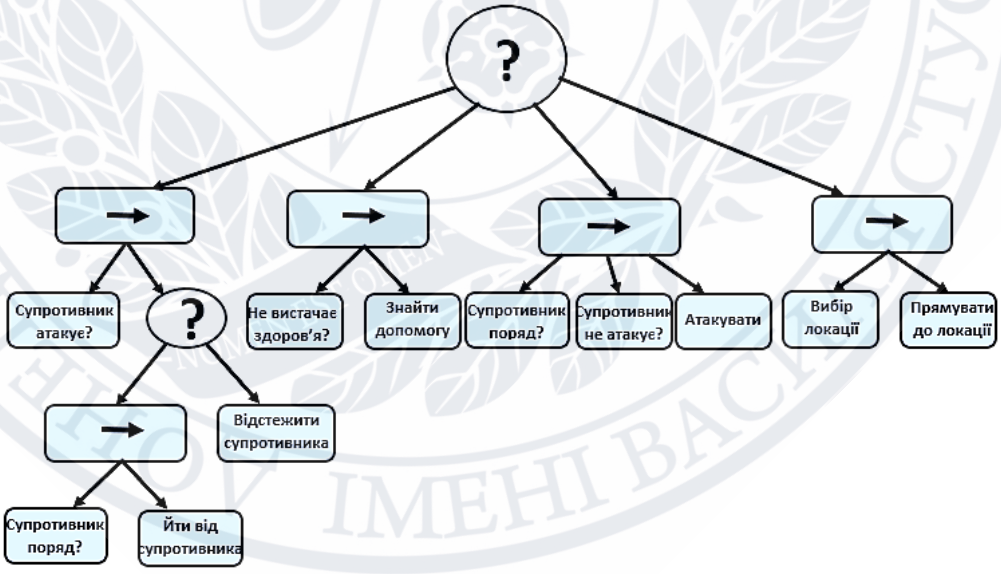

Рисунок 2.6 – Модель дерева поведінки атаки супротивника

Дерево поведінки починається з ROOT вузла та оброблюється за допомогою пошуку в глибину [18]. Це виконання поведінки здійснюється на кожному кроку процесу ігрового штучного інтелекту (рис.2.7).

Для переміщення в просторі гри використовується система переміщення по навігаційній сітки – NavMesh (Navigation System). Ця навігація для ігрового штучного інтелекту використовується для:

–переміщення ботів (супротивників) зі штучним інтелектом за допомогою функції Move To;

–автоматичне переміщення до вказаної точки Target Point;

–переслідування гравця ботом (супротивником);

–переміщення ботів між великою кількістю точок патрулювання;

–переміщення ботів між рандомними точками.

Також для навігації на ігровому полі використовується система Environment Quary System (EQS) [19]. Ця система запитів середовища відповідає за побудову сітки на площині, для:

– пошуку бота (NPC);

– виявлення останньої точки на локації, де знаходився персонаж.

Система на основі результатів запитів, які отримала з середовища ігрової локації, приймає рішення як діяти. Наприклад, це може використовуватись для знаходження сховища персонажа та здоров’я, або патронів для зброї.

Бот (супротивник), який використовує ігровий штучний інтелект, рандомним чином переміщується по ігровому полю, поки не побачить персонажа гравця. Тоді використовується EQS для пошуку місця в середовищі гри, яке посприяє кращої точки огляду на відстані. Це потрібно, наприклад, для організації далекобійної атаки, коли потрібно зберігати відстань від супротивників.

На ігровому полі EQS представляється як кольорові сфери [20]:

- Зелені та червоні сфери вказують на рівень відповідності до запиті;
- Сині сфери вказують на невдачу, при якій не був відібрано результат запиту.

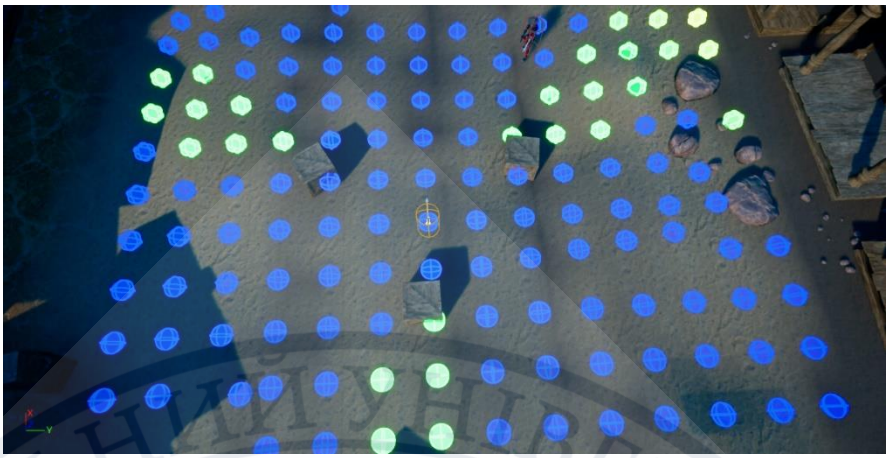

Рисунок 2.7 – Використання EQS в Unreal Engine

Висновки до розділу 2

Для розробки гри в жанрі шутер від третьої особи був обраний алгоритм, на якому базується побудова логіки поведінки супротивників – дерево поведінки. Також були розглянуті внутрішні інструменти Navigation System для пошуку шляху супротивника в локації гри та допоміжну підсистему штучного інтелекту Environment Quary System, яка допомагає персонажу аналізувати простір та при заданих значеннях відбирати локації гри, які підходять найбільше.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ ГРИ З РЕАЛІЗАЦІЄЮ МЕХАНІЗМУ АДАПТАЦІЇ

3.1 Архітектура і особливості засобів розробки

Для розробки комп'ютерної гри в жанрі шутер від третьої особи «Guns and Enemies» використовувалися ігровий движок Unreal Engine 4 [21], об'єктно-орієнтована мова програмування C++, також графічна мова програмування Blueprint, яка базується на C++ та використовується в ігровому двигуні Unreal Engine. Та використовується інтегроване середовище розробки програмного забезпечення Microsoft редактор коду Visual Studio Code[22].

Unreal Engine 4 – ігровий «движок» (набір інструментів для розробки ігор), який був створений компанією Epic Games. В 1998 році була створена гра шутер від першої особи Unreal [23]. Спочатку движок використовували для створення шутерів від першої особи, але наступні версії Unreal Engine використовували при розробці ігор різних жанрів – файтинг, MMROPG. Також движок використовується в рекламі, кіноіндустрії, архітектурі.

Переваги:

- велика кількість інструментів розробки;
- розвинене співтовариство через велику кількість розробників, які користуються движком;
- велика кількість документації, асетів, плагінів;
- техпідтримка механізму оновлення, постійний розвиток движка;
- сумісність майже зі всіма платформами, починаючи від ПК та консолей до мобільних пристроїв та HTML5 [26];
- з версії Unreal Engine 4 (2015) движок став безкоштовним, але розробники повинні передавати 5% від виручки з продажі гри компанії Epic Games, якщо дохід перевищує 3000 дол [24];
- низький поріг входження, якщо був досвід роботи в графічних редакторах розробки Unity3D, Maya [26].

Недоліки:

- висока ціна для асетів в офіційному магазині движка AssetStore Marketplace.

- потребує великих ресурсів ПК.

3.2 Реалізація модулів гри

3.2.1 Модуль «Налаштування проекту гри»

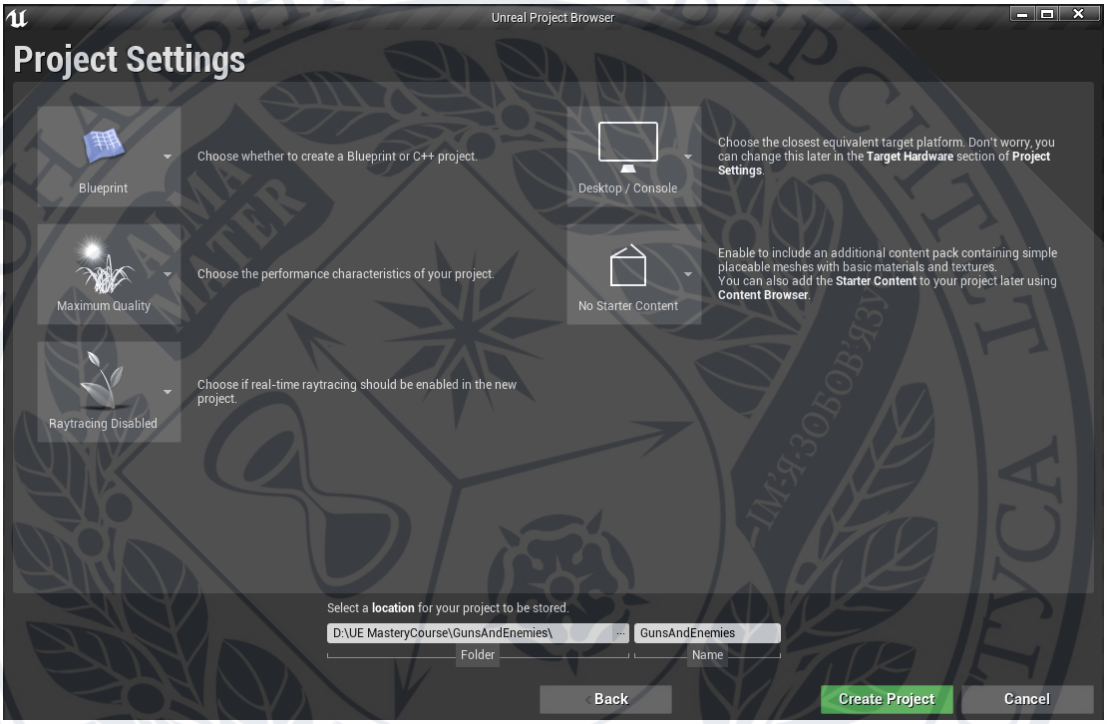

Рисунок 3.1 – Вікно налаштувань проекту в редакторі Unreal Engine

При налаштування проекту були обрані категорія **Games** та порожній шаблон **Blank** (рис.3.1).

Створюємо основний C++ клас в проекті гри – GAEGameModeBase. Заголовочні файли розміщюються в папці public, а C++ файли в папці private (рис.3.3 – 3.4).

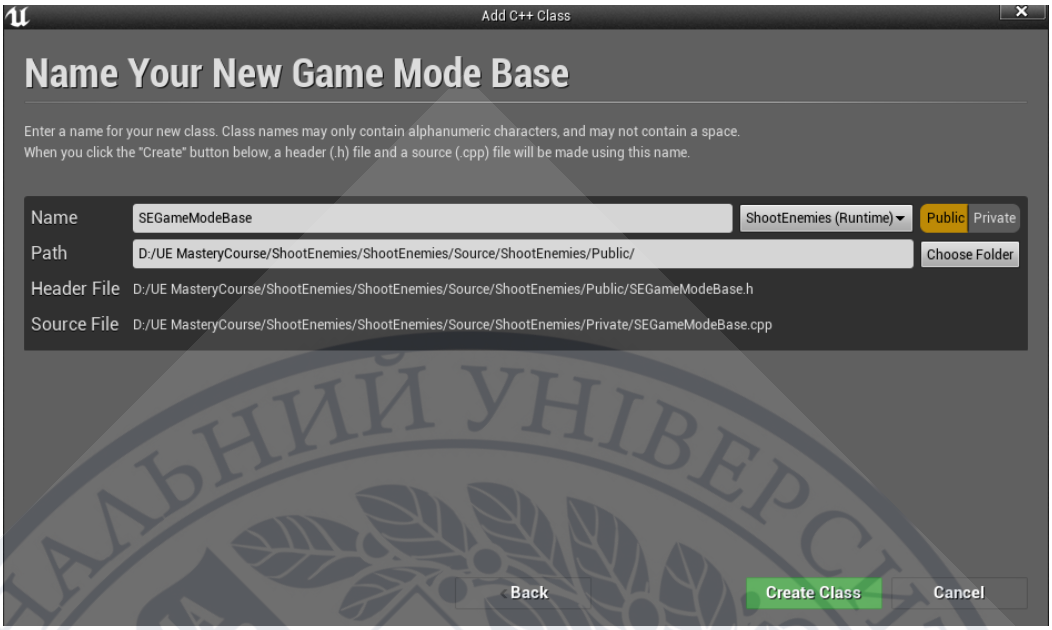

Рисунок 3.3 – Вікно створення C++ класу

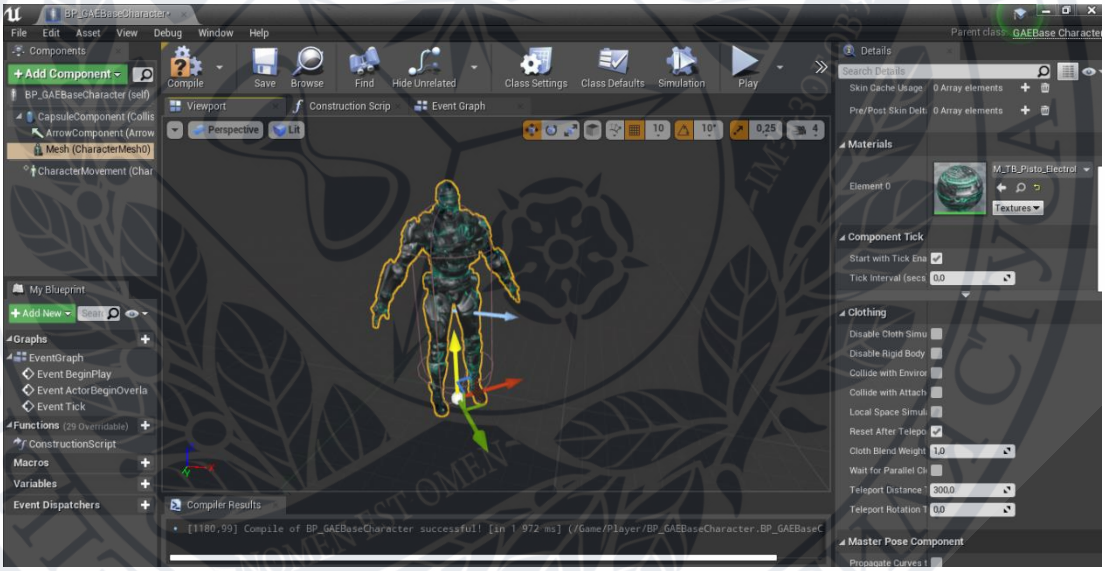

Рисунок 3.4 – Вікно представлення SkeletalMesh

Ця модель персонажа була імпортована з AssetStore[2]

2. https://www.unrealengine.com/marketplace/en-US/product/paragon-twinblast

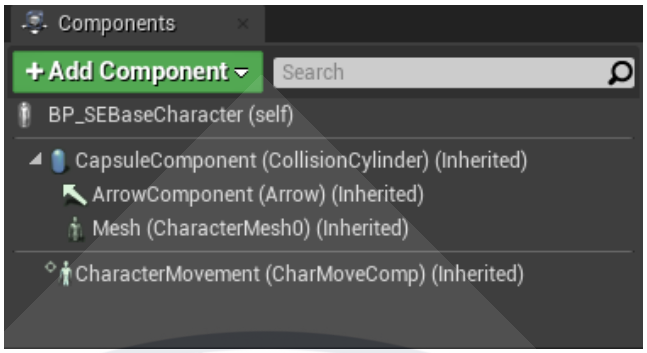

Рисунок 3.5 – Створення Mesh персонажа

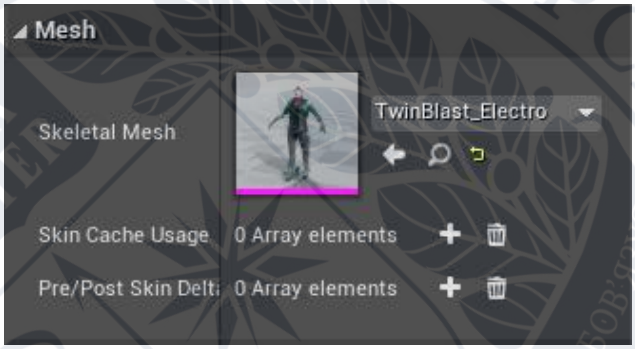

Рисунок 3.6 – Завантаження ассету Mesh персонажа

При додаванні Mesh персонажу (рис.3.5-3.6), з ним завантажуються текстури матеріалу, які накладені на модель персонажа (рис.3.7 – 3.8).

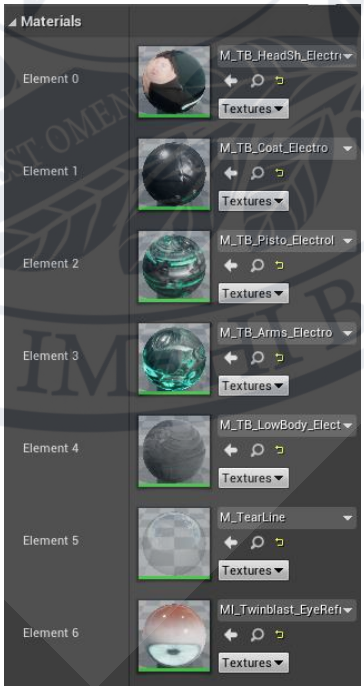

Рисунок 3.7 – Текстури Mesh персонажа

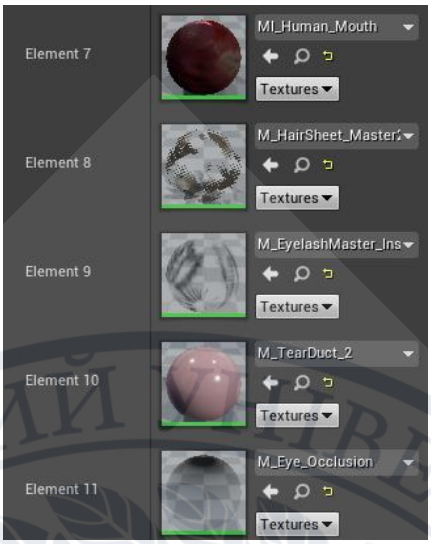

Рисунок 3.8 – Текстури Mesh персонажа

Створюємо класи GAEBaseCharacter та GAEPlayerController.

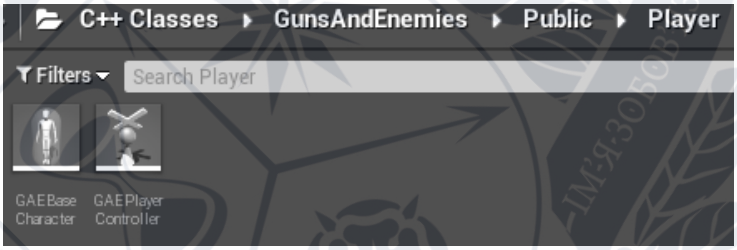

Рисунок 3.9 – Створені класи GAECharacter та GAEPlayerController

Ці класи потрібні, щоб налаштовувати анімацію персонажа.

Налаштування камери (CameraComponent)[25].

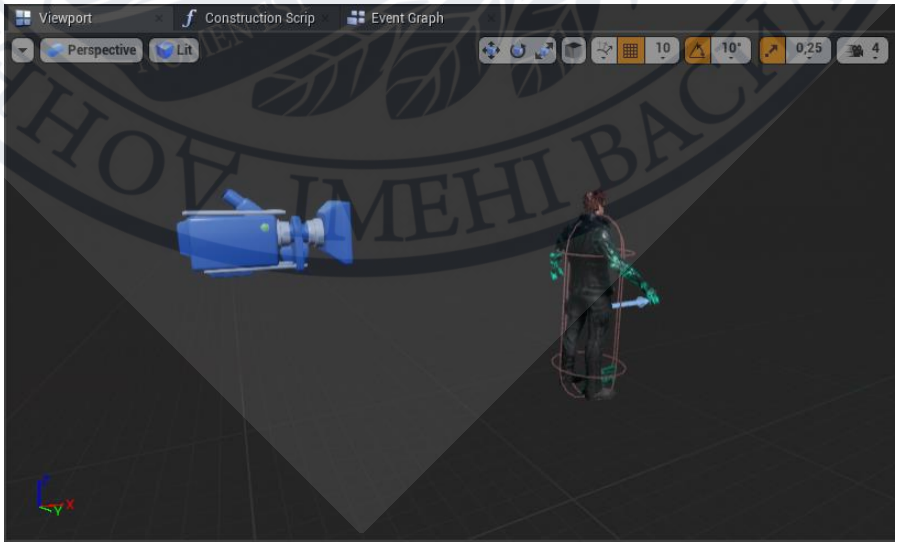

Рисунок 3.10 - Налаштування камери

3.2.2 Модуль «Реалізація анімації руху персонажа»

Для обробки руху в Unreal Engine використовується Input axis mapping. Це механізм для зручного відображення вісей з поведінкою ввода, шляхом вставки рівня непрямого звернення між поведінкою вводу та клавішами, які його викликають.

Рисунок 3.11 – Налаштування axis mappings персонажа

Управління персонажем, залежно від того який буде input (натискання на клавішу, рух мишки або натискання на її кнопки). Ці налаштування проводяться в класі проекту гри GAEBaseCharacter.

```
void AGAEBaseCharacter::MoveForward(float Amount)
{
	AddMovementInput(GetActorForwardVector(), Amount);
}

void AGAEBaseCharacter::MoveRight(float Amount)
{
	AddMovementInput(GetActorRightVector(), Amount);
}
```

Повертання камери навколо персонажа при взаємодії з мишкою. Створюємо axis mapping для камери.

Рисунок 3.12 – Налаштування axis mappings камери

Код управління мишкою при повертанні.

GAEBaseCharacter.cpp

```
void AGAEBaseCharacter::SideTurns(float Amount)
{
	AddControllerYawInput(Amount);
}
void AGAEBaseCharacter::LookingUp(float Amount)
{
	AddControllerPitchInput(Amount);
}
```

Створюємо анімацію персонажа. Для цього створюється компонент CharacterMovement. Налаштування анімації проводиться в Blueprints.

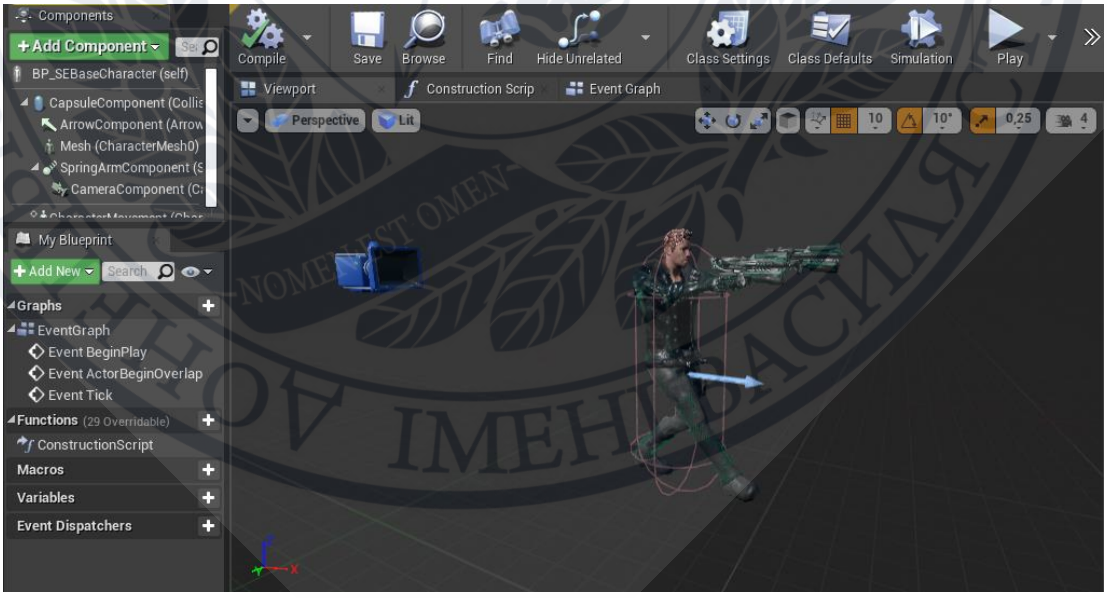

Рисунок 3.13 – Анімація переміщення персонажа та управління камерою

Робимо зміну анімації персонажа залежно від його швидкості.

Функція Get Velocity повертає поточну швидкість персонажа по кожній з осей XYZ. Тип повертаємого значення F vector. Щоб отримати значення довжини F вектора, треба використати функцію VectorLength.

Функція Tick. Нода Print String виводить виклик функції на екран гри.

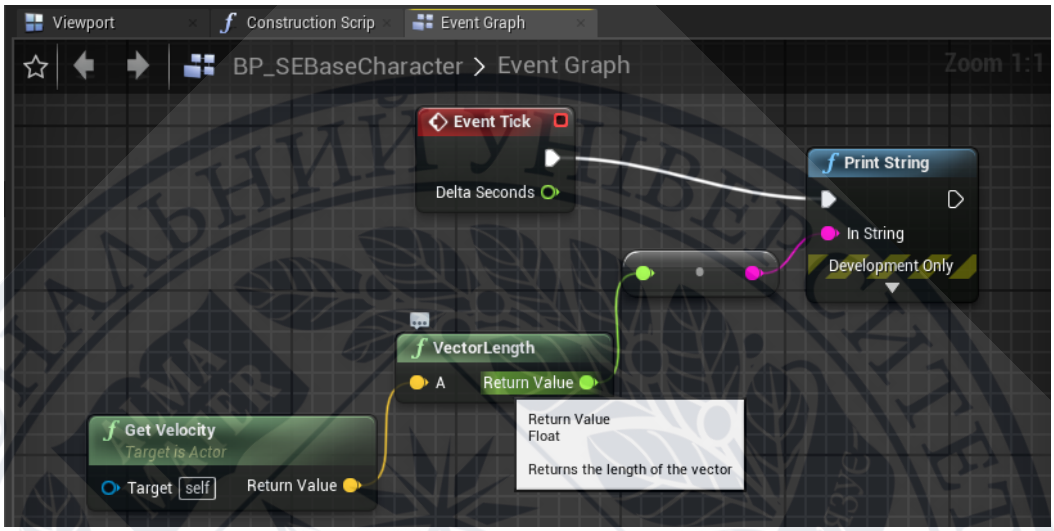

Рисунок 3.14 – Налаштування зміни швидкості персонажа в Blueprints

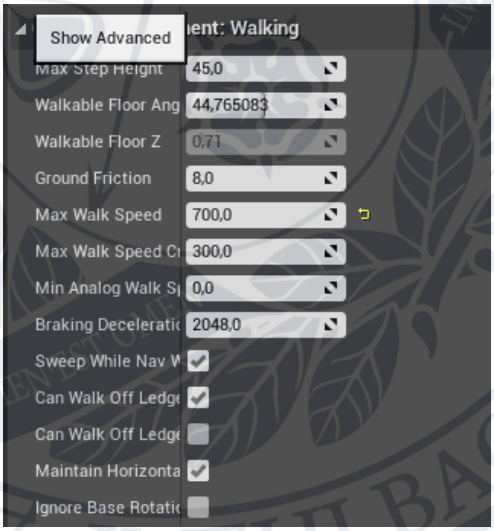

Рисунок 3.15 – Встановлення параметрів швидкості руху персонажа

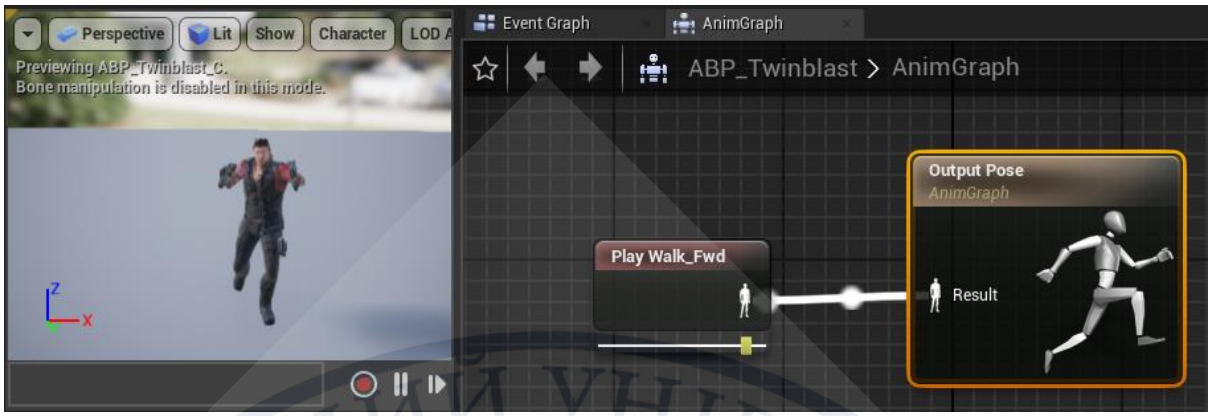

Рисунок 3.16 – Вікно зображення анімації ходьби персонажа з заданною швидкістю

Для повільного руху зі стану спокою в стан швидкої ходьби створюємо додатковий тип ассету Blend Space.

Рисунок 3.17 – Вікно з вибором ассета Blend Space

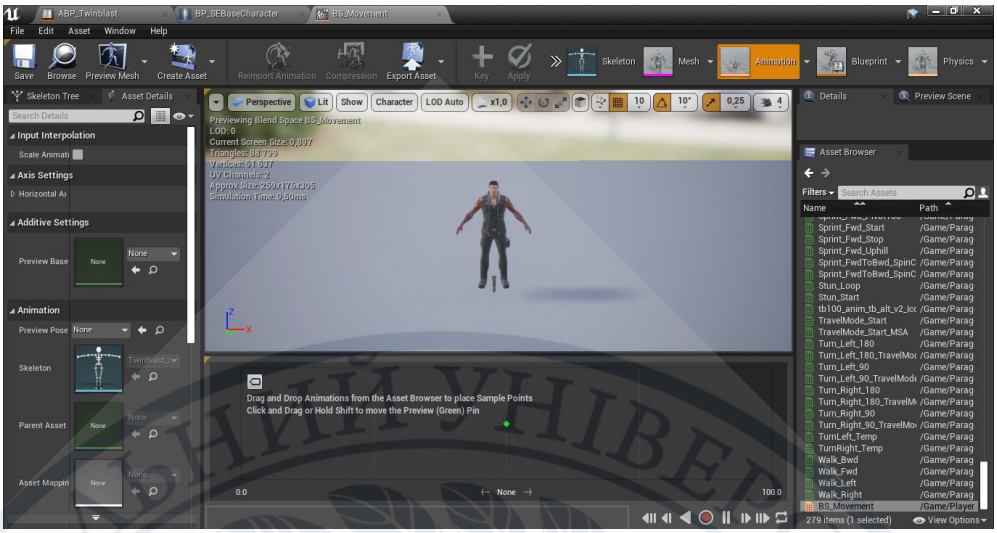

Рисунок 3.18 – Вікно ассета BS_Movement

Цей ассет потрібен для повільності зміни від одної анімації до іншої. Він видає проміжний результат між анімаційними позами.

В вікні axis settings змінюємо швидкість

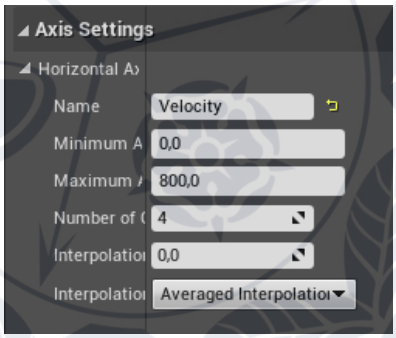

Рисунок 3.19 – Зміна швидкості в вікні axis settings

В анімаційному вікні Blueprint налагоджуємо зв’язки між змінною швидкості Velocity та асетом BS_Movement.

Нода Try Get Pawn Owner повертає посилання на поточне значення BaseCharacter. VectorLenght – вектор швидкості

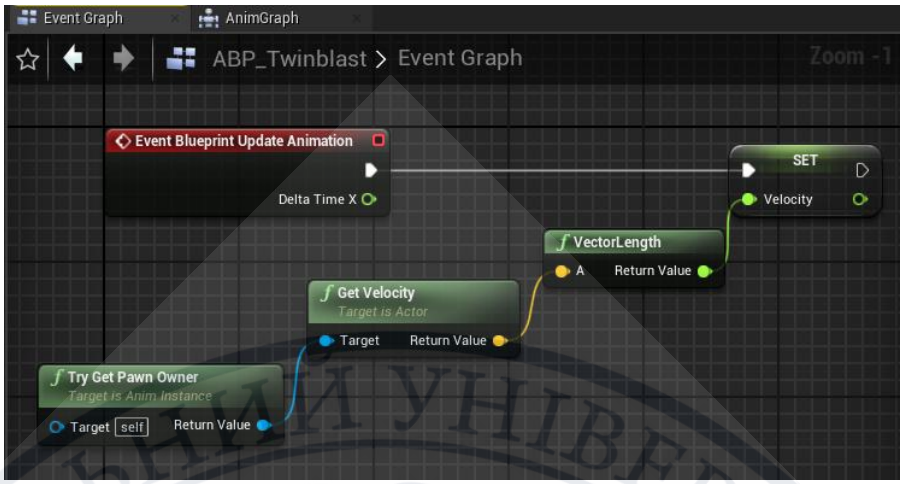

Рисунок 3.20 – Вікно зміни налаштувань анімації швидкості

Для додавання анімації стрибка створюємо Action Mappings – Jump

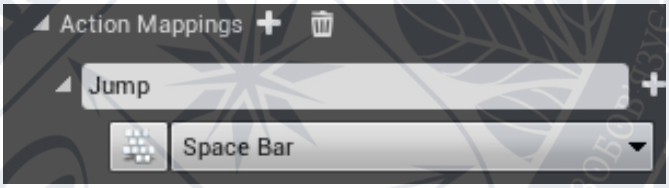

Рисунок 3.21 – Вікно додавання клавіши стрибка Space

Створюємо Binding. Програмуємо код логіки стрибка.

В класі GAEBaseCharacter.cpp в функції розміщуємо код в функції:

void AGAEBaseCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)

Налаштуємо анімацію стрибка.

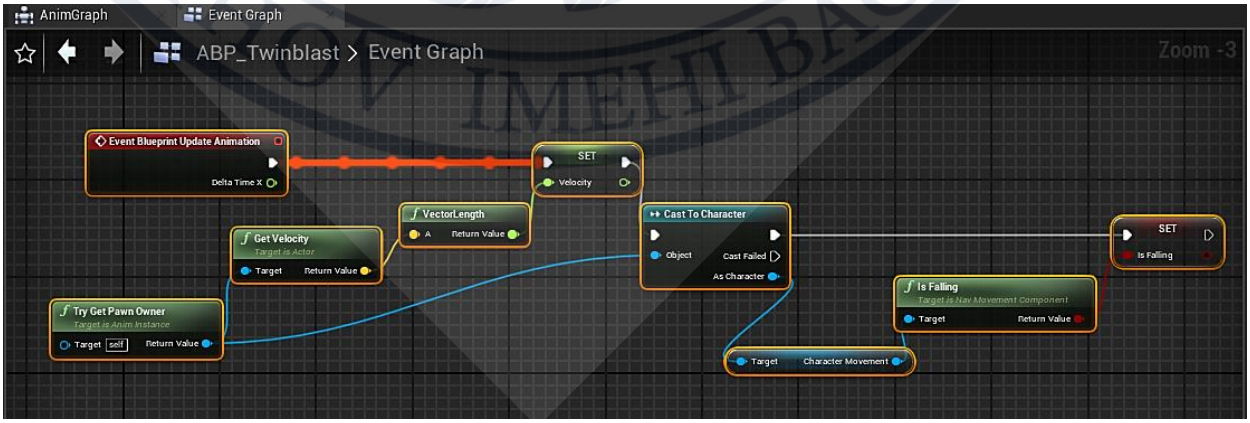

Рисунок 3.22 – Налаштування анімації в Blueprint

Додаємо ноду State machine в AnimGraph. Нода State Machine потрібна для управління декількома анімаціями та налаштовувати переходи між ними.

Нода буде відповідати за різні анімації пересування персонажа в грі.

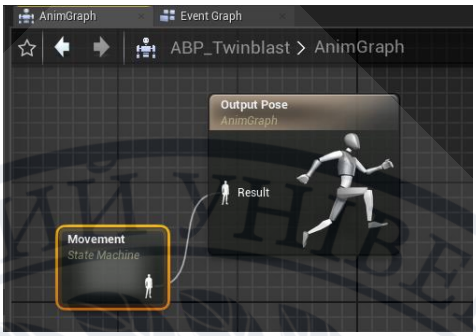

Рисунок 3.23 – Додавання ноди state machine – Movement в Blueprints.

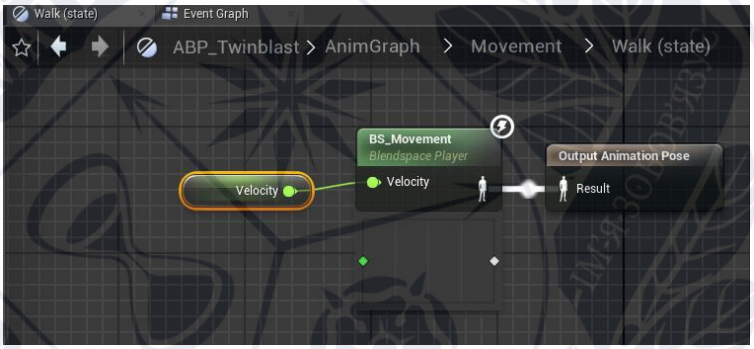

Рисунок 3.24 – Налаштування анімації руху

Побудування графу переходу з одного анімаційного стану (Walk) в інший (Jump). Між станами є умови для переходу зі стану в стан.

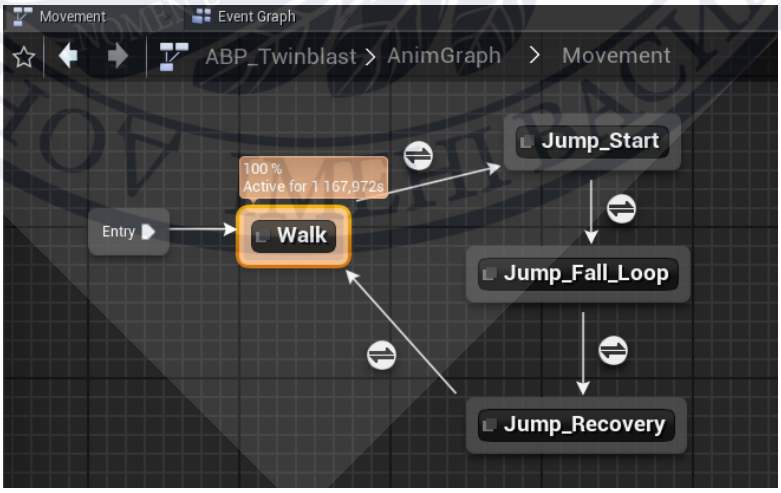

Рисунок 3.25 – Налаштування state machine Movement

Створюємо action mapping Run. При натисканні клавіши Left Shift на клавіатурі, персонаж буде бігти.

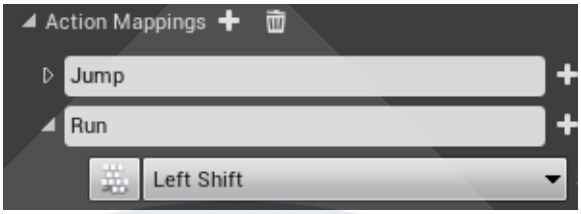

Рисунок 3.26 – Налаштування action mapping для бігу

Додаємо в файлі GAEBaseCharacter.cpp в функції AGAEBaseCharacter::SetupPlayerInputComponent змінні натискання клавіш бігу

Щоб визначити, що персонаж буде бігти вперед. Можна вичислити вектор швидкості та вектор куди поглядає персонаж, та при вичіслюванні угла визначати куди біжить персонаж.

Додаємо функцію для Blueprint в файлі GAEBaseCharacter.h (коли персонаж біжить або ні)

```
UFUNCTION(BlueprintCallable, Category = "MoveTo")
        bool ToRun() const;
```

Прописуємо в файлі GAEBaseCharacter.cpp функцію бігу залежно від значення вектора швидкості.

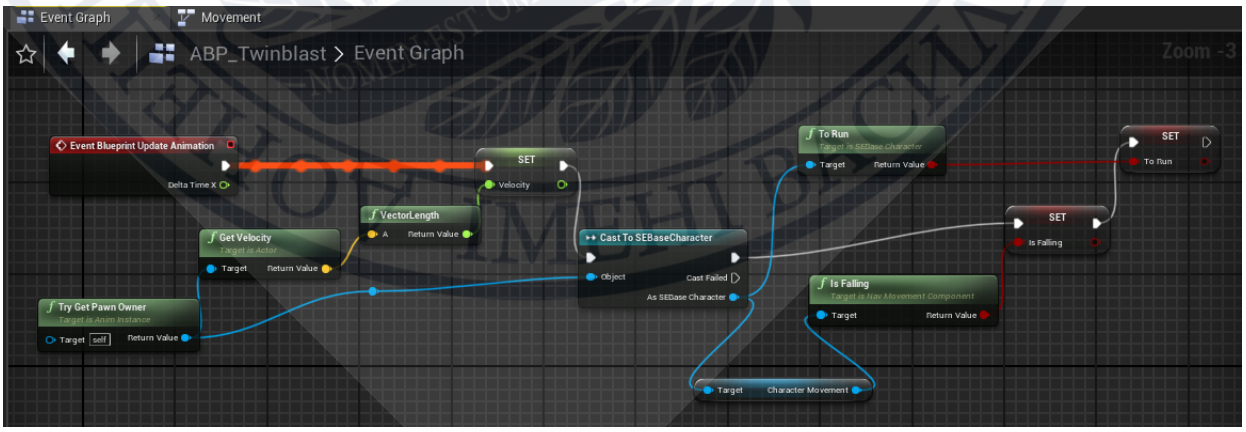

Рисунок 3.27 – Додавання кавіши бігу вперед

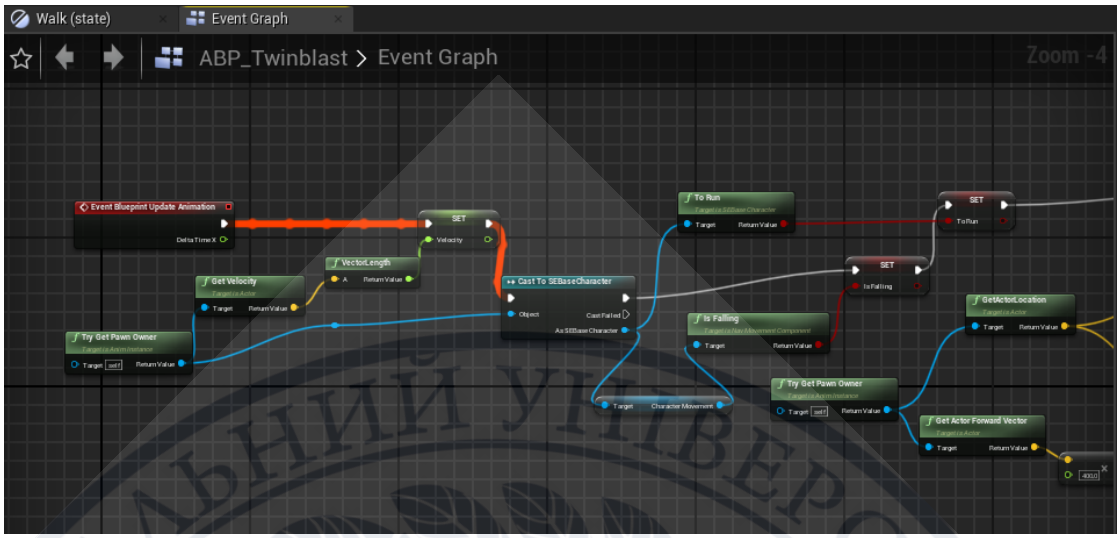

Рисунок 3.28 – Анімація персонажа

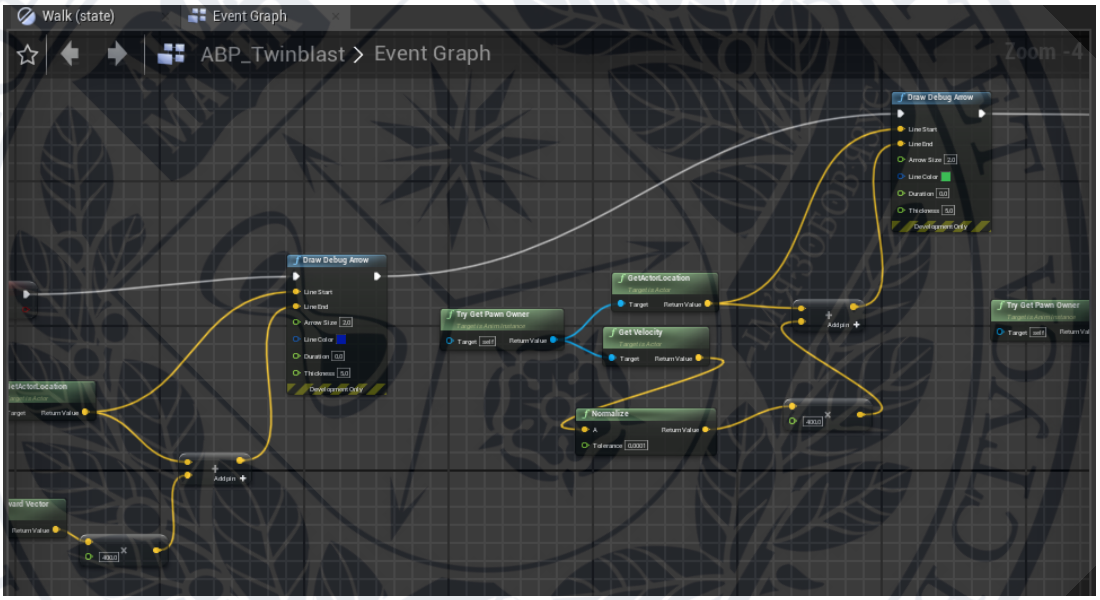

Рисунок 3.29 – Анімація персонажа

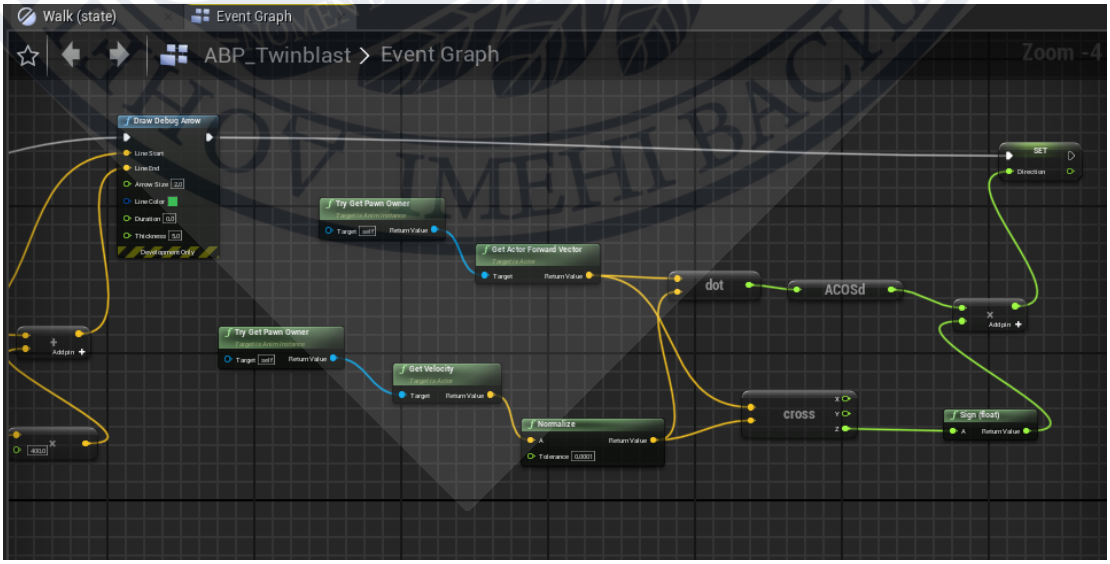

Рисунок 3.30 – Анімація персонажа

3.2.3 Модуль «Здоров’я персонажа»

Для створення компоненту здоров’я персонажа, в Unreal Engine створюємо клас GAEHealthComponent, в якому прописуються всі взаємодії зі станом здоров’я персонажа. Наприклад, нанесення пошкодження здоров’ю.

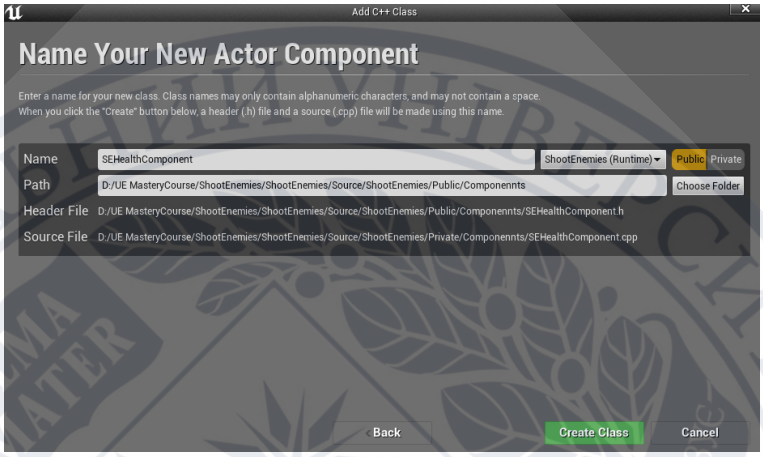

Рисунок 3.31 – Створення класу здоров’я GAEHealthComponent

Приклад коду для опису нанесення пошкодження здоров’ю та виведення результату про смерть персонажа реалізується в функції:

void UGAEHealthComponent::OnTakeAnyDamage

Також, щоб було виведено вікно паузи, після смерті персонажа, створюємо клас SpectatorPawn.

Наведено код умови налаштування контролю:

```
if (Controller)
{Controller->ChangeState(NAME_Spectating); }
```

Щоб відновити здоров’я персонажа після пошкодження застосуємо логіку відновлення здоров’я в функції:

void UGAEHealthComponent::HealUpdate().

Якщо трапилось падіння з висоти, тоді персонаж отримує пошкодження здоров’я.

3.2.4 Модуль «Зброя»

Створюємо клас зброї GAEPlayWeapon.

Рисунок 3.32 – Створення класу зброї

Для додавання функції прицілювання створюємо клас GAEPlayHUD .

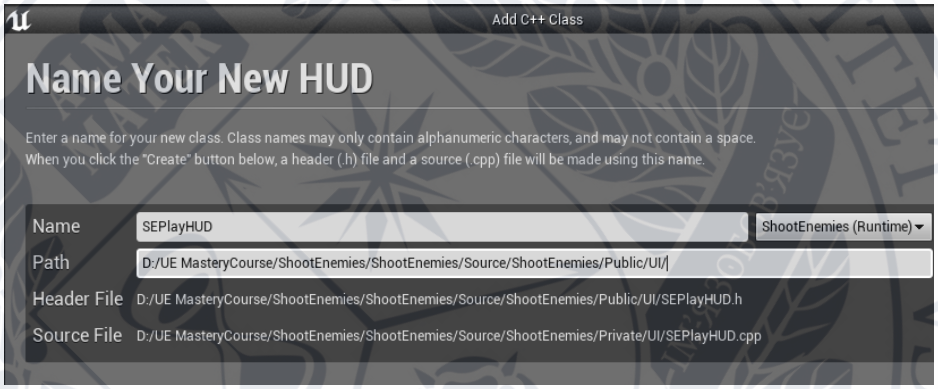

Рисунок 3.33 – Створення класу HUD

Створюємо WeaponComponent. Всю логіку для зброї помістимо в цей клас. Та створимо вхідну точку EntryPoint для стрільби. Створюємо також Binding Action Mapping – Fire. При натисканні лівої клавіши мишки, гравець зможе стріляти зі зброї.

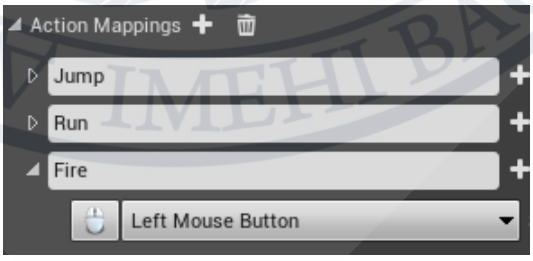

Рисунок 3.34 – Action Mapping – Fire (Mouse Button)

Рисунок 3.35 – Клас WeaponComponent

Реалізовуємо стрільбу за таймером в функціях:

void UGAEWeaponComponent::StartFire() та

void UGAEWeaponComponent::StopFire().

Додаємо також два класи додаткової зброї: гвинтівки та гранатомета. Логіку пострілу прописуємо в класах GAERifleWeapon та GAELauncherWeapon.

Створюємо функції для додавання набору зброї:

void UGAEWeaponComponent::AttachWeaponToSocket.

3.2.5 Модуль «Створення віджетів»

В Blueprints BP_GAEGameHUD створюємо віджет шкали здоров’я персонажа WBP_HealthBar. Налаштування віджетів проводиться в класі GAEPlayerHUDWidget.

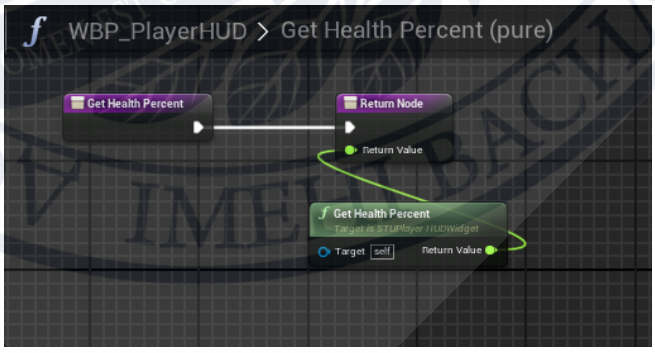

Рисунок 3.36 – Налаштування функції віджета здоров’я

Віджет прицілу зброї та для відображення зображення зброї та кількості патронів налаштовується в класах GAEWeaponComponent та GAEPlayerHUDWidget.

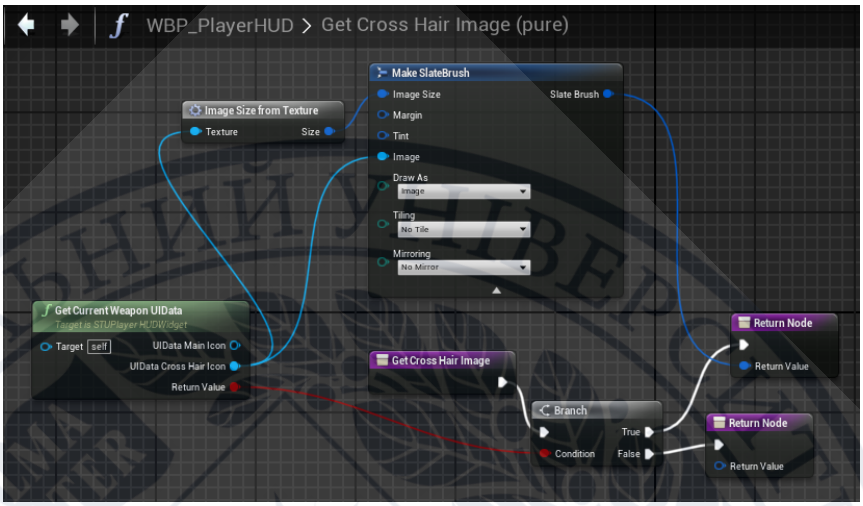

Рисунок 3.37 – Налаштування функції віджету прицілу зброї

3.2.6 Модуль «Штучний інтелект»

Для налаштування штучного інтелекту ботів (супротивників), було створено два класи GAEAICharacter та GAEAIController. Ці класи успадковуються від початкових класів GAEBaseCharacter та GAEPlayerController.

Додана функція MoveTo, яка переміщує персонажа до вказаних координат ігрового поля. Щоб персонаж між пересуватись, було додано Nav Mesh Bound Volume. Цей об’єкт визначає простір рівня гри, в якому персонаж буде пересуватись.

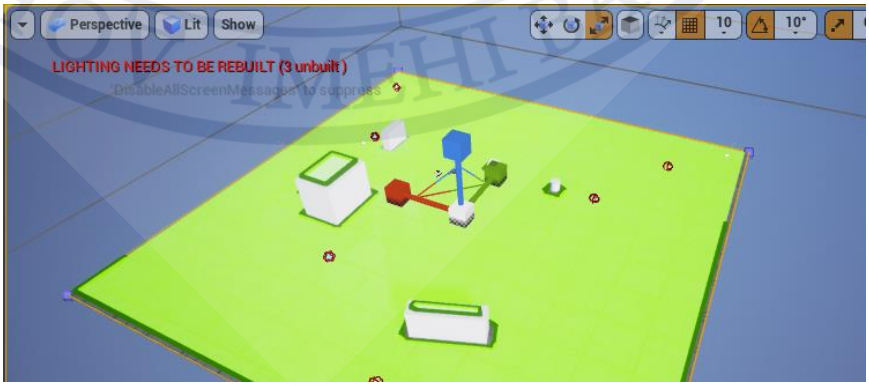

Рисунок 3.38 – Налаштування Nav Mesh Bound Volume

Для створення штучного інтелекту для персонажа було побудовано дерево поведінки – Behavior Tree. Для цього були створені асети BT_GAECharacter (Behavior Tree) та BB_GAECharacter (Blackboard). В них налаштовується логіка поведінки персонажа. Blackboard – це набір змінних, на зміну яких реагує Behavior Tree. Дерево поведінки складається з вузлів (ноди керування) та листя – завдання, в яких описується дія персонажа.

Нода ROOT точка входу дерева поведінки. З неї робиться вибір однієї з трьох нод (вузлів) керування [27].

Selector (селектор) – виконує завдання зліва направо та зупиняє їх виконання, коли один з них виконується успішно.

Sequence – нода, яка забезпечує послідовне виконання декількох завдань (Task) зліва направо. Якщо завдання не виконується, тоді виконання завдань перерветься та повертається до Sequence і виконання завдань поновлюється.

Simple Parallel – нода, яка дозволяє одному завданню ноди головної задачі бути виконаним поряд з другим піддеревом. Тобто, паралельно виконується дві задачі- головна та додаткова [26].

Після вибору ноди, можна обрати завдання (Task). Task - Mode To, визначає пошук шляху до певної локації. Task – Wait – персонаж ніяк не реагує, йде час очікування наступної дії.

Дерево виконується з верху, починаючи з ноди ROOT та пересувається далі вниз, виконуючи завдання з лівої частини дерева на право. Також додається пріоритет виконання завдання. Якщо завдання було виконане успішно, тоді виконується наступне. Якщо завдання не виконалось, тоді ланцюг переривається.

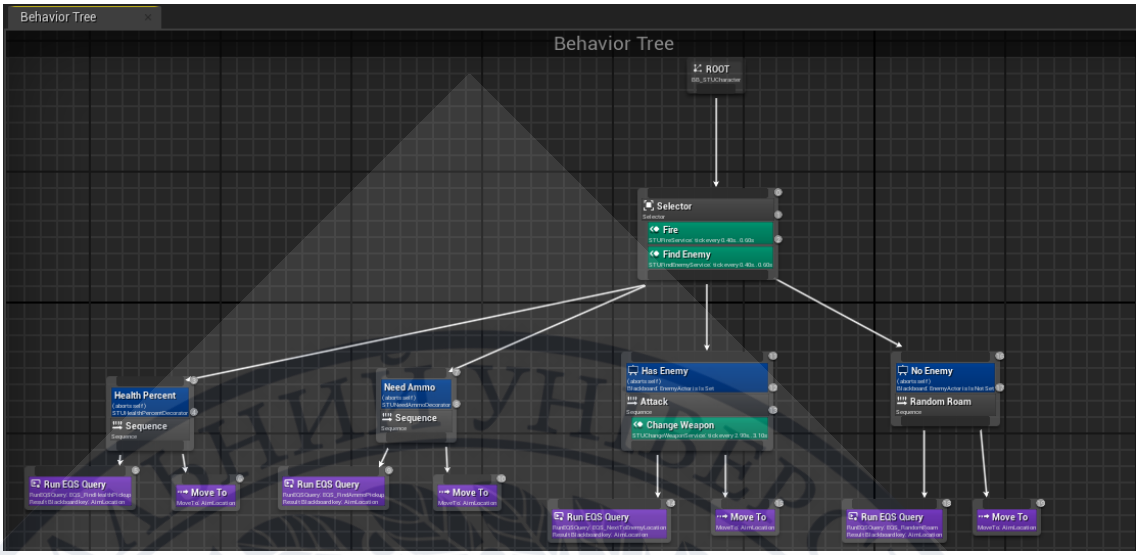

Рисунок 3.39 – Дерево поведінки персонаж

Переміщення персонажа-бота було прописано в класі GAENewLocTask.

Також, було використано клас NavigationSystem [28]. Який є навігаційною системою і потрібен для знаходження точки локації. Ця система має в собі алгоритм пошуку шляху, за допомогою якого можна вичислити точку, куди персонаж може переміститись.

3.2.7 Модуль «Ігровий процес»

Блоки програмного забезпечення ігрового процесу

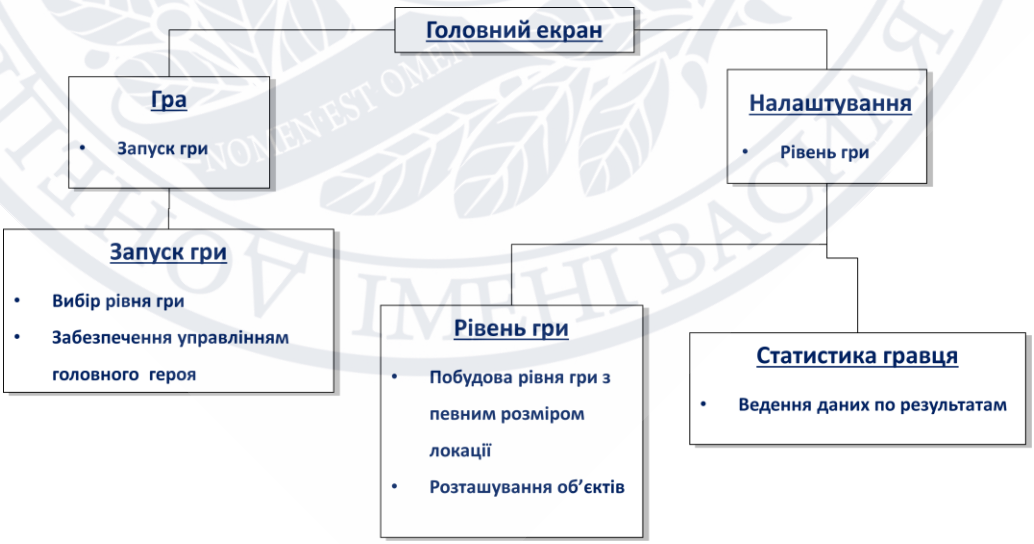

Рисунок 3.40 – Зображення компонентів ігрового процесу

3.3 Інструкція оператору

3.3.1 Тестування програмного забезпечення

Гра створювалась на комп’ютері з такими апаратними даними:

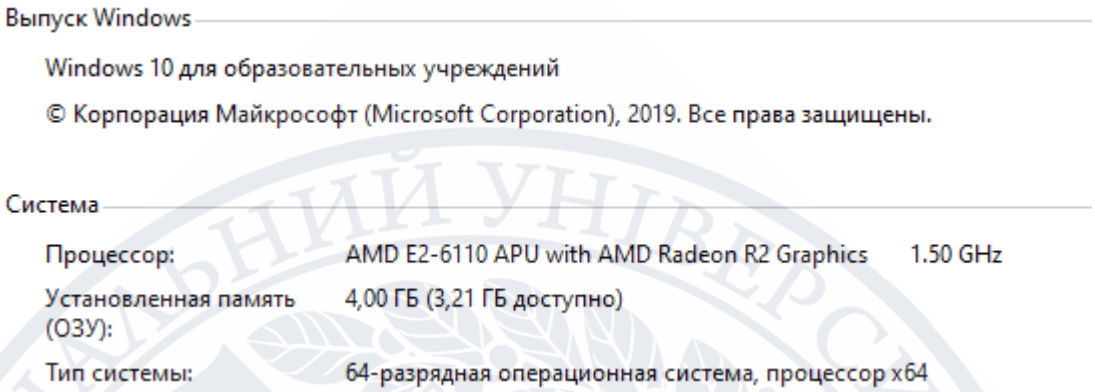

Рисунок 3.41 - Відомості про систему комп’ютера

Це також мінімальні системні вимоги для комп’ютера гравця. В іншому випадку гра буде «виснути» та не буде змоги повноцінно використовувати програму.

Висновки до розділу 3

В ході розробки комп’ютерної гри шутер від третього лиця з реалізацією механізму адаптації було використане ігровий движок Unreal Engine 4 , об’єктно-орієнтована мова програмування C++, також графічна мова програмування Blueprint, яка базується на C++ та використовується в ігровому двигуні Unreal Engine. Та використовується інтегроване середовище розробки програмного забезпечення Microsoft редактор коду Visual Studio Code. Використання цих інструментів розробки спрощує процес розробки ігор.

ВИСНОВКИ

У ході виконання бакалаврської роботи було проведено аналіз предметної області та розглянуті класифікації ігор.

Була розглянута історія появи та розвитку відеоігор в жанрі шутер.

Були розглянуті методи та алгоритми, які використовуються при розробці ігор в жанрі шутер, обрані інструменти для розробки. В ході дослідження була обрана модель штучного інтелекту дерево поведінки, яка використовується при побудові поведінки супротивників в комп'ютерній грі.

Кінцевим результатом бакалаврської роботи є реалізоване програмне забезпечення відеогри в жанрі шутер від третьої особи. Гра дозволяє гравцю обрати рівень гри. В самій грі гравцю потрібно відстрілювати супротивників за певний відлік часу та з певною кількістю ресурсів.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Компьютерная игра. Циклопедия - универсальная нейтральная викиэнциклопедия. URL: http://cyclowiki.org/wiki/Компьютерная_игра (дата звернення: 28.04.2021).
2. Why We Like Video Games (Maybe We're Control Freaks). AARP. URL: https://www.aarp.org/home-family/personal-technology/info-2021/video-games-pastimes.html (дата звернення 28.04.2021).
3. Видео-игры - это искусство! Или?.. Artifex. URL: https://artifex.ru/цифра/видео-игры (дата звернення 28.04.2021).
4. Классификация компьютерных игр. Википедия. URL: https://ru.wikipedia.org/wiki/Классификация_компьютерных_игр (дата звернення: 29.04.2021).
5. Guide to Video Game Genres: 10 Popular Video Game Types. MasterClass. URL: https://www.masterclass.com/articles/guide-to-video-game-genres. (дата звернення: 07.05.2021).
6. Классификация компьютерных игр. Компьютерные игры как искусство. URL: https://gamesisart.ru/game_class_all.html (дата звернення 29.04.2021).
7. The Many Different Types of Video Games & Their Subgenres. iD Tech. URL: https://www.idtech.com/blog/different-types-of-video-game-genres (дата звернення 30.04.2021).
8. The Complete History Of First-Person Shooters. PCMAG. K. Thor Jensen. URL: https://www.pcmag.com/news/the-complete-history-of-first-person-shooters (дата звернення 07.05.2021).
9. First-Person Games vs. Third-Person Games: What Are the Differences? MUO. Ben Stegner. URL: https://www.makeuseof.com/first-person-games-vs-third-person-games-differences (дата звернення 08.05.2021).
10. 10 старых игровых механик шутера, которые должны быть в следующем поколении. Den of Geek. Дэниел Хилл. URL:

https://www.denofgeek.com/games/10-old-shooter-gameplay-mechanics-that-should-be-in-next-gen (дата звернення 08.05.2021).

11. Игровая механика. Википедия. URL: https://ru.wikipedia.org/wiki/Игровая_механика (дата звернення 10.05.2021).
12. Game Mechanics Patterns. Serge Himmelreich. URL: https://www.slideshare.net/flashgamm/game-mechanics-patterns (дата звернення 11.05.2021).
13. Artificial Intelligence. Unreal Engine Documentation. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/ArtificialIntelligence/(дата звернення 03.05.2021).
14. Artificial intelligence in video games. Wikipedia. URL: https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games (дата звернення 14.05.2021).
15. Дерево поведения (искусственный интеллект, робототехника и управление). Википедия. URL: https://datewiki.ru/wiki/Behavior_tree (дата звернення 14.05.2021).
16. А. О. Анохин, Н. П. Садовникова, А. В. Катаев, Д. С. Парыгин. Моделирование поведения агентов для реализации игрового искусственного интеллекта. Прикаспийский журнал: управление и высокие технологии. – 2020. – № 2(50). – С. 85-99.
17. Ögren Petter, Colledanchise, Michele, Behavior Trees in Robotics and AI: An Introduction: CRC Press, 2018. 192 p.
18. Ryan Keith Marcotte. Modelling Artificial Intelligence in Games Using MindSet Behavior Trees: A Thesis Submitted to the Faculty of Graduate Studies and Research, 2017
19. Environment Query System Quick Start. Unreal Engine Documentation. URL: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/ArtificialIntelligence/EQS/EQSQuickStart/ (дата звернення 22.04.2021).

20. Environment Query System Testing Pawn. Документація Unreal Engine 4. URL: https://uedev.blogspot.com/2016/10/eqs-testing-pawn.html (дата звернення 25.04.2021).
21. Офіційний сайт Unreal Engine. URL: https://www.unrealengine.com/en-US/ (дата звернення 17.04.2021).
22. Офіційний сайт Visual Studio Code. URL: https://code.visualstudio.com/ (дата звернення 17.04.2021).
23. Unreal Engine. Википедия. URL: https://ru.wikipedia.org/wiki/Unreal_Engine (дата звернення 28.05.2021).
24. Преимущества Unreal Engine. ArtCraft. URL: https://blog.artcraft.net.ua/page11435532html (дата звернення 28.05.2021).
25. Unreal Engine 4 для игр и прототипирования. VC.RU. Блинцов Александр URL: https://vc.ru/pixonic/51306-ue4-guide (дата звернення 28.05.2021).
26. The Behavior Tree Starter Kit. Gameaipro. Alex J. Champandard, Philip Dunstan. URL:https://www.gameaipro.com/GameAIPro/GameAIPro_Chapter06_The_Behavior_Tree_Starter_Kit.pdf (дата звернення 19.05.2021).
27. Andrew Sanders, An Introduction to Unreal Engine 4: CRC Press, 2017. 270 p.
28. Rachel Cordone, Unreal Engine 4 Game Development Quick Start Guide: Packt Publishing, 2019. 204 p.

ДОДАТКИ

Додаток А. Лістинги програми

Клас GAEGameModeBase.cpp

```
#include "GAEGameModeBase.h"
#include "Player/GAEBaseCharacter.h"
#include "Player/GAEPlayerController.h"
#include "UI/GAEGameHUD.h"
#include "AIController.h"
#include "Player/GAEPlayerState.h"
#include "GAEUtils.h"
#include "Components/GAERespawnComponent.h"
#include "Components/GAEWeaponComponent.h"
#include "EngineUtils.h"

DEFINE_LOG_CATEGORY_STATIC(LogGAEGameModeBase, All, All);
constexpr static int32 MinRoundTimeForRespawn = 10;
AGAEGameModeBase::AGAEGameModeBase()
{
   DefaultPawnClass = AGAEBaseCharacter::StaticClass();
   PlayerControllerClass = AGAEPlayerController::StaticClass();
   HUDClass = AGAEGameHUD::StaticClass();
   PlayerStateClass = AGAEPlayerState::StaticClass();
}
void AGAEGameModeBase::StartPlay()
{
   Super::StartPlay();
   SpawnBots();
   CreateTeamsInfo();
```

```
    CurrentRound = 1;
    StartRound();
    SetMatchState(EGAEMatchState::InProgress);
}
UClass*
AGAEGameModeBase::GetDefaultPawnClassForController_Implementation(ACont
roller* InController)
{
    if (InController && InController->IsA<AAIController>())
    {
        return AIPawnClass;
    }
    return Super::GetDefaultPawnClassForController_Implementation(InController);
}
void AGAEGameModeBase::SpawnBots()
{
    if (!GetWorld()) return;
    for (int32 i = 0; i < GameData.PlayersNum - 1; ++i)
    {
        FActorSpawnParameters SpawnInfo;
        SpawnInfo.SpawnCollisionHandlingOverride =
ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
        const auto GAEAIController = GetWorld()-
>SpawnActor<AAIController>(AIControllerClass, SpawnInfo);
        RestartPlayer(GAEAIController);
    }
}
void AGAEGameModeBase::StartRound()
{
```

```
    RoundCountDown = GameData.RoundTime;
    GetWorldTimerManager().SetTimer(GameRoundTimerHandle, this,
&AGAEGameModeBase::GameTimerUpdate, 1.0f, true);
}
void AGAEGameModeBase::GameTimerUpdate()
{
    // UE_LOG(LogGAEGameModeBase, Display, TEXT("Time: %i / Round:
%i/%i"), RoundCountDown, CurrentRound, GameData.RoundsNum);
    if (--RoundCountDown == 0)
    {
        GetWorldTimerManager().ClearTimer(GameRoundTimerHandle);
        if (CurrentRound + 1 <= GameData.RoundsNum)
        {
            ++CurrentRound;
            ResetPlayers();
            StartRound();
        }
        else
        {
            GameOver();
        }
    }
}
void AGAEGameModeBase::ResetPlayers()
{
    if (!GetWorld()) return;
    for (auto It = GetWorld()->GetControllerIterator(); It; ++It)
    {
        ResetOnePlayer(It->Get());
```

```
    }
}
void AGAEGameModeBase::ResetOnePlayer(AController* Controller)
{
    if (Controller && Controller->GetPawn())
    {
        Controller->GetPawn()->Reset();
    }
    RestartPlayer(Controller);
    SetPlayerColor(Controller);
}
void AGAEGameModeBase::CreateTeamsInfo()
{
    if (!GetWorld()) return;
    int32 TeamID = 1;
    for (auto It = GetWorld()->GetControllerIterator(); It; ++It)
    {
        const auto Controller = It->Get();
        if (!Controller) continue;
        const auto PlayerState = Cast<AGAEPlayerState>(Controller->PlayerState);
        if (!PlayerState) continue;
        PlayerState->SetTeamID(TeamID);
        PlayerState->SetTeamColor(DetermineColorByTeamID(TeamID));
        PlayerState->SetPlayerName(Controller->IsPlayerController() ? "Player" :
"Bot");
        SetPlayerColor(Controller);
        TeamID = TeamID == 1 ? 2 : 1;
    }
```

```
}
FLinearColor AGAEGameModeBase::DetermineColorByTeamID(int32 TeamID)
const
{
   if (TeamID - 1 < GameData.TeamColors.Num())
   {
      return GameData.TeamColors[TeamID - 1];
   }
   UE_LOG(
      LogGAEGameModeBase, Warning, TEXT("No color for team id: %i, set to
default: %s"), TeamID, *GameData.DefaultTeamColor.ToString());
   return GameData.DefaultTeamColor;
}
void AGAEGameModeBase::SetPlayerColor(AController* Controller)
{
   if (!Controller) return;
   const auto Character = Cast<AGAEBaseCharacter>(Controller->GetPawn());
   if (!Character) return;
   const auto PlayerState = Cast<AGAEPlayerState>(Controller->PlayerState);
   if (!PlayerState) return;
   Character->SetPlayerColor(PlayerState->GetTeamColor());
}
void AGAEGameModeBase::Killed(AController* KillerController, AController*
VictimController)
{
   const auto KillerPlayerState = KillerController ?
Cast<AGAEPlayerState>(KillerController->PlayerState) : nullptr;
   const auto VictimPlayerState = VictimController ?
Cast<AGAEPlayerState>(VictimController->PlayerState) : nullptr;
```

```
    if (KillerPlayerState)
    {
        KillerPlayerState->AddKill();
    }
    if (VictimPlayerState)
    {
        VictimPlayerState->AddDeath();
    }
    StartRespawn(VictimController);
}
void AGAEGameModeBase::LogPlayerInfo()
{
    if (!GetWorld()) return;
    for (auto It = GetWorld()->GetControllerIterator(); It; ++It)
    {
        const auto Controller = It->Get();
        if (!Controller) continue;
        const auto PlayerState = Cast<AGAEPlayerState>(Controller->PlayerState);
        if (!PlayerState) continue;
        PlayerState->LogInfo();
    }
}
void AGAEGameModeBase::StartRespawn(AController* Controller)
{
    const auto RespawnAvailable = RoundCountDown > MinRoundTimeForRespawn
+ GameData.RespawnTime;
    if (!RespawnAvailable) return;
```

```
    const auto RespawnComponent =
GAEUtils::GetGAEPlayerComponent<UGAERespawnComponent>(Controller);
    if (!RespawnComponent) return;
    RespawnComponent->Respawn(GameData.RespawnTime);
}
void AGAEGameModeBase::RespawnRequest(AController* Controller)
{
    ResetOnePlayer(Controller);
}
void AGAEGameModeBase::GameOver()
{
    UE_LOG(LogGAEGameModeBase, Display, TEXT("- GAME OVER -"));
    LogPlayerInfo();
    for (auto Pawn : TActorRange<APawn>(GetWorld()))
    {
      if (Pawn)
      {
        Pawn->TurnOff();
        Pawn->DisableInput(nullptr);
      }
    }
    SetMatchState(EGAEMatchState::GameOver);
}
void AGAEGameModeBase::SetMatchState(EGAEMatchState State)
{
    if (MatchState == State) return;
    MatchState = State;
    OnMatchStateChanged.Broadcast(MatchState);
```

```
}
bool AGAEGameModeBase::SetPause(APlayerController* PC, FCanUnpause
CanUnpauseDelegate)
{
   const auto PauseSet = Super::SetPause(PC, CanUnpauseDelegate);
   if (PauseSet)
   {
      StopAllFire();
      SetMatchState(EGAEMatchState::Pause);
   }
   return PauseSet;
}
bool AGAEGameModeBase::ClearPause()
{
   const auto PauseCleared = Super::ClearPause();
   if (PauseCleared)
   {
      SetMatchState(EGAEMatchState::InProgress);
   }
   return PauseCleared;
}
void AGAEGameModeBase::StopAllFire()
{
   for (auto Pawn : TActorRange<APawn>(GetWorld()))
   {
      const auto WeaponComponent =
GAEUtils::GetGAEPlayerComponent<UGAEWeaponComponent>(Pawn);
      if (!WeaponComponent) continue;
      WeaponComponent->StopFire();
```

```
        WeaponComponent->Zoom(false);
    }
}
```

Файл GAEGameModeBase.h

```
#include "CoreMinimal.h"
#include "GameFramework/GameModeBase.h"
#include "GAECoreTypes.h"
#include "GAEGameModeBase.generated.h"

class AAIController;
UCLASS()
class GUNSANDENEMIES_API AGAEGameModeBase : public AGameModeBase
{
    GENERATED_BODY()
public:
    AGAEGameModeBase();
    FOnMatchStateChangedSignature OnMatchStateChanged;
    virtual void StartPlay() override;
    virtual UClass* GetDefaultPawnClassForController_Implementation(AController*
InController) override;
    void Killed(AController* KillerController, AController* VictimController);
    FGameData GetGameData() const { return GameData; }
    int32 GetCurrentRoundNum() const { return CurrentRound; }
    int32 GetRoundSecondsRemaining() const { return RoundCountDown; }
    void RespawnRequest(AController* Controller);

    virtual bool SetPause(APlayerController* PC, FCanUnpause CanUnpauseDelegate
= FCanUnpause()) override;
```

```
    virtual bool ClearPause() override;
protected:
    UPROPERTY(EditDefaultsOnly, Category = "Game")
    TSubclassOf<AAIController> AIControllerClass;
    UPROPERTY(EditDefaultsOnly, Category = "Game")
    TSubclassOf<APawn> AIPawnClass;
    UPROPERTY(EditDefaultsOnly, Category = "Game")
    FGameData GameData;
private:
    EGAEMatchState MatchState = EGAEMatchState::WaitingToStart;
    int32 CurrentRound = 1;
    int32 RoundCountDown = 0;
    FTimerHandle GameRoundTimerHandle;
    void SpawnBots();
    void StartRound();
    void GameTimerUpdate();
    void ResetPlayers();
    void ResetOnePlayer(AController* Controller);
    void CreateTeamsInfo();
    FLinearColor DetermineColorByTeamID(int32 TeamID) const;
    void SetPlayerColor(AController* Controller);
    void LogPlayerInfo();
    void StartRespawn(AController* Controller);
    void GameOver();
    void SetMatchState(EGAEMatchState State);
    void StopAllFire();
};
```

Клас GAEBaseCharacter.cpp

```
#include "Player/GAEBaseCharacter.h"
#include "Components/GAECharacterMovementComponent.h"
#include "Components/GAEHealthComponent.h"
#include "Components/GAEWeaponComponent.h"
#include "Components/CapsuleComponent.h"
#include "GameFramework/Controller.h"
#include "Kismet/GameplayStatics.h"
#include "Sound/SoundCue.h"

DEFINE_LOG_CATEGORY_STATIC(LogBaseCharacter, All, All);
AGAEBaseCharacter::AGAEBaseCharacter(const FObjectInitializer& ObjInit) :
Super(ObjInit.SetDefaultSubobjectClass<UGAECharacterMovementComponent>(A
Character::CharacterMovementComponentName))
{
   PrimaryActorTick.bCanEverTick = true;
   HealthComponent =
CreateDefaultSubobject<UGAEHealthComponent>("HealthComponent");
   WeaponComponent =
CreateDefaultSubobject<UGAEWeaponComponent>("WeaponComponent");
}
void AGAEBaseCharacter::BeginPlay()
{
   Super::BeginPlay();
   check(HealthComponent);
   check(GetCharacterMovement());
   check(GetCapsuleComponent());
   check(GetMesh());
```

```
    OnHealthChanged(HealthComponent->GetHealth(), 0.0f);
    HealthComponent->OnDeath.AddUObject(this,
&AGAEBaseCharacter::OnDeath);
    HealthComponent->OnHealthChanged.AddUObject(this,
&AGAEBaseCharacter::OnHealthChanged);
    LandedDelegate.AddDynamic(this, &AGAEBaseCharacter::OnGroundLanded);
}
void AGAEBaseCharacter::OnHealthChanged(float Health, float HealthDelta) {}
void AGAEBaseCharacter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}
bool AGAEBaseCharacter::IsRunning() const
{
    return false;
}
float AGAEBaseCharacter::GetMovementDirection() const
{
    if (GetVelocity().IsZero()) return 0.0f;
    const auto VelocityNormal = GetVelocity().GetSafeNormal();
    const auto AngleBetween =
FMath::Acos(FVector::DotProduct(GetActorForwardVector(), VelocityNormal));
    const auto CrossProduct = FVector::CrossProduct(GetActorForwardVector(),
VelocityNormal);
    const auto Degrees = FMath::RadiansToDegrees(AngleBetween);
    return CrossProduct.IsZero() ? Degrees : Degrees * FMath::Sign(CrossProduct.Z);
}
void AGAEBaseCharacter::OnDeath()
{
```

```
    UE_LOG(LogBaseCharacter, Display, TEXT("Player %s is dead"), *GetName());
    // PlayAnimMontage(DeathAnimMontage);
    GetCharacterMovement()->DisableMovement();
    SetLifeSpan(LifeSpanOnDeath);
    GetCapsuleComponent()->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
    WeaponComponent->StopFire();
    WeaponComponent->Zoom(false);
    GetMesh()->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
    GetMesh()->SetSimulatePhysics(true);
    UGameplayStatics::PlaySoundAtLocation(GetWorld(), DeathSound, GetActorLocation());
}
void AGAEBaseCharacter::OnGroundLanded(const FHitResult& Hit)
{
    const auto FallVelocityZ = -GetVelocity().Z;
    if (FallVelocityZ < LandedDamageVelocity.X) return;

    const auto FallDamage = FMath::GetMappedRangeValueClamped(LandedDamageVelocity, LandedDamage, FallVelocityZ);
    TakeDamage(FallDamage, FDamageEvent{}, nullptr, nullptr);
    UE_LOG(LogBaseCharacter, Display, TEXT("Player %s recived landed damage: %f"), *GetName(), FallDamage);
}
void AGAEBaseCharacter::SetPlayerColor(const FLinearColor& Color)
{
    const auto MaterialInst = GetMesh()->CreateAndSetMaterialInstanceDynamic(0);
    if (!MaterialInst) return;
    MaterialInst->SetVectorParameterValue(MaterialColorName, Color);
```

```
}
void AGAEBaseCharacter::TurnOff()
{
   WeaponComponent->StopFire();
   WeaponComponent->Zoom(false);
   Super::TurnOff();
}
void AGAEBaseCharacter::Reset()
{
   WeaponComponent->StopFire();
   WeaponComponent->Zoom(false);
   Super::Reset();
}
```

Клас GAEPlayerController.cpp

```
#include "Player/GAEPlayerController.h"
#include "Components/GAERespawnComponent.h"
#include "GAEGameModeBase.h"
#include "GAEGameInstance.h"

AGAEPlayerController::AGAEPlayerController()
{
   RespawnComponent =
CreateDefaultSubobject<UGAERespawnComponent>("RespawnComponent");
}
void AGAEPlayerController::BeginPlay()
{
   Super::BeginPlay();
```

```
    if (GetWorld())
    {
        if (const auto GameMode = Cast<AGAEGameModeBase>(GetWorld()->GetAuthGameMode()))
        {
            GameMode->OnMatchStateChanged.AddUObject(this, &AGAEPlayerController::OnMatchStateChanged);
        }
    }
}
void AGAEPlayerController::OnMatchStateChanged(EGAEMatchState State)
{
    if (State == EGAEMatchState::InProgress)
    {
        SetInputMode(FInputModeGameOnly());
        bShowMouseCursor = false;
    }
    else
    {
        SetInputMode(FInputModeUIOnly());
        bShowMouseCursor = true;
    }
}
void AGAEPlayerController::OnPossess(APawn* InPawn)
{
    Super::OnPossess(InPawn);
    OnNewPawn.Broadcast(InPawn);
}
```

```
void AGAEPlayerController::SetupInputComponent()
{
   Super::SetupInputComponent();
   if (!InputComponent) return;
   InputComponent->BindAction("PauseGame", IE_Pressed, this,
&AGAEPlayerController::OnPauseGame);
   InputComponent->BindAction("Mute", IE_Pressed, this,
&AGAEPlayerController::OnMuteSound);
}
void AGAEPlayerController::OnPauseGame()
{
   if (!GetWorld() || !GetWorld()->GetAuthGameMode()) return;
   GetWorld()->GetAuthGameMode()->SetPause(this);
}
void AGAEPlayerController::OnMuteSound()
{
   if (!GetWorld()) return;
   const auto GAEGameInstace = GetWorld()-
>GetGameInstance<UGAEGameInstance>();
   if (!GAEGameInstace) return;

   GAEGameInstace->ToggleVolume();
}
```

Клас GAEHealthComponent.cpp

```
#include "Components/GAEHealthComponent.h"
#include "GameFramework/Character.h"
#include "GameFramework/Controller.h"
```

```
#include "Engine/World.h"
#include "TimerManager.h"
#include "Camera/CameraShake.h"
#include "GAEGameModeBase.h"
#include "PhysicalMaterials/PhysicalMaterial.h"
#include "Perception/AISense_Damage.h"

DEFINE_LOG_CATEGORY_STATIC(LogHealthComponent, All, All);
UGAEHealthComponent::UGAEHealthComponent()
{
    PrimaryComponentTick.bCanEverTick = false;
}
void UGAEHealthComponent::BeginPlay()
{
    Super::BeginPlay();
    check(MaxHealth > 0);
    SetHealth(MaxHealth);
    AActor* ComponentOwner = GetOwner();
    if (ComponentOwner)
    {
        ComponentOwner->OnTakeAnyDamage.AddDynamic(this,
&UGAEHealthComponent::OnTakeAnyDamage);
        ComponentOwner->OnTakePointDamage.AddDynamic(this,
&UGAEHealthComponent::OnTakePointDamage);
        ComponentOwner->OnTakeRadialDamage.AddDynamic(this,
&UGAEHealthComponent::OnTakeRadialDamage);
    }
}
```

```
void UGAEHealthComponent::OnTakePointDamage(AActor* DamagedActor, float Damage, class AController* InstigatedBy, FVector HitLocation,
    class UPrimitiveComponent* FHitComponent, FName BoneName, FVector ShotFromDirection, const class UDamageType* DamageType,
    AActor* DamageCauser)
{
    const auto FinalDamage = Damage * GetPointDamageModifier(DamagedActor, BoneName);
    // UE_LOG(LogHealthComponent, Display, TEXT("On point damage: %f, final damage: %f, bone: %s"), Damage, FinalDamage, *BoneName.ToString());
    ApplyDamage(FinalDamage, InstigatedBy);
}
void UGAEHealthComponent::OnTakeRadialDamage(AActor* DamagedActor, float Damage, const class UDamageType* DamageType, FVector Origin,
    FHitResult HitInfo, class AController* InstigatedBy, AActor* DamageCauser)
{
    // UE_LOG(LogHealthComponent, Display, TEXT("On radial damage: %f"), Damage);
    ApplyDamage(Damage, InstigatedBy);
}
void UGAEHealthComponent::OnTakeAnyDamage(
    AActor* DamagedActor, float Damage, const class UDamageType* DamageType, class AController* InstigatedBy, AActor* DamageCauser)
{
    // UE_LOG(LogHealthComponent, Display, TEXT("On any damage: %f"), Damage);
}
void UGAEHealthComponent::ApplyDamage(float Damage, AController* InstigatedBy)
{
    if (Damage <= 0.0f || IsDead() || !GetWorld()) return;
```

```
    SetHealth(Health - Damage);
    GetWorld()->GetTimerManager().ClearTimer(HealTimerHandle);
    if (IsDead())
    {
        Killed(InstigatedBy);
        OnDeath.Broadcast();
    }
    else if (AutoHeal)
    {
        GetWorld()->GetTimerManager().SetTimer(HealTimerHandle, this,
&UGAEHealthComponent::HealUpdate, HealUpdateTime, true, HealDelay);
    }
    PlayCameraShake();
    ReportDamageEvent(Damage, InstigatedBy);
}
void UGAEHealthComponent::HealUpdate()
{
    SetHealth(Health + HealModifier);
    if (IsHealthFull() && GetWorld())
    {
        GetWorld()->GetTimerManager().ClearTimer(HealTimerHandle);
    }
}
void UGAEHealthComponent::SetHealth(float NewHealth)
{
    const auto NextHealth = FMath::Clamp(NewHealth, 0.0f, MaxHealth);
    const auto HealthDelta = NextHealth - Health;
```

```
    Health = NextHealth;
    OnHealthChanged.Broadcast(Health, HealthDelta);
}
bool UGAEHealthComponent::TryToAddHealth(float HealthAmount)
{
    if (IsDead() || IsHealthFull()) return false;
    SetHealth(Health + HealthAmount);
    return true;
}

bool UGAEHealthComponent::IsHealthFull() const
{
    return FMath::IsNearlyEqual(Health, MaxHealth);
}
void UGAEHealthComponent::PlayCameraShake()
{
    if (IsDead()) return;
    const auto Player = Cast<APawn>(GetOwner());
    if (!Player) return;
    const auto Controller = Player->GetController<APlayerController>();
    if (!Controller || !Controller->PlayerCameraManager) return;
    Controller->PlayerCameraManager->StartCameraShake(CameraShake);
}
void UGAEHealthComponent::Killed(AController* KillerController)
{
    if (!GetWorld()) return;
```

```
    const auto GameMode = Cast<AGAEGameModeBase>(GetWorld()->GetAuthGameMode());
    if (!GameMode) return;
    const auto Player = Cast<APawn>(GetOwner());
    const auto VictimController = Player ? Player->Controller : nullptr;
    GameMode->Killed(KillerController, VictimController);
}
float UGAEHealthComponent::GetPointDamageModifier(AActor* DamagedActor, const FName& BoneName)
{
    const auto Character = Cast<ACharacter>(DamagedActor);
    if (!Character ||           //
        !Character->GetMesh() ||  //
        !Character->GetMesh()->GetBodyInstance(BoneName))
        return 1.0f;
    const auto PhysMaterial = Character->GetMesh()->GetBodyInstance(BoneName)->GetSimplePhysicalMaterial();
    if (!PhysMaterial || !DamageModifiers.Contains(PhysMaterial)) return 1.0f;
    return DamageModifiers[PhysMaterial];
}
void UGAEHealthComponent::ReportDamageEvent(float Damage, AController* InstigatedBy)
{
    if (!InstigatedBy || !InstigatedBy->GetPawn() || !GetOwner()) return;
    UAISense_Damage::ReportDamageEvent(GetWorld(),    //
        GetOwner(),                          //
        InstigatedBy->GetPawn(),                  //
        Damage,                              //
        InstigatedBy->GetPawn()->GetActorLocation(),  //
```

```
        GetOwner()->GetActorLocation());
}
```

Клас GAEWeaponComponent.cpp

```
#include "Components/GAEWeaponComponent.h"
#include "Weapon/GAEBaseWeapon.h"
#include "GameFramework/Character.h"
#include "Animations/GAEEquipFinishedAnimNotify.h"
#include "Animations/GAEReloadFinishedAnimNotify.h"
#include "Animations/AnimUtils.h"

DEFINE_LOG_CATEGORY_STATIC(LogWeaponComponent, All, All);
constexpr static int32 WeaponNum = 2;
UGAEWeaponComponent::UGAEWeaponComponent()
{
    PrimaryComponentTick.bCanEverTick = false;
}
void UGAEWeaponComponent::BeginPlay()
{
    Super::BeginPlay();
    checkf(WeaponData.Num() == WeaponNum, TEXT("Our character can hold only
%i weapon items"), WeaponNum);
    CurrentWeaponIndex = 0;
    InitAnimations();
    SpawnWeapons();
    EquipWeapon(CurrentWeaponIndex);
}
void UGAEWeaponComponent::EndPlay(const EEndPlayReason::Type
EndPlayReason)
```

```
{
    CurrentWeapon = nullptr;
    for (auto Weapon : Weapons)
    {
        Weapon-
>DetachFromActor(FDetachmentTransformRules::KeepWorldTransform);
        Weapon->Destroy();
    }
    Weapons.Empty();
    Super::EndPlay(EndPlayReason);
}
void UGAEWeaponComponent::SpawnWeapons()
{
    ACharacter* Character = Cast<ACharacter>(GetOwner());
    if (!Character || !GetWorld()) return;
    for (auto OneWeaponData : WeaponData)
    {
        auto Weapon = GetWorld()-
>SpawnActor<AGAEBaseWeapon>(OneWeaponData.WeaponClass);
        if (!Weapon) continue;
        Weapon->OnClipEmpty.AddUObject(this,
&UGAEWeaponComponent::OnClipEmpty);
        Weapon->SetOwner(Character);
        Weapons.Add(Weapon);
        AttachWeaponToSocket(Weapon, Character->GetMesh(),
WeaponArmorySocketName);
    }
}
void UGAEWeaponComponent::AttachWeaponToSocket(AGAEBaseWeapon*
Weapon, USceneComponent* SceneComponent, const FName& SocketName)
```

```
{
    if (!Weapon || !SceneComponent) return;

    FAttachmentTransformRules AttachmentRules(EAttachmentRule::SnapToTarget, false);
    Weapon->AttachToComponent(SceneComponent, AttachmentRules, SocketName);
}
void UGAEWeaponComponent::EquipWeapon(int32 WeaponIndex)
{
    if (WeaponIndex < 0 || WeaponIndex >= Weapons.Num())
    {
        UE_LOG(LogWeaponComponent, Warning, TEXT("Invalid weapon index"));
        return;
    }
    ACharacter* Character = Cast<ACharacter>(GetOwner());
    if (!Character) return;
    if (CurrentWeapon)
    {
        CurrentWeapon->Zoom(false);
        CurrentWeapon->StopFire();
        AttachWeaponToSocket(CurrentWeapon, Character->GetMesh(), WeaponArmorySocketName);
    }
    CurrentWeapon = Weapons[WeaponIndex];
    // CurrentReloadAnimMontage = WeaponData[WeaponIndex].ReloadAnimMontage;
    const auto CurrentWeaponData = WeaponData.FindByPredicate([&](const FWeaponData& Data) {  //
```

```
        return Data.WeaponClass == CurrentWeapon->GetClass();
//
    });
    CurrentReloadAnimMontage = CurrentWeaponData ? CurrentWeaponData->ReloadAnimMontage : nullptr;
    AttachWeaponToSocket(CurrentWeapon, Character->GetMesh(), WeaponEquipSocketName);
    EquipAnimInProgress = true;
    PlayAnimMontage(EquipAnimMontage);
}
void UGAEWeaponComponent::StartFire()
{
    if (!CanFire()) return;
    CurrentWeapon->StartFire();
}
void UGAEWeaponComponent::StopFire()
{
    if (!CurrentWeapon) return;
    CurrentWeapon->StopFire();
}
void UGAEWeaponComponent::NextWeapon()
{
    if (!CanEquip()) return;
    CurrentWeaponIndex = (CurrentWeaponIndex + 1) % Weapons.Num();
    EquipWeapon(CurrentWeaponIndex);
}
void UGAEWeaponComponent::PlayAnimMontage(UAnimMontage* Animation)
{
    ACharacter* Character = Cast<ACharacter>(GetOwner());
```

```
    if (!Character) return;
    Character->PlayAnimMontage(Animation);
}
void UGAEWeaponComponent::InitAnimations()
{
    auto EquipFinishedNotify = AnimUtils::FindNotifyByClass<UGAEEquipFinishedAnimNotify>(EquipAnimMontage);
    if (EquipFinishedNotify)
    {
        EquipFinishedNotify->OnNotified.AddUObject(this, &UGAEWeaponComponent::OnEquipFinished);
    }
    else
    {
        UE_LOG(LogWeaponComponent, Error, TEXT("Equip anim notify is forgotten to set"));
        checkNoEntry();
    }
    for (auto OneWeaponData : WeaponData)
    {
        auto ReloadFinishedNotify = AnimUtils::FindNotifyByClass<UGAEReloadFinishedAnimNotify>(OneWeaponData.ReloadAnimMontage);
        if (!ReloadFinishedNotify)
        {
            UE_LOG(LogWeaponComponent, Error, TEXT("Reload anim notify is forgotten to set"));
            checkNoEntry();
        }
```

```
        ReloadFinishedNotify->OnNotified.AddUObject(this,
&UGAEWeaponComponent::OnReloadFinished);
    }
}
void UGAEWeaponComponent::OnEquipFinished(USkeletalMeshComponent*
MeshComp)
{
    ACharacter* Character = Cast<ACharacter>(GetOwner());
    if (!Character || MeshComp != Character->GetMesh()) return;

    EquipAnimInProgress = false;
}
void UGAEWeaponComponent::OnReloadFinished(USkeletalMeshComponent*
MeshComp)
{
    ACharacter* Character = Cast<ACharacter>(GetOwner());
    if (!Character || MeshComp != Character->GetMesh()) return;

    ReloadAnimInProgress = false;
}
bool UGAEWeaponComponent::CanFire() const
{
    return CurrentWeapon && !EquipAnimInProgress && !ReloadAnimInProgress;
}
bool UGAEWeaponComponent::CanEquip() const
{
    return !EquipAnimInProgress && !ReloadAnimInProgress;
}
bool UGAEWeaponComponent::CanReload() const
```

```
{
   return CurrentWeapon            //
         && !EquipAnimInProgress   //
         && !ReloadAnimInProgress  //
         && CurrentWeapon->CanReload();
}
void UGAEWeaponComponent::Reload()
{
   ChangeClip();
}
void UGAEWeaponComponent::OnClipEmpty(AGAEBaseWeapon*
ClipEmptyWeapon)
{
   if (!ClipEmptyWeapon) return;
   if (CurrentWeapon == ClipEmptyWeapon)
   {
      ChangeClip();
   }
   else
   {
      for (const auto Weapon : Weapons)
      {
         if (Weapon == ClipEmptyWeapon)
         {
            Weapon->ChangeClip();
         }
      }
   }
```

```
}
void UGAEWeaponComponent::ChangeClip()
{
    if (!CanReload()) return;
    CurrentWeapon->StopFire();
    CurrentWeapon->ChangeClip();
    ReloadAnimInProgress = true;
    PlayAnimMontage(CurrentReloadAnimMontage);
}
bool UGAEWeaponComponent::GetCurrentWeaponUIData(FWeaponUIData&
UIData) const
{
    if (CurrentWeapon)
    {
        UIData = CurrentWeapon->GetUIData();
        return true;
    }
    return false;
}
bool UGAEWeaponComponent::GetCurrentWeaponAmmoData(FAmmoData&
AmmoData) const
{
    if (CurrentWeapon)
    {
        AmmoData = CurrentWeapon->GetAmmoData();
        return true;
    }
    return false;
}
```

```
bool
UGAEWeaponComponent::TryToAddAmmo(TSubclassOf<AGAEBaseWeapon>
WeaponType, int32 ClipsAmount)
{
   for (const auto Weapon : Weapons)
   {
      if (Weapon && Weapon->IsA(WeaponType))
      {
         return Weapon->TryToAddAmmo(ClipsAmount);
      }
   }
   return false;
}

bool UGAEWeaponComponent::NeedAmmo(TSubclassOf<AGAEBaseWeapon>
WeaponType)
{
   for (const auto Weapon : Weapons)
   {
      if (Weapon && Weapon->IsA(WeaponType))
      {
         return !Weapon->IsAmmoFull();
      }
   }
   return false;
}
void UGAEWeaponComponent::Zoom(bool Enabled)
{
   if (CurrentWeapon)
```

```
    {
        CurrentWeapon->Zoom(Enabled);
    }
}
```

Клас GAEPlayerHUDWidget.cpp

```
#include "UI/GAEPlayerHUDWidget.h"
#include "Components/GAEHealthComponent.h"
#include "Components/GAEWeaponComponent.h"
#include "GAEUtils.h"
#include "Components/ProgressBar.h"
#include "Player/GAEPlayerState.h"

void UGAEPlayerHUDWidget::NativeOnInitialized()
{
    Super::NativeOnInitialized();
    if (GetOwningPlayer())
    {
        GetOwningPlayer()->GetOnNewPawnNotifier().AddUObject(this,
&UGAEPlayerHUDWidget::OnNewPawn);
        OnNewPawn(GetOwningPlayerPawn());
    }
}
void UGAEPlayerHUDWidget::OnNewPawn(APawn* NewPawn)
{
    const auto HealthComponent =
GAEUtils::GetGAEPlayerComponent<UGAEHealthComponent>(NewPawn);
    if (HealthComponent && !HealthComponent-
>OnHealthChanged.IsBoundToObject(this))
```

```
    {
        HealthComponent->OnHealthChanged.AddUObject(this,
&UGAEPlayerHUDWidget::OnHealthChanged);
    }
    UpdateHealthBar();
}
void UGAEPlayerHUDWidget::OnHealthChanged(float Health, float HealthDelta)
{
    if (HealthDelta < 0.0f)
    {
        OnTakeDamage();
        if (!IsAnimationPlaying(DamageAnimation))
        {
            PlayAnimation(DamageAnimation);
        }
    }
    UpdateHealthBar();
}
float UGAEPlayerHUDWidget::GetHealthPercent() const
{
    const auto HealthComponent =
GAEUtils::GetGAEPlayerComponent<UGAEHealthComponent>(GetOwningPlayer
Pawn());
    if (!HealthComponent) return 0.0f;
    return HealthComponent->GetHealthPercent();
}
bool UGAEPlayerHUDWidget::GetCurrentWeaponUIData(FWeaponUIData&
UIData) const
{
```

```
    const auto WeaponComponent =
GAEUtils::GetGAEPlayerComponent<UGAEWeaponComponent>(GetOwningPlaye
rPawn());
    if (!WeaponComponent) return false;
    return WeaponComponent->GetCurrentWeaponUIData(UIData);
}
bool UGAEPlayerHUDWidget::GetCurrentWeaponAmmoData(FAmmoData&
AmmoData) const
{
    const auto WeaponComponent =
GAEUtils::GetGAEPlayerComponent<UGAEWeaponComponent>(GetOwningPlaye
rPawn());
    if (!WeaponComponent) return false;
    return WeaponComponent->GetCurrentWeaponAmmoData(AmmoData);
}
bool UGAEPlayerHUDWidget::IsPlayerAlive() const
{
    const auto HealthComponent =
GAEUtils::GetGAEPlayerComponent<UGAEHealthComponent>(GetOwningPlayer
Pawn());
    return HealthComponent && !HealthComponent->IsDead();
}
bool UGAEPlayerHUDWidget::IsPlayerSpectating() const
{
    const auto Controller = GetOwningPlayer();
    return Controller && Controller->GetStateName() == NAME_Spectating;
}
int32 UGAEPlayerHUDWidget::GetKillsNum() const
{
    const auto Controller = GetOwningPlayer();
```

```
    if (!Controller) return 0;
    const auto PlayerState = Cast<AGAEPlayerState>(Controller->PlayerState);
    return PlayerState ? PlayerState->GetKillsNum() : 0;
}
void UGAEPlayerHUDWidget::UpdateHealthBar()
{
    if (HealthProgressBar)
    {
        HealthProgressBar->SetFillColorAndOpacity(GetHealthPercent() >
PercentColorThreshold ? GoodColor : BadColor);
    }
}
FString UGAEPlayerHUDWidget::FormatBullets(int32 BulletsNum) const
{
    const int32 MaxLen = 3;
    const TCHAR PrefixSymbol = '0';
    auto BulletStr = FString::FromInt(BulletsNum);
    const auto SymbolsNumToAdd = MaxLen - BulletStr.Len();
    if (SymbolsNumToAdd > 0)
    {
        BulletStr = FString::ChrN(SymbolsNumToAdd,
PrefixSymbol).Append(BulletStr);
    }
    return BulletStr;
}
```

Клас GAENewtLocTask.cpp

```
#include "AI/Tasks/GAENewtLocTask.h"
#include "BehaviorTree/BlackboardComponent.h"
```

```
#include "AIController.h"
#include "NavigationSystem.h"

UGAENextLocationTask::UGAENextLocationTask()
{
   NodeName = "Next Location";
}

EBTNodeResult::Type
UGAENextLocationTask::ExecuteTask(UBehaviorTreeComponent& OwnerComp,
uint8* NodeMemory)
{
   const auto Controller = OwnerComp.GetAIOwner();
   const auto Blackboard = OwnerComp.GetBlackboardComponent();
   if (!Controller || !Blackboard) return EBTNodeResult::Failed;

   const auto Pawn = Controller->GetPawn();
   if (!Pawn) return EBTNodeResult::Failed;

   const auto NavSys = UNavigationSystemV1::GetCurrent(Pawn);
   if (!NavSys) return EBTNodeResult::Failed;

   FNavLocation NavLocation;
   auto Location = Pawn->GetActorLocation();
   if (!SelfCenter)
   {
      auto CenterActor = Cast<AActor>(Blackboard-
>GetValueAsObject(CenterActorKey.SelectedKeyName));
      if (!CenterActor) return EBTNodeResult::Failed;
```

```
        Location = CenterActor->GetActorLocation();
    }
    const auto Found = NavSys->GetRandomReachablePointInRadius(Location,
Radius, NavLocation);
    if (!Found) return EBTNodeResult::Failed;
    Blackboard->SetValueAsVector(AimLocationKey.SelectedKeyName,
NavLocation.Location);
    return EBTNodeResult::Succeeded;
}
```